Piotr Rygielski

Flexible Modeling of Data Center Networks for Capacity Management

CC BY SA

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2017
Erster Gutachter: Prof. Dr.-Ing. Samuel Kounev
Zweiter Gutachter: Dr. rer. nat. Steffen Zschaler
Datum der Mündlicher Prüfung: 22.03.2017

Contents

Abstract xiii

Zusammenfassung xvii

Acknowledgements xxi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 5
  - 1.2.1 Flexible Performance Prediction . . . 5
  - 1.2.2 Formal View of the Problem . . . 10
- 1.3 Approach and Contributions . . . 12
  - 1.3.1 Primary Research Contributions . . . 14
  - 1.3.2 Secondary Research Contributions . . . 16
  - 1.3.3 Technical Contributions . . . 17
- 1.4 Application Areas . . . 19
- 1.5 Thesis Organization . . . 21

2 Foundations 23
- 2.1 Generic Data Center Network Virtualization . . . 23
  - 2.1.1 Link Virtualization . . . 24
  - 2.1.2 Virtual Network Appliances . . . 25
  - 2.1.3 Protocol Overlaying . . . 25
- 2.2 Network Virtualization Technologies . . . 27
  - 2.2.1 Software-Defined Networking . . . 27
  - 2.2.2 Network Function Virtualization . . . 30
- 2.3 Performance Modeling of Networks . . . 32
  - 2.3.1 Network Traffic Models . . . 32
  - 2.3.2 Black-Box and Gray-Box Performance Models . . . 33
  - 2.3.3 Simulation Approaches . . . 36
- 2.4 Run-Time and Design-Time Aspect of Performance Prediction . . . 37
- 2.5 Modeling Languages, Meta-Models and Descriptive Models . . . 40

3 Related Work 41
- 3.1 Related Areas . . . 41
  - 3.1.1 Performance Modeling of Data Center Networks (P1) . . . 42
  - 3.1.2 Performance Modeling of SDN-based Networks (P2) . . . 43
  - 3.1.3 Architecture-Level Modeling of Data Center Networks (A1) . 44
  - 3.1.4 Architecture-Level Modeling of SDN-based Networks (A2) . . 44

3.1.5 Architecture-Level Performance Modeling (F0) . . . 45
3.1.6 Architecture-Level Performance Modeling of Data Center Networks (F1) . . . 46
3.1.7 Architecture-Level Performance Modeling of SDN-based Networks (F2) . . . 46
3.2 Traffic Model Extraction . . . 46

4 Network Performance Abstractions 49
4.1 Modeling Classical Networks . . . 50
4.1.1 Network Structure . . . 52
4.1.2 Network Traffic . . . 56
4.1.3 Network Configuration . . . 57
4.2 Modeling SDN Networks . . . 59
4.2.1 Processing in an SDN Node . . . 59
4.2.2 Network Structure . . . 60
4.2.3 Network Traffic . . . 62
4.2.4 Network Configuration . . . 62
4.3 Flexibility of Modeling . . . 64
4.3.1 Flexibility in Building DNI Models . . . 64
4.3.2 miniDNI Meta-Model . . . 66
4.4 Integration of the DNI Abstractions with Descartes Modeling Language 67
4.4.1 Overview of Descartes Modeling Language . . . 68
4.4.2 Data Center Structure . . . 68
4.4.3 Data Center Applications . . . 69
4.4.4 Traffic Workload . . . 71
4.4.5 Network Deployment Meta-Model . . . 74
4.4.6 Example . . . 75
4.5 Summary . . . 81

5 Model Transformations and Solving 83
5.1 Model Parametrization and Validity Checking . . . 88
5.1.1 Model Validity Checking . . . 88
5.1.2 Transformation Parametrization . . . 89
5.1.3 In-Place DNI Transformations . . . 90
5.2 Steady-State Performance Analysis with Queueing Petri Nets . . . 92
5.2.1 QPN Notation . . . 92
5.2.2 Network Topology . . . 93
5.2.3 Node . . . 93
5.2.4 Virtual Nodes . . . 98
5.2.5 Traffic Source . . . 99
5.2.6 Routing Information . . . 102
5.2.7 QPN Colors and Traffic Clustering . . . 102
5.3 Abstracting DNI with miniDNI and Solving with QPN . . . 103
5.3.1 Transformation of DNI to miniDNI . . . 103
5.3.2 Transformation of miniDNI to QPN . . . 105

5.4 Solving DNI with Discrete Time Simulation . . . 107
5.4.1 Classical Network: *OMNeT++INET* . . . 107
5.4.2 SDN-based Network: *OMNeT++generic* Simulation . . . 110
5.4.3 Limitations of *OMNeT++*-based Solvers and their Transformations . . . 114
5.5 Layered Queueing Networks: Transformation and Solvers . . . 115
5.5.1 QPN and LQN Solvers and their Limitations . . . 115
5.5.2 QPN-to-LQN Transformation . . . 116
5.5.3 Transformation Limitations . . . 123
5.6 Selection of Optimal Solver . . . 126
5.6.1 Differences between Predictive Models and Solvers . . . 126
5.6.2 Optimal Solvers for Performance Prediction . . . 129
5.7 Summary . . . 132

6 Extraction and Calibration of the DNI Models 135
6.1 DNI Model Extraction . . . 135
6.2 Traffic Model Extraction . . . 137
6.2.1 Network Traffic Generator Model . . . 138
6.2.2 Traffic Model in DNI . . . 140
6.2.3 Approach based on Multi-Scale Decomposition . . . 141
6.3 DNI Model Calibration . . . 148
6.4 Summary . . . 153

7 Validation 155
7.1 Evaluation Goals . . . 155
7.1.1 Modeling Capabilities . . . 157
7.1.2 Prediction Capabilities . . . 157
7.1.3 Flexibility of Performance Prediction . . . 158
7.2 Performance Prediction of Classical Networks . . . 158
7.2.1 Case Study: Event-oriented Message Bus . . . 159
7.2.2 Validation of DNI-to-*OMNeT++INET* Transformation . . . 160
7.2.3 Validation of the DNI and miniDNI QPN Transformations . . . 171
7.3 Performance Prediction of SDN-based Networks . . . 178
7.3.1 Case Study: Cloud Files Backup . . . 178
7.3.2 Hardware Testbed and Experiment Setup . . . 180
7.3.3 Modeling . . . 182
7.3.4 Scenario #4: Upgrading Hardware to SDN . . . 183
7.3.5 Scenario #5: Physical and Virtual Nodes . . . 190
7.3.6 Scenario #6: SDN Switch Misconfiguration . . . 193
7.3.7 Scenario #7: Network Load-balancing with ECMP . . . 196
7.3.8 Scenario #8: Server Load-balancing as Network Function . . . 198
7.4 Flexibility of Performance Prediction . . . 202
7.4.1 Model Solving Time for non-SDN DNI Models . . . 202
7.4.2 Solving Time and Memory Consumption for SDN-based DNI Models . . . 205

7.4.3 Discussion . . . 209
7.5 Traffic Model Extraction . . . 211
7.5.1 Robot Telemaintenance Case Study . . . 212
7.5.2 Traffic in the Telemaintenance Case Study . . . 212
7.5.3 Evaluating Model Compactness and Extraction Errors . . . 213
7.6 Transformation QPN-to-LQN and LQN Solvers . . . 216
7.6.1 Example #1: Simple QPN Model . . . 216
7.6.2 Example #2: SPECjAppServer2001 . . . 217
7.6.3 Analysis of Solving Time and Memory Consumption . . . 218
7.7 Summary . . . 220

8 Concluding Remarks **223**
8.1 Summary . . . 223
8.2 Open Challenges and Directions for Future Work . . . 227
8.2.1 Primary Research Directions . . . 227
8.2.2 Secondary Research Directions . . . 229
8.2.3 Technical Directions . . . 231

List of Figures **233**

List of Tables **239**

Acronyms **241**

Bibliography **245**

Publication List

Publications at University of Würzburg and Karlsruhe Institute of Technology in Years 2012–2016

Peer-Reviewed Journal and Magazine Articles

[MIK+16] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Diane E. Mularz, Jonathan A. Curtiss, Jason J. Ding, Florian Rosenberg, and Piotr Rygielski. CUP: A Formalism for Expressing Cloud Usage Patterns for Experts and Non-Experts. *IEEE Cloud Computing*, 2016. Paper accepted for publication.

[MRSK16] Christoph Müller, Piotr Rygielski, Simon Spinner, and Samuel Kounev. Enabling Fluid Analysis for Queueing Petri Nets via Model Transformation. *Electronic Notes in Theoretical Computer Science*, 327:71–91, 2016. The 8th International Workshop on Practical Application of Stochastic Modeling, (PASM 2016).

[RK13] Piotr Rygielski and Samuel Kounev. Network Virtualization for QoS-Aware Resource Management in Cloud Data Centers: A Survey. *PIK — Praxis der Informationsverarbeitung und Kommunikation*, 36(1):55–64, 2013.

Peer-Reviewed International Conference Contributions

Full Research Papers

[RSKK16] Piotr Rygielski, Marian Seliuchenko, Samuel Kounev, and Mykhailo Klymash. Performance Analysis of SDN Switches with Hardware and Software Flow Tables. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2016)*, October 2016. Paper accepted for publication.

[RSK16] Piotr Rygielski, Marian Seliuchenko, and Samuel Kounev. Modeling and Prediction of Software-Defined Networks Performance using Queueing Petri Nets. In *Proceedings of the Ninth International Conference on Simulation Tools and Techniques (SIMUTools 2016)*, August 2016.

[RSS+16] Piotr Rygielski, Viliam Simko, Felix Sittner, Doris Aschenbrenner, Samuel Kounev, and Klaus Schilling. Automated Extraction of Network Traffic Models Suitable for Performance Simulation. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*, pages 27–35. ACM, March 2016.

[RKTG15] Piotr Rygielski, Samuel Kounev, and Phuoc Tran-Gia. Flexible Performance Prediction of Data Center Networks using Automatically Generated Simulation Models. In *Proceedings of the Eighth International Conference on Simulation Tools and Techniques (SIMUTools 2015)*, pages 119–128, August 2015.

[LGS+15] Stanislav Lange, Steffen Gebert, Joachim Spoerhase, Piotr Rygielski, Thomas Zinner, Samuel Kounev, and Phuoc Tran-Gia. Specialized Heuristics for the Controller Placement Problem in Large Scale SDN Networks. In *27th International Teletraffic Congress (ITC 2015)*, pages 210–218, Ghent, Belgium, September 2015.

[RK14a] Piotr Rygielski and Samuel Kounev. Data Center Network Throughput Analysis using Queueing Petri Nets. In *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Workshops). 4th International Workshop on Data Center Performance, (DCPerf 2014)*, pages 100–105, June 2014.

[RKZ13] Piotr Rygielski, Samuel Kounev, and Steffen Zschaler. Model-Based Throughput Prediction in Data Center Networks. In *Proc. of the 2nd IEEE Int. Workshop on Measurements and Networking*, pages 167–172, 2013.

Short/Work-in-progress Papers

[KBM+16] Samuel Kounev, Fabian Brosig, Philipp Meier, Steffen Becker, Anne Koziolek, Heiko Koziolek, and Piotr Rygielski. Analysis of the Trade-offs in Different Modeling Approaches for Performance Prediction of Software Systems. In *Software Engineering 2016 (SE 2016), Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. März 2016, Vienna, Austria*, Lecture Notes in Informatics (LNI), pages 47–48, Vienna, Austria, February 2016. GI.

[RZK13] Piotr Rygielski, Steffen Zschaler, and Samuel Kounev. A Meta-Model for Performance Modeling of Dynamic Virtualized Network Infrastructures. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, pages 327–330, New York, NY, USA, April 2013. ACM.

Research Grants

[MOD16] MODELS: Performance Modeling of Software-Defined Data Center Networks, 2016. Research grant Awarded by the German Research Foundation; Deutsche Forschungs-gemeinschaft (DFG).

The project proposal written for this grant, submitted with Prof. Dr. Samuel Kounev, is based on this thesis and captures future work directly related to the contributions of the thesis.

Technical Reports

[RK14b] Piotr Rygielski and Samuel Kounev. Descartes Network Infrastructures (DNI) Manual: Meta-models, Transformations, Examples. Technical Report v.0.3, Chair of Software Engineering, University of Würzburg, Am Hubland, 97074 Würzburg, September 2014.

[MIK+13] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Piotr Rygielski, Jason Ding, Walfredo Cirne, and Florian Rosenberg. Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios. Technical Report SPEC-RG-2013-001 v.1.0.1, SPEC Research Group - Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, May 2013.

Publications at Wrocław Univeristy of Technology in Years 2009–2012

Peer-Reviewed Journal and Magazine Articles

[SRJG12] Pawel Świątek, Piotr Rygielski, Krzysztof Juszczyszyn, and Adam Grzech. User Assignment and Movement Prediction in Wireless Networks. *Cybernetics and Systems: An International Journal (CBS)*, 43(4):340–353, May 2012. 5-Year Impact Factor (2014): 0.968.

[BR11] Krzysztof Brzostowski and Piotr Rygielski. Automatyczna kompozycja uslug monitorowania dla systemu wspomagania treningu wytrzymalościowego sportowców. *Przegląd Telekomunikacyjny, Wiadomości Telekomunikacyjne*, 84:1017–1022, 2011. In Polish.

[GSR10c] Adam Grzech, Pawel Świątek, and Piotr Rygielski. Translations of service level agreement in systems based on service oriented architectures. *Cybernetics and Systems: An International Journal (CBS)*, 41(8):610–627, November 2010. 5-Year Impact Factor (2014): 0.968.

[GSR10a] Adam Grzech, Pawel Świątek, and Piotr Rygielski. Adaptive Packet

Scheduling for Requests Delay Guaranties in Packet-Switched Communication Network. *Systems Science*, 36(1):7–12, 2010.

[RS10a] Piotr Rygielski and Pawel Świątek. Graph-fold: an Efficient Method for Complex Service Execution Plan Optimization. *Systems Science*, 36(3):25–32, 2010.

Peer-Reviewed International Conference Contributions

Full Research Papers

[SR11] Paweł Świątek and Piotr Rygielski. Universal Communication Platform for QoS-aware Delivery of Complex Services. In *Proceedings of the VIth International Scientific and Technical Conference (CSIT 2011)*, pages 136–139. Publishing House Vezha&Co, November 2011.

[RSJG11] Piotr Rygielski, Pawel Świątek, Krzysztof Juszczyszyn, and Adam Grzech. Prediction Based Handovers for Wireless Networks Resources Management. In A. Koenig, A. Dengel, K. Hinkelmann, K. Kise, R.J. Howlett, and L.C. Jain, editors, *Knowledge-Based and Intelligent Information and Engineering Systems, Part II (KES 2011)*, volume 6882 of *Lecture Notes in Computer Science (LNCS)*, pages 687–696, Berlin, Heidelberg, September 2011. Springer-Verlag.

[RT11] Piotr Rygielski and Jakub Tomczak. Context Change Detection for Resource Allocation in Service-Oriented Systems. In A. Koenig, A. Dengel, K. Hinkelmann, K. Kise, R.J. Howlett, and L.C. Jain, editors, *Knowledge-Based and Intelligent Information and Engineering Systems, Part II (KES 2011)*, volume 6882 of *Lecture Notes in Computer Science (LNCS)*, pages 591–600, Berlin, Heidelberg, September 2011. Springer-Verlag.

[RG11] Piotr Rygielski and Adam Gonczarek. Migration-aware Optimization of Virtualized Computational Resources Allocation in Complex Systems. In Henry Selvaraj and Dawid Zydek, editors, *ICSEng 2011: Proceedings of the 21st international conference on Systems Engineering (ICSEng 2011)*, pages 212–216. IEEE Computer Society's Conference Publishing Services, August 2011.

[GSR10b] Adam Grzech, Pawel Świątek, and Piotr Rygielski. Dynamic Resources Allocation for Delivery of Personalized Services. In Wojciech Cellary and Elsa Estevez, editors, *Software Services for e-World*, volume 341 of *IFIP Advances in Information and Communication Technology*, pages 17–28. Springer, Boston, November 2010.

[RS10c] Piotr Rygielski and Paweł Świątek. Wireless Network Management Through Network-Enforced Handover. In Adam Grzech, Pawel Świątek, and

Krzysztof Brzostowski, editors, *Applications of Systems Science*, pages 227–236. Exit, ul. Maszynowa 5/9, 02-392 Warszawa, Poland, September 2010.

[GR10] Adam Grzech and Piotr Rygielski. Translations of Service Level Agreement in Systems Based on Service Oriented Architecture. In Rossitza Setchi and Ivan Jordanov, editors, *Proceedings of the 14th International Conference on Knowledge-based and Intelligent Information and Engineering Systems (KES 2010)*, volume 6277 of *Lecture Notes in Computer Science (LNCS)*, pages 523–532, Berlin, Heidelberg, September 2010. Springer-Verlag.

[RS10b] Piotr Rygielski and Paweł Świątek. Optimal Complex Services Composition in SOA Systems. In Adam Grzech, Pawel Świątek, and Krzysztof Brzostowski, editors, *Applications of Systems Science*, pages 217–226. Exit, ul. Maszynowa 5/9, 02-392 Warszawa, Poland, September 2010. Most Promising Student Paper Award.

[GRS10b] Adam Grzech, Piotr Rygielski, and Pawel Świątek. QoS-aware Infrastructure Resources Allocation in Systems Based on Service-Oriented Architecture Paradigm. In Tadeusz Czachórski, editor, *Performance modelling and evaluation of heterogeneous networks, 6th Working International Conference (HET-NETs 2010)*, pages 35–47, Baltycka 5, 44-100 Gliwice, Poland, January 2010. Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences.

[GRS10c] Adam Grzech, Piotr Rygielski, and Pawel Świątek. Simulation Environment for Delivering Quality of Service in Systems Based on Service-Oriented Architecture Paradigm. In Tadeusz Czachórski, editor, *Performance modeling and evaluation of heterogeneous networks, 6th Working International Conference (HET-NETs 2010)*, pages 89–97, Baltycka 5, 44-100 Gliwice, Poland, January 2010. Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences.

[GRS10a] Adam Grzech, Piotr Rygielski, and Pawel Świątek. QoS-aware Complex Service Composition in SOA-based Systems. In Stanislaw Ambroszkiewicz, Jerzy Brzeziński, Wojciech Cellary, Adam Grzech, and Krzysztof Zieliński, editors, *SOA Infrastructure Tools: Concepts and Methods*, pages 289–312. Poznań University of Economics Press, Poznań, Poland, 2010.

[GSR09b] Adam Grzech, Pawel Świątek, and Piotr Rygielski. Twostage Packet Scheduling in the Network Node. In Keith J. Burnham and Olivier C. L. Haas, editors, *Proceedings of Twentieth International Conference on Systems Engineering, (ICSE 2009)*, pages 181–184. Coventry University Press, September 2009.

[GSR09a] Adam Grzech, Pawel Świątek, and Piotr Rygielski. Improving QoS Guaranties via Adaptive Packet Scheduling. In *Proceedings of the 16th Polish Teletraffic Symposium (PTS 2009)*, pages 53–56. Technical University of Łódź, September 2009. Young Author Best Paper Award.

Abstract

Nowadays, data centers are becoming increasingly dynamic due to the common adoption of virtualization technologies. Systems can scale their capacity on demand by growing and shrinking their resources dynamically based on the current load. However, the complexity and performance of modern data centers is influenced not only by the software architecture, middleware, and computing resources, but also by network virtualization, network protocols, network services, and configuration. The field of network virtualization is not as mature as server virtualization and there are multiple competing approaches and technologies. Performance modeling and prediction techniques provide a powerful tool to analyze the performance of modern data centers. However, given the wide variety of network virtualization approaches, no common approach exists for modeling and evaluating the performance of virtualized networks.

The performance community has proposed multiple formalisms and models for evaluating the performance of infrastructures based on different network virtualization technologies. The existing performance models can be divided into two main categories: coarse-grained analytical models and highly-detailed simulation models. Analytical performance models are normally defined at a high level of abstraction and thus they abstract many details of the real network and therefore have limited predictive power. On the other hand, simulation models are normally focused on a selected networking technology and take into account many specific performance influencing factors, resulting in detailed models that are tightly bound to a given technology, infrastructure setup, or to a given protocol stack.

Existing models are *inflexible*, that means, they provide a single solution method without providing means for the user to influence the solution accuracy and solution overhead. To allow for *flexibility* in the performance prediction, the user is required to build multiple different performance models obtaining multiple performance predictions. Each performance prediction may then have different focus, different performance metrics, prediction accuracy, and solving time.

The goal of this thesis is to develop a modeling approach that does not require the user to have experience in any of the applied performance modeling formalisms. The approach offers the flexibility in the modeling and analysis by balancing between: (a) generic character and low overhead of coarse-grained analytical models, and (b) the more detailed simulation models with higher prediction accuracy.

The contributions of this thesis intersect with technologies and research areas, such as: software engineering, model-driven software development, domain-specific modeling, performance modeling and prediction, networking and data center

networks, network virtualization, Software-Defined Networking (SDN), Network Function Virtualization (NFV). The main contributions of this thesis compose the Descartes Network Infrastructure (DNI) approach and include:

- Novel modeling abstractions for virtualized network infrastructures. This includes two meta-models that define modeling languages for modeling data center network performance. The DNI and *miniDNI* meta-models provide means for representing network infrastructures at two different abstraction levels. Regardless of which variant of the DNI meta-model is used, the modeling language provides generic modeling elements allowing to describe the majority of existing and future network technologies, while at the same time abstracting factors that have low influence on the overall performance. I focus on SDN and NFV as examples of modern virtualization technologies.

- Network deployment meta-model—an interface between DNI and other meta-models that allows to define mapping between DNI and other descriptive models. The integration with other domain-specific models allows capturing behaviors that are not reflected in the DNI model, for example, software bottlenecks, server virtualization, and middleware overheads.

- Flexible model solving with model transformations. The transformations enable solving a DNI model by transforming it into a predictive model. The model transformations vary in size and complexity depending on the amount of data abstracted in the transformation process and provided to the solver. In this thesis, I contribute *six* transformations that transform DNI models into various predictive models based on the following modeling formalisms: (a) *OMNeT++* simulation, (b) Queueing Petri Nets (QPNs), (c) Layered Queueing Networks (LQNs). For each of these formalisms, multiple predictive models are generated (e.g., models with different level of detail): (a) two for *OMNeT++*, (b) two for QPNs, (c) two for LQNs. Some predictive models can be solved using multiple alternative solvers resulting in up to *ten* different automated solving methods for a single DNI model.

- A model extraction method that supports the modeler in the modeling process by automatically prefilling the DNI model with the network traffic data. The contributed traffic profile abstraction and optimization method provides a trade-off by balancing between the size and the level of detail of the extracted profiles.

- A method for selecting feasible solving methods for a DNI model. The method proposes a set of solvers based on trade-off analysis characterizing each transformation with respect to various parameters such as its specific limitations, expected prediction accuracy, expected run-time, required resources in terms of CPU and memory consumption, and scalability.

- An evaluation of the approach in the context of two realistic systems. I evaluate the approach with focus on such factors like: prediction of network

capacity and interface throughput, applicability, flexibility in trading-off between prediction accuracy and solving time. Despite not focusing on the maximization of the prediction accuracy, I demonstrate that in the majority of cases, the prediction error is low—up to 20% for uncalibrated models and up to 10% for calibrated models depending on the solving technique.

In summary, this thesis presents the first approach to flexible run-time performance prediction in data center networks, including network based on SDN. It provides ability to flexibly balance between performance prediction accuracy and solving overhead. The approach provides the following key benefits:

- It is possible to predict the impact of changes in the data center network on the performance. The changes include: changes in network topology, hardware configuration, traffic load, and applications deployment.
- DNI can successfully model and predict the performance of multiple different of network infrastructures including proactive SDN scenarios.
- The prediction process is flexible, that is, it provides balance between the granularity of the predictive models and the solving time. The decreased prediction accuracy is usually rewarded with savings of the solving time and consumption of resources required for solving.
- The users are enabled to conduct performance analysis using multiple different prediction methods without requiring the expertise and experience in each of the modeling formalisms.

The components of the DNI approach can be also applied to scenarios that are not considered in this thesis. The approach is generalizable and applicable for the following examples: (a) networks outside of data centers may be analyzed with DNI as long as the background traffic profile is known; (b) uncalibrated DNI models may serve as a basis for design-time performance analysis; (c) the method for extracting and compacting of traffic profiles may be used for other, non-network workloads as well.

Zusammenfassung

Durch Virtualisierung werden moderne Rechenzentren immer dynamischer. Systeme sind in der Lage ihre Kapazität hoch und runter zu skalieren , um die ankommende Last zu bedienen. Die Komplexität der modernen Systeme in Rechenzentren wird nicht nur von der Softwarearchitektur, Middleware und Rechenressourcen sondern auch von der Netzwerkvirtualisierung beeinflusst. Netzwerkvirtualisierung ist noch nicht so ausgereift wie die Virtualisierung von Rechenressourcen und es existieren derzeit unterschiedliche Netzwerkvirtualisierungstechnologien. Man kann aber keine der Technologien als Standardvirtualisierung für Netzwerke bezeichnen. Die Auswahl von Ansätzen durch Performanzanalyse von Netzwerken stellt eine Herausforderung dar, weil existierende Ansätze sich mehrheitlich auf einzelne Virtualisierungstechniken fokussieren und es keinen universellen Ansatz für Performanzanalyse gibt, der alle Techniken in Betracht nimmt.

Die Forschungsgemeinschaft bietet verschiedene Performanzmodelle und Formalismen für Evaluierung der Performanz von virtualisierten Netzwerken an. Die bekannten Ansätze können in zwei Gruppen aufgegliedert werden: Grob-detaillierte analytische Modelle und fein-detaillierte Simulationsmodelle. Die analytischen Performanzmodelle abstrahieren viele Details und liefern daher nur beschränkt nutzbare Performanzvorhersagen. Auf der anderen Seite fokussiert sich die Gruppe der simulationsbasierenden Modelle auf bestimmte Teile des Systems (z.B. Protokoll, Typ von Switches) und ignoriert dadurch das große Bild der Systemlandschaft.

Darüber hinaus sind die existierende Ansätze *inflexibel* – mit anderen Worten – sie bieten nur ein einzelnes Lösungsverfahren an, ohne die Genauigkeit oder den Lösungsaufwand beeinflussen zu können. Um *Flexibilität* anzubieten, wird der Benutzer gezwungen, mehrere verschiedene Performanzmodelle zu bauen, die auf verschiedenen Lösungsansätzen basieren. Jedes Performanzmodell bietet verschiedene Performanzmetriken, Schwerpunkte, Analysegenauigkeit, und Lösungsdauer an.

Als Ziel dieser Dissertation wird ein Modellierungsansatz vorgeschlagen, der mehrere verschiedene Performanzmodelle anbietet, ohne von dem Benutzer die Expertise in jedem Formalismus zu verlangen. Der Ansatz unterstützt die Flexibilität durch die Möglichkeit des automatischen Balancierens zwischen: (a) Dem generischen Charakter mit niedrigen Lösungsaufwand von analytischen Modellen, und (b) den detaillierten Simulationsmodellen, die bessere Genauigkeit der Vorhersage anbieten.

Die Beiträge der Dissertation schneiden Forschungsgebiete wie: Softwareentwurf, Modell-basierte Softwareentwicklung, Domänen-spezifische Modellierungssprachen, Modellieren und Vorhersage der Performanz, Netzwerke in Rechenzentren und Netzwerkprotokolle, Software-Defined Networking (SDN), Network Function Vir-

tualization (NFV). Die Hauptbeitrag der Dissertation stellt der DNI-Ansatz dar, welcher die folgenden Teile beinhaltet:

- Neuartige Modellierungsabstraktionen für virtualisierte Netzwerkinfrastrukturen. Das schließt zwei Meta-Modelle ein, die zwei Modellierungssprachen für Performanz der Netzwerke definieren. Die Descartes Network Infrastructure (DNI) und *miniDNI* Meta-Modelle bieten vielfältige Modellierungsmöglichkeiten, um die Netzwerkinfrastrukturen auf verschiedenen Abstraktionsgraden darzustellen. Unabhängig von der Variante der Sprache, bietet die Modellierungssprache generische Elemente, die die Mehrheit von existierenden und zukünftigen Netzwerken abdecken. Zeitgleich abstrahieren die Modelle solche Faktoren, die wenig Einfluss auf der Performanz des Systems haben. In der Dissertation, werden SDN und NFV als repräsentative Beispiele von Netzwerksvirtualisierungstechnologien herangezogen.

- Das Netzwerk deployment Meta-Modell stellt eine Schnittstelle zwischen DNI und anderen deskriptiven Modellierungssprachen dar. Es ermöglicht die Integration mit den Modellen und unterstützt damit die Modellierungsdomäne anderer Gebiete, z.B. der Softwarearchitektur, der Servervirtualisierung, und der Betriebssystem-Overheads.

- Flexibles Lösen der Modelle mit Hilfe von Modelltransformationen. Die Modelltransformationen ermöglichen das Lösen von den deskriptiven DNI Modellen mittels Übersetzung des Modells in ein prädiktives Modell, das mit existierenden Lösungsverfahren gelöst werden kann. Die Modelltransformationen unterscheiden sich vor einander hinsichtlich von der Komplexität und der Menge an in der Transformation abstrahierte Daten. Im Rahmen der Dissertation trage ich *sechs* Modelltransformationen bei, die die DNI Modelle in verschiedene prädiktive Modelle übersetzen. Die unterstützten prädiktiven Modelle basieren auf folgenden Formalismen: (a) *OMNeT++* Simulation, (b) Warteschlangen-Petri-Netze (QPNs), (c) Layered Queueing Networks (LQNs). Für jeden der oben genannten Formalismen werden mehrere prädiktive Modelle generiert, die sich von einander im Detailgrad unterscheiden. Es werden zwei Modelle jeweils für *OMNeT++*, QPN, und LQN generiert, die in Kombination mit verschiedenen Lösungsverfahren bis zu *zehn* verschiedene automatisierte Lösungsmethoden für ein DNI Modell anbieten können.

- Eine Methode für Modellextraktion, die den Modellierer beim Modellieren des Netzwerkverkehrs durch automatisches Vorausfüllen des Modells unterstützt. Die beigetragene Methode bietet eine Abstraktion für Netzwerkverkehrsmodelle an, die dem Benutzer eine Abwägung zwischen der Größe und dem Detailgrad des Modells erlaubt.

- Eine Methode zur Selektion eines optimalen Modelllösungsverfahrens. Basierend auf den von dem Benutzer gegebenen Präferenzen, schlägt die Methode ein Set von Lösungsverfahren vor. Die Lösungsverfahren werden dabei mit Fokus auf verschiedenen Aspekten analysiert, um das Optimum zu empfehlen.

Es werden bei der Analyse die folgende Faktoren betrachtet: Anwendbarkeit der Transformation, erwartete Genauigkeit der Vorhersage, Skalierung des Verfahrens, und den Lösungsaufwand – ausgedrückt als die "Zeit bis zum Ergebnis", und auch als Verbrauch von Ressourcen wie CPU und Arbeitsspeicher.

- Eine Evaluierung des Ansatzes im Kontext von zwei realistischen Systemen. Der Ansatz wird mit Fokus auf mehreren Faktoren analysiert, zum Beispiel: Vorhersage der Netzwerkkapazität und des Durchsatzes, Anwendbarkeit, Flexibilität in der Abwägung zwischen Genauigkeit und Mehraufwand. Obwohl der Ansatz sich nicht auf der Vorhersagegenauigkeit fokussiert, demonstriere ich, dass er geringe Vorhersagefehler liefert – bis zu 20% für nicht kalibrierte und bis 10% für kalibrierte DNI Modelle, abhängig von dem Lösungsverfahren.

Zusammenfassend präsentiert diese Dissertation den ersten Ansatz für flexible Performanzanalyse zur Laufzeit von SDN-basierten Netzwerken in Rechenzentren. Er bietet eine flexible Möglichkeit zwischen der Genauigkeit und dem Mehraufwand des Lösungsverfahren abzuwägen. Der Ansatz bietet die folgende Vorteile:

- Es ist möglich die Auswirkung von Änderungen im Rechenzentrum auf Netzwerkperformanz vorherzusagen. Mögliche Änderungen beinhalten hierbei: Änderungen der Netzwerkstopologie, der Hardwarekonfiguration, Last des Netzwerkverkehrs, und der Konfiguration der Softwareanwendungen.
- Das DNI Meta-Modell kann erfolgreich die Performanz von verschiedenen Netzwerkinfrastrukturen abbilden und vorherzusagen, inklusive proaktiven SDN-basierten Szenarien.
- Der Prozess der Vorhersage ist flexibel. Das bedeutet, dass der Ansatz die Balance zwischen Detailgrad von Modellen und Mehraufwand des Lösens anbietet. Hierbei wird eine voraussichtlich verringerte Genauigkeit durch Einsparungen in der Lösungszeit und den verbrauchten Ressourcen belohnt.
- Der Ansatz ermöglicht dem Benutzer verschiedene Lösungsverfahren anzuwenden, ohne ihn zu zwingen, dass er über Expertise und Erfahrung im Anwenden der Lösungsformalismen verfügt.

Die Komponenten des DNI Ansatzes können auch bei solchen Szenarien angewendet werden, die in der Dissertation nicht direkt genannt werden. Der Ansatz bietet ein breites Anwendungsgebiet, zum Beispiel: (a) Es können die Netzwerke außerhalb vom Rechenzentrum analysiert werden, solange der zugrunde liegende Netzwerkverkehr bekannt ist; (b) nichtkalibrierte DNI Modelle können zur Entwurfszeitanalyse der Performanz dienen; (c) die Extraktion- und Komprimierungsmethode von Lastprofilen kann auf andere, nicht netzwerkbezogene Lastprofilen angewendet werden.

Acknowledgements

This work would have been impossible without support of many people. First of all, I would like to thank my advisor Prof. Samuel Kounev for giving me the opportunity to join his group and all his care and mentorship during my work. His patience, guidance, and ideas allowed me to keep my motivation high and reach many goals regarding this thesis and multiple other publications and proposals. I would also like to thank Prof. Ralf Reussner for hosting the Descartes research group and the RELATE project at SDQ.

I thank my colleagues at the University of Würzbrug, Karlsruhe Institute of Technology, and Marie Curie-Skłodowska RELATE Training Network who were always ready to discuss ideas, problems, share thoughts, and provide feedback. In alphabetical order (hoping to not forget anyone): Spiros Alexakis, Rima Al Ali, Ioannis Arampatzis, Andre Bauer, Markus Bauer, Fabian Brosig, Tomáš Bureš, Erik Burger, Axel Busch, Kleopatra Chatziprimou, Rustem Dautov, Zoya Durdik, Dionysios Efstathiou, Simon Eismann, Michael Faber, Ilias Gerostathopoulos, Fotis Gonidis, Inti Gonzalez Herrera Henning Groenda, Johannes Grohmann, Prof. Jürgen Wolff v. Gudenberg, Kahina Hamadache, Lucia Happe, Michael Hauck, Christoph Heger, Robert Heinrich, Jörg Henß, Nikolas Herbst, Heinz Herrmann, Georg Hinkel, Matthias Hirth, Petr Hnětynka, David Hock, Oliver Hummel, Matthias Huber, Nikolaus Huber, Lukas Iffländer, Adrián Juan Verdejo, Isaak Kavasidis, Elena Kienhöfer, Joakim v. Kistowski, Michał Kit, Benjamin Klatt, Fritz Kleemann, Dimitrios Kourtesis, Anne Koziolek, Rouven Krebs, Klaus Krogmann, Max Kramer, Steffen Krämer, Martin Küster, Stanislav Lange, Michael Langhammer, David Mendez-Acuna, Philipp Merkle, Aleksandar Milenkoski, Seyed Vahid Mohammadi, Marco Nehmeier, Qais Noorshams, Fouad ben Nasr Omri, Ivan Dario Paez Anaya, Stelios Pantelopoulos, Michal Papez, Iraklis Paraskakis, Suresh Pillay, Ariana Polyviou, Chris Rathfelder, Andreas Rentschler, Tatiana Rhode, Stamatia Rizou, Vanessa Martin Rodriguez, Jose María Alvarez Rodriguez, Kiana Rostami, Norbert Schmitt, Michael Seufert, Marian Seliuchenko, Viliam Šimko, Simon Spinner, Johannes Stammel, Susanne Stenglin, Christian Stier, Misha Strittmatter, Bholanathsingh Surajbali, Chala Tesgera, Prof. Phuoc Tran-Gia, Mircea Trifu, Robert Vaupel, Jiri Vinarek, Jürgen Walter, Florian Wamser, Alexander Wert, Dennis Westermann, Paraskevi Zerva, Thomas Zinner, Steffen Zschaler, and Saskia Zocher.

Furthermore, I thank students and student assistants for supporting me in various—often very difficult—tasks during my research time: Stefan Herrnleben, Maximilian Kiesner, Frederik König, Ben Morgan, Christoph Müller, Nikolai Reed, Christopher Sendner, Jonathan Stoll.

My passion to research started at the Wrocław University of Science and Technology in the group of Prof. dr hab. inż. Adam Grzech whom I would like to thank

in a special way for giving me the opportunity to start the research career and leading me through my first years at the university as my advisor. His attitude, style of mentorship, and patience shaped my attitude to research and life. I am sad that he has not lived to see me graduate.

Moreover, I would like to thank Dr Paweł Świątek for giving me a chance to work with him and for convincing me to start a PhD. His guidance, feedback, and great support in the first years of my research was irreplaceable. I was very lucky to work in a great atmosphere that would be impossible without him and other colleagues: Krzysztof Brzostowski, Jarosław Drapała, Maciej Drwal, Grzegorz Filcek, Dariusz Gąsior, Adam Gonczarek, Krzysztof Juszczyszyn, Andrzej Kozik, Radosław Rudek, Paweł Stelmach, Prof. Jerzy Świątek, Jakub Tomczak, and Maciej Zięba.

Finally, I would like to thank my parents, grandparents, and my fiancé Agnieszka who provided enormous support and believed in me throughout the years. Without them, this work would not be possible.

Chapter 1

Introduction

1.1 Motivation

Nowadays, data centers are becoming increasingly dynamic due to the common adoption of virtualization technologies. Virtual machines, data and services can be migrated on demand between physical hosts to optimize resource efficiency while enforcing service-level agreements [GGP12, LCE10]. Systems can scale their capacity on demand by growing and shrinking their resources dynamically based on the current load. Moreover, new services (both computing and network) can be composed and deployed on-the-fly leveraging their loosely-coupled nature. These are examples of factors that make modern enterprise systems complex, dynamic, and challenging to manage.

Multiple approaches have been proposed to address the challenges that modern network infrastructures face. The main trend among the various approaches is network virtualization. Modern network infrastructures are increasingly adopting virtualization with the emergence of paradigms such as SDN and NFV. The field of network virtualization is not as mature as server virtualization and there are multiple competing approaches and technologies [BBE+12, CB10]. This increases the sources from where the performance-relevant system complexity originates. The complexity and thus performance of data centers is influenced not only by the software architecture, middleware, and computing resources, but also by network virtualization, network protocols, network services, and configuration.

The complexity of modern enterprise systems running on virtualized computing and network infrastructures makes it challenging to analyze and predict their performance in an accurate and cost efficient manner [MLP+13, WN10a, MST+05]. Nevertheless, modern data center systems are expected to deliver reliable performance, thus performance-aware design, optimal configuration, and resource management policies need to be applied [BAB12, Kan09]. Performance modeling and prediction techniques provide powerful tools to analyze the performance of IT systems and applications running in a modern data center. However, given the wide variety of network virtualization approaches, no common approach exists for modeling and evaluating the performance of virtualized networks.

The performance community has proposed multiple formalisms and models that can be applied to evaluate infrastructures based on different network virtualization technologies, for example: simulation (*OMNeT++* [Var01], NS-3 [RH10]), stochastic

queueing networks, or Markov chains [BGdMT06]. The existing modeling approaches can be divided into two main categories: coarse-grained analytical models (e.g., [JOS+11, ANP+13]) and detailed simulation models (e.g., [RH10, Omn16, DLWJ08]).

Analytical performance models (e.g., [ANP+13, Jar14]) are normally defined at a high level of abstraction (i.e., they are technology independent and can be applied to various network setups). Such coarse-grained performance models abstract many details of the real network without explicitly taking into account aspects such as, for example, the software deployment context, influence of the server virtualization, or the network virtualization technology; therefore, they have limited predictive power. Moreover, analytical performance models are normally expressed as mathematical formulas and they do not explicitly capture the internal system structure. Modeling the internal system structure allows to predict the performance of different system configurations enabling *what-if* analysis.

On the other hand, simulation models usually capture the internal structure of the modeled system explicitly. Such model are normally focused on a selected networking technology and take into account numerous technology-specific parameters, resulting in detailed models that can be used for accurate performance prediction. Unfortunately, the models are usually tightly bound to a given technology, infrastructure setup, or to a given protocol stack. This narrows the applicability of conventional network simulation models to specific cases and thus narrows the scope and feasibility of *what-if* analysis. Moreover, simulation models are often impractical for use at run-time, given that simulations are usually expensive to build and solve [WvLW09, BMB+15].

Existing network performance models do not adequately capture the link of the network infrastructure to the virtualized servers and the applications that generate the traffic in the network. In a virtualized data center, the performance of the virtualized computing infrastructure plays an important role for an end-to-end performance analysis: according to [WN10b], "the fundamental problem is that the simple textbook end-to-end delay model composed of network transmission delay, propagation delay, and router queueing delay is no longer sufficient. Our results show that in the virtualized data center, the delay caused by end host virtualization can be much larger than the other delay factors and cannot be overlooked."

Existing data center network models are usually tightly bound to a given technology, infrastructure setup, or to a given protocol stack, which makes them *inflexible*. A performance model is considered *inflexible* when it provides a single solution method without providing means for the user to influencing the solution accuracy and solution overhead. To allow for *flexibility* in the performance prediction, the user is required to build multiple different performance models obtaining multiple performance predictions. Each performance prediction may then have different focus, different performance metrics, prediction accuracy, and solving time. However, building multiple semantically different performance models requires the network operator (or the modeler) to understand and master multiple different modeling formalisms. Without sufficient expertise and experience in each of the modeling

formalisms, building such models is error prone and may lead to incorrect outcomes, misleading conclusions regarding the performance, and unfeasible resource management decisions.

The goal of this thesis is to develop a modeling approach that bridges the gap between coarse-grained analytical models and detailed simulation models (see Fig. 1.1). The proposed approach is intended to provide a balance between the generic character of coarse-grained analytical models and the more accurate and detailed simulation models. This is achieved by focusing on the major performance influencing factors of virtualized network infrastructures while abstracting the less relevant parameters.

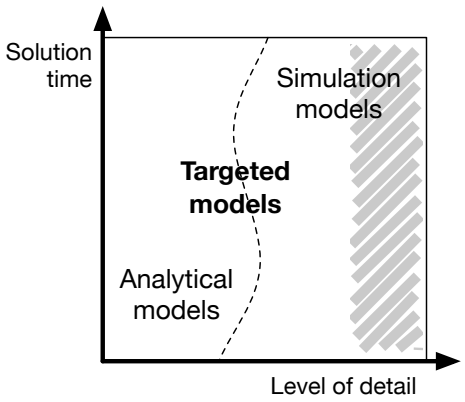


Figure 1.1: Division of the performance model space into analytical and simulation models. Models with high level of detail (gray-striped area) are out of scope.

This thesis bridges the following gaps:

- First, the currently existing performance modeling approaches does not offer *flexibility*. The main missing feature is the ability to select required modeling granularity (coarse-grained analytical models versus highly detailed simulation models) while still keeping its generic nature of the model (i.e., the ability to model different data center networks).
- Second, there is a gap between the performance models of the software architecture, the computing infrastructure, and the network infrastructure. These domains are intertwined and influence each other.
- Third, the existing models lack the generic character and obstruct the usage of the modeling formalism by non-experts. The gap exists between professional performance analysts and the users who are not experts in performance modeling (e.g., network operators).
- Finally, there is a gap between a novice user and a new modeling formalism. The users require guidance during the process of building a model. Preferably,

a model should be automatically extracted and presented to the user for final tuning.

Before I present the problem and the approach, I define the scope and focus of this work. The scope intersects with many technologies and research areas. It can be placed at the intersection of the following domains:

- Software: software engineering, model-driven software development,
- Modeling: domain-specific languages, meta-modeling,
- Performance: prediction using analytical and simulation methods,
- Networking: networking and data center networks, network virtualization, SDN, NFV.

I limit the scope of the modeling approach to network infrastructures managed by a single administrative entity. A good example of such a network is a private cloud data center managed by a single company or organization. The model must contain all data about background traffic in the modeled network, thus I assume that the modeler should have access to all components of the network in order to obtain the required data. A single administrative entity shall ease the access to the extraction of the background traffic profiles as the centralized administration simplifies the privacy-related procedures for capturing traffic traces. Nevertheless, the approach can be theoretically applied to any network infrastructure (even the Internet) yet the performance prediction accuracy will strongly depend on the modeling accuracy of the background traffic profile.

I focus on the network infrastructures of modern virtualized data centers. As a representative network virtualization technology, I select SDN [MAB+08a]. I justify the selection of SDN by its popularity, novelty, and wide adoption. According to a report published by Transparency Market Research [Res13], the global SDN market is expected to reach USD 3.52 billion by 2018, growing at a compound annual growth rate of 61.5% from 2012 to 2018.

I target medium-detailed solvers to demonstrate the *flexibility* of the performance prediction process. The *flexibility* aims at delivering multiple different predictions, each with different accuracy and solving time, so that the optimal solving method can be selected for each run-time situation. I treat equally important accurate, long-running simulations and coarse but quickly solvable performance models.

The medium-detailed modeling granularity is additionally motivated by the *generic* nature of the proposed modeling approach. I aim at supporting any network infrastructure regardless of the protocols, algorithms, technologies, and used hardware. This aim excludes the usage of fine-grained predictive models as they are usually designed to analyze a single defined protocol or device.

Regarding the performance metrics, I focus on the *capacity of the network*. I analyze the throughput of network devices to conclude how much network capacity is used at a given moment. The approach, however, is not limited to capacity-related metrics and supports any performance metric that is offered by the predictive model solvers delivered in this work.

1.2 Problem Statement

Managing system resources to ensure optimal capacity and performance without violating service level agreements (SLAs) is a challenging problem. It requires the ability to predict the behavior of the system in the face of changes that may originate from the dynamic workload, virtualization, or online reconfiguration of the system, so that resource allocations can be adapted *before* such changes occur.

Methods that predict the performance of a system can be characterized in terms of multiple different features. For example, the methods may differ in their prediction accuracy and the time it takes to provide the performance prediction. Additionally, many other features may characterize a performance prediction approach, for example: level of detail of the returned performance analysis (the returned metrics, averages, probability distributions, etc.), scalability of the solution (size of the network/number of flows that can be modeled), consumption of resources (CPU, memory), repeatability of the analysis, reliability of the solving (i.e., a guarantee that a method will finish and return results).

Providing various approaches to solve a single performance prediction problem allows to see the system from different point of view, each of them disclosing different possible positive and negative aspects. Finding a set of heterogeneous performance predictors that differ in their characteristics but can be used interchangeably or simultaneously to solve a model is called *flexible performance prediction*. In the following, I describe the challenges of *flexible performance prediction* and provide a formal view on the problem formulation.

1.2.1 Flexible Performance Prediction

Existing approaches to performance prediction usually focus on optimizing a single prediction criterion (e.g., maximizing prediction accuracy or minimizing solving time) and provide limited the balance between multiple criteria. Due to that, an analysis of possible trade-offs between the parameters that characterize a performance prediction method is missing. An approach to flexible performance prediction is expected to provide different performance analysis methods that analyze the same system but represent it differently, for example: abstracting selected parts of the system, representing it using different formalisms, solving using different solving methods.

Considering the focus of this thesis, I present the flexible performance prediction problem in the area of data center networks and in the system run-time phase (in contrast to the system design-time, when the system has not been built yet). An approach to *run-time flexible data center network performance prediction* has to fulfill several requirements:

- The performance prediction approach should express the prediction results using at least one standard metric, such as: device/port capacity, network interface throughput, end-to-end transmission delay, ratio of dropped traffic for an interface. The metrics can be expressed as averages or distributions, so that various statistics can be calculated for more detailed analysis.

- The prediction mechanism should support conducting an analysis of an impact of *a change* in the system on the system performance. The analysis should be based on data acquired from the running system. The data should be extracted from the system and fed into a model (preferably automatically), so that various changes can be introduced to the model without affecting the operation of the system. To possible system changes, I account, for example: switch reconfiguration, topology change, migration of an application service, change in the workload profile or intensity.
- The model of a running system should be represented using terms that are familiar to an average user, for example, a network operator or a data center operator. The users should be able to tune the extracted model according to their needs.
- To call the performance prediction flexible, the user should be offered multiple solving methods for a single input model. The solving methods should differ in at least one characteristic (e.g., prediction accuracy, solving time, solver resource consumption).
- The performance prediction method should abstract parts of the system that have low influence on the performance. The criterion for including or abstracting a given part of the system in the model should be configurable. This enables generating models at different abstraction levels and thus supporting different performance analyses.
- The extremes of the finest and the coarsest modeling granularity should be defined for the modeling formalism. The former defines the upper bound in term of data required by the model, whereas the latter defines the minimum set of data that needs to be provided.
- The modeling approach should be accessible from a programming language and should offer a form of an application programming interface (API), so that automation and integration with other tools is possible.

An ideal approach that meets the requirements of *run-time flexible data center network performance prediction* faces many challenges. In the following, I present the decomposition of the problem statement and provide most relevant research questions and challenges for each part.

Design of the descriptive modeling language. The most challenging design decision is to find a *balance between the level of detail and granularity* of the modeling language (i.e., its expressiveness), on the one hand, and its generality and wide applicability, on the other hand. The less technical details are required and the more generic an universal the used modeling elements are, the more network infrastructures can be represented using the language. On the other hand, with increasing generality, specific network technologies need to be represented in

a more abstract manner limiting the types of existing predictive models (e.g., protocol, or domain-specific) that can be automatically generated.

The SDN-based networking introduces several new challenges and problems. As stated in [KPK15], the SDN paradigm introduces strong correlations and mutual performance influences between the SDN controllers, the SDN-applications, the SDN switches (data planes), and the performance of the underlying network. These performance influencing factors are not present in classical (non-SDN) networks.

Another issue is the naming of the model entities such that experts from different technology domains can quickly understand their intention. This is important to reduce the learning curve for practitioners using the language. The scope of this thesis spans two expertise areas: data center networking and software applications deployed in virtualized servers, thus the descriptive models can be built by experts from both domains and the terms used in the model must be understandable for both.

The challenges can be summarized with the following research questions: (1) At what level of abstraction should the network be modeled to support different trade-offs between model accuracy and analysis overhead? (2) Which network entities should be modeled directly (e.g., "a protocol" versus Transmission Control Protocol (TCP)) and which entities should be abstracted (e.g., a server versus a node)? (3) How should SDN-based networks be modeled distinguishing between physical and logical implementation of the network control plane? (4) How to represent the correlation between highly-specialized software SDN-applications and the overall network performance?

Automatic model-to-model transformations. Descriptive models organize information about a domain. The proposed modeling language have descriptive nature and do not provide any means of performance prediction. To leverage the descriptive domain information in the performance prediction process, a model needs to be transformed into a predictive model using a model transformation. Kleppe et al. [KWB03] defines a model transformation as follows. "A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

I identify at least three challenges concerning the automatic transformations between models. The aim of a model transformation is to transform a model specified in one language into a respective model in another language, possibly without any human involvement. Full automation of this process requires to solve many challenges assuming no external sources of additional information.

The first challenge concerns the level of detail that should be abstracted from the source modeling language in a transformation. More detailed descriptive models allow to abstract more parts of the source model and thus increase the flexibility of the approach. The more detailed the language is, the more technology-specific

and less generic it becomes. Moreover, larger models increase the time of the transformation, as more objects need to be transformed. On the other hand, more detail enable transformations to produce more accurate predictive models.

The second challenge is the parameterization of the source modeling language. Descriptive models may define some of their parameters as optional. The user building a model is required to provide only the mandatory data. Proper optionality settings of the language elements may be crucial for selected transformations to deliver usable predictive models and feasible predictions.

The third challenge concerns the support in selecting feasible model transformations for a given scenario. Assuming that multiple model transformations can be applied, the approach should filter-out the incompatible transformations and suggest which transformations are better suited for the specific scenario.

Based on the stated challenges, I identify the following research questions: (1) How to design a transformation if the target model requires more information than currently available in the modeling language? (2) Which data should be abstracted if the target model is less detailed than the input model? (3) How to optimize the performance of the transformation? (4) Which are the most important factors that influence the prediction accuracy and solving time? (5) How can an optimal set of transformations be recommended considering the user-provided criteria regarding the prediction? How fine-granular should the recommendations be?

Support for semi-automatic model extraction. Other challenges concern the support for extracting the system models based on monitoring data from a running system. Experienced users may value the wide variety of modeling possibilities and model the network manually from a scratch, but providing an initial preconfigured model saves time and requires less involvement from the user. However, automatically extracted data may not always represent the actual system in a reliable manner. For example, tools that analyze the topology of a network may only see the logical topology based on routing information, whereas the actual physical topology may differ. Additionally, some of the extracted information related to hardware devices may differ from the official technical specification and thus be uncertain.

Traffic profile extraction faces similar challenges. The realistic network traffic is usually too difficult to model manually assuming the existence of complex traffic patterns, high traffic volumes, and the required measurement time resolution expressed usually in microseconds. Fortunately, traffic models can be extracted from captured traffic traces. The traffic profiles are represented as time-series that approximate the captured traces representing only model-relevant data without any payloads. It may be desirable to store the traffic models in a compressed form to avoid increasing the solving times due to the model size. An optimization method may be developed to compact the traffic models and ensure balance between the model size and the level of detail.

I formulate the following research questions regarding the support for model extraction: (1) Which information can be extracted automatically out of a running

data center? (2) How to design an extraction process that supports heterogeneous hardware? (3) How reliable are the extracted data? (4) Which mechanisms (e.g., security policies) may prohibit access to the respective parts of the modeled system and thus affect the completeness of extracted information? (5) How long should the traffic be observed to obtain a reliable traffic model? (6) At which level should the traffic data be abstracted to find balance between size of the model and accuracy of the representation?

Compatibility and integration with other domain modeling languages. Modeling the data center network infrastructure is insufficient to provide a complete view over the data center as a whole. Computing resources, software architecture, and storage equipment can also limit the performance of the system in specific scenarios. Thus, solving network-focused or software-focused performance models separately provides only a partial coverage of the modeled system. Such focused performance models may represent properly a system under certain circumstances, that is, if the modeled fragment of the data center is a bottleneck (e.g., network bottlenecks cause delays in distributed business applications). In general, however, domain-specific models should offer an interface allowing to integrate the model with other models, so that a holistic representation of the data center can be build and used to evaluate the end-to-end application performance.

However, integrating different modeling languages poses several challenges that need to be addressed. The main concerns can be summarized with the following research questions: (1) How to design a minimal weaving model to integrate two or more models representing different aspects of the system? (2) How to ensure that the integrated models can be separated again if needed? (3) How do design an approach to the solving of the integrated models?

Building a testbed for validating the approach (technical). Building a realistic data center testbed for validation of the proposed approach is an important technical challenge. I target virtualized network infrastructures in this thesis and thus require a reference network infrastructure that represents one of the up-to-date data center network virtualization technologies. This requires acquiring modern network hardware, building a representative topology, configuring the hardware, and running realistic workloads to generate representative network traffic. The technical challenges can be summarized with the following questions: (1) How to build a minimal representative data center assuming budget limitations? (2) How to select a representative, future-proof network virtualization technology? (3) Shall the devices be homogeneous or heterogeneous? (4) What hardware and in what amounts should be obtained? (5) Which network topology is scalable enough to treat it as representative? (6) Which applications/benchmarks shall be used for generating realistic data center network workloads?

1.2.2 Formal View of the Problem

In this section, I provide a formal specification of the problem of flexible performance prediction. It is challenging to define the tackled problem as an instance of a known optimization or decision problem given that it is unclear how the objective function should be defined. Thus, instead of proposing an optimization problem with a formally defined objective function, I define the problem as a constraint satisfaction problem.

The formal notations presented in this section allow to understand the problem from the formal perspective. However, further in the thesis, I refer to the problem using the natural language.

Formalization

The notation used in this section is summarized in Tables 1.1 and 1.2.

Table 1.1: Denotation of the real and modeled entities used in the formal problem description.

Real entity	Description	Model	Description
Assume all happens in a time span $t_0 \rightarrow t_1$ starting in moment t_0 and ending in t_1.			
net_i	i-th network	$\bar{\Phi}_{i,k}$	k-th descriptive model of i-th network
net_i	i-th network	$\Phi_{i,j}$	j-th predictive model of i-th network
$\forall_k \bar{\Phi}_{i,k}$ describes net_i and allows to produce multiple different $\Phi_{i,j}$.			
$\forall_j \Phi_{i,j}$ represents the performance of net_i.			
$f_n(net_i)$	Measured value of n-th performance metric of network net_i	$g_{s,n}(\Phi)$	Value of n-th performance metric obtained from model Φ using s-th solver g.
$\forall_s \forall_i g_{s,n}(\Phi_{i,j})$ approximates $f_n(net_i)$.			

Table 1.2: Further notation used in the formal problem description.

Symbol	Description
$\mathcal{I} = \{net_1, \ldots, net_i, \ldots, net_I\}$	Set of networks
$\bar{\Phi}_{i,k}$	k-th descriptive model representing i-th network
$\mathcal{J}_i = \{\Phi_{i,1}, \ldots, \Phi_{i,j}, \ldots, \Phi_{i,J}\}$	Set of J predictive models representing i-th network
$\mathcal{N} = \{P_1, \ldots, P_n, \ldots, P_N\}$	Set of performance metrics
$\mathcal{S} = \{g_1, \ldots, g_s, \ldots, g_S\}$	Set of network model solvers
$\mathcal{R} = \{r_1, \ldots, r_l, \ldots, r_L\}$	Set of model transformations

Assume there exists i-th real network $net_{i,t_0 \rightarrow t_1}$ observed in a time span starting in moment t_0 and ending in t_1. The i-th network can be represented by a k-th descriptive model $\bar{\Phi}_{i,k}$ and j-th predictive model $\Phi_{i,j}$ (regardless of its form; time span indexes are omitted for brevity). The performance of a network can be represented with N performance metrics, so that the value of the n-th metric $f_n(net_i)$ can be measured or estimated empirically. For a network model $\Phi_{i,j}$, the

performance can be predicted using an s-th solver $g_{s,n}(\Phi_{i,j})$, which solves the model and returns the performance metric values. I assume that only predictive models can be solved directly. The descriptive model $\bar{\Phi}_{i,k}$ cannot be solved directly and must be transformed into a predictive model first.

The relative performance prediction error with respect to the n-th performance metric is defined as follows:

$$\varepsilon_{s,n} = err_n(net_i, f_n, \Phi_{i,j}, g_{s,n}) = \left| \frac{f_n(net_i) - g_{s,n}(\Phi_{i,j})}{f_n(net_i)} \right|. \tag{1.1}$$

For an ideal model solver g^* and ideal network model $\Phi_{i,*}$, the prediction accuracy is maximal and the error $\varepsilon_{*,n} = 0$ for each n in this case.

The solvers are characterized by solving time defined by function $h(g_s, \Phi_{i,j})$ that returns the solving time of model $\Phi_{i,j}$ with the s-th solver. The maximal duration of the model solving may be constrained by the user, so I assume that

$$\bigvee_{s=1}^{S} 0 < h(g_s, \Phi_{i,j}) \leq h^{max}, \tag{1.2}$$

whereas the prediction accuracy error for solver s may be constrained with respect to the performance metrics by defining ε_n^{max}

$$\bigvee_{n=1}^{N} 0 < \varepsilon_{s,n} < \varepsilon_n^{max}. \tag{1.3}$$

In general, different network models and different solvers offer different solving time and prediction accuracy.

$$\begin{gathered} \text{If} \quad s' \neq s'' \quad \text{and} \quad j' \neq j'' \quad \text{then:} \\ h(g_{s'}, \Phi_{i,j}) \neq h(g_{s''}, \Phi_{i,j}), \\ h(g_s, \Phi_{i,j'}) \neq h(g_s, \Phi_{i,j''}), \\ err_n(net_i, f_n, \Phi_{i,j}, g_{s',n}) \neq err_n(net_i, f_n, \Phi_{i,j}, g_{s'',n}), \\ err_n(net_i, f_n, \Phi_{i,j'}, g_{s,n}) \neq err_n(net_i, f_n, \Phi_{i,j''}, g_{s,n}). \end{gathered} \tag{1.4}$$

The problem is formulated as follows. Find a set of solvers $\mathcal{S}^* \subseteq \mathcal{S}$ and network models $\mathcal{J}_i^* \subseteq \mathcal{J}_i$, such that the constraints given by Equations 1.2 and 1.3 are satisfied, that is: the solving time is lower than the user-given maximum h^{max} (Eq. 1.2) and the prediction error is lower than the user-given maximum ε_n^{max} (Eq. 1.3). Assuming that the solvers offer different trade-offs between accuracy and cost (Eq. 1.4), the problem is a constraint satisfaction problem given with the following constraints:

$$\begin{gathered} \bigvee_{s \in \mathcal{S}^*} : h(g_s, \Phi_{i,j}) \leq h^{max}, \\ \bigvee_{j \in \mathcal{J}_i^*} \bigvee_{s \in \mathcal{S}^*} \bigvee_{n=1}^{N} : err_n(net_i, f_n, \Phi_{i,j}, g_{s,n}) < \varepsilon_n^{max}. \end{gathered} \tag{1.5}$$

The Approach in the Formal Context

Before presenting the approach of this thesis in detail (Section 1.3), I place its elements (the descriptive model, transformations, and solvers) in the context of the above problem formalization. This helps to better under understand of both—the formalization and the approach.

In this thesis, the models $\Phi_{i,j}$ are generated using model transformations (defined as functions $r_l \colon \bar{\Phi} \mapsto \Phi$) from a k-th descriptive model $\bar{\Phi}_{i,k}$ representing the i-th network. The set of K descriptive models representing the i-th network and L model transformations allow to generate up to J predictive network models $\Phi_{i,j} = r_l(\bar{\Phi}_{i,k})$ (assuming that $0 < l < L$ and $0 < k < K$, then $J \leq K \cdot L$). In practice, not all J models can be solved as not all model transformations support producing a valid network model for each scenario.

Based on the introduced notation and the problem statement, the main contributions of this thesis are:

1. the modeling language that allows building multiple (assume K) descriptive models $\bar{\Phi}_{i,k}$ that represent the i-th network,
2. a set of model transformations $\mathcal{R} = \{r_l \colon l = 1, \ldots, L\}$ that convert the descriptive model $\bar{\Phi}_{i,k}$ into up to J different predictive models $\Phi_{i,j}$.

1.3 Approach and Contributions

As a main contribution of this thesis, I propose a generic approach to modeling and analyzing the performance of virtualized data center networks in a flexible way. The approach proposed in this thesis is named **DNI**. DNI is both an approach to performance prediction as well as a modeling language on which the approach is based. The main part of the approach is the DNI meta-model—a novel descriptive modeling language providing modeling abstractions that can be used to describe the performance-influencing aspects of both virtualized and non-virtualized data center network infrastructures. DNI is a sister language to the Descartes Modeling Language (DML) [KHBZ16]. DML targets software architecture and server virtualization, whereas DNI focuses on network infrastructures in a data center.

The approach includes six automatic model-to-model transformations that transform a descriptive DNI model instance into solvable predictive performance models. The obtained predictive models can be solved by up to ten solvers—each of them employing a different abstract representation of the modeled network. Furthermore, techniques are provided to facilitate the extraction of the DNI models from monitoring data collected from the system under study, as well as to support the user in selecting appropriate model solvers for the specific performance prediction scenario.

The components of the approach are illustrated in Figure 1.2. The goal is to analyze the performance of a data center network, used to connect a computing infrastructure hosting a set of applications. An engineer builds a model of the

data center network either manually or with the help of model extraction tools. Next, the network representation is stored in a DNI model, that is, an instance of the DNI meta-model. Once the DNI model is checked for validity, the execution of all further steps is automated. Multiple model-to-model transformations read the data stored in the model and automatically generate predictive performance models. Finally, the predictive models are solved using the available model solvers and the prediction results are returned to the user.

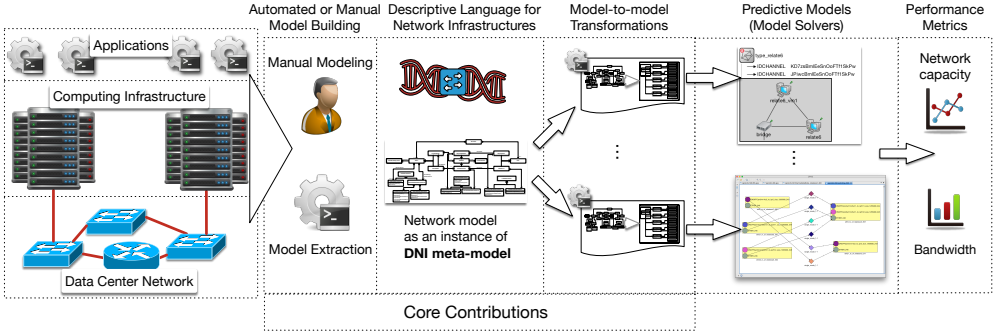


Figure 1.2: DNI Approach: areas, contributions, and focus.

The main benefit of the proposed approach—the *flexibility* of the performance analysis—is achieved through using multiple different transformations of the DNI model that captures the performance-relevant network aspects. The DNI model is automatically processed to obtain different predictive models that represent various modeling granularities and can thus be used to provide multiple performance predictions with various characteristics. By integrating multiple different predictive models and solution techniques, the approach allows the user to flexibly trade-off between prediction accuracy and overhead according to the user requirements and constraints. In contrast to this, a traditional approach to model a system using multiple different predictive models would require significant manual effort to build each predictive model separately. It would also require expertise in the respective modeling formalisms and thus would limit the applicability of the approach to data center operators who are performance analysis experts at the same time. This is illustrated in Figure 1.3. As discussed in Chapter 3, currently no such flexible approaches exist in the domain of virtualized data center networks.

The approach is characterized by the following novel aspects:

(1) The generic character of the modeling language makes it technology-independent allowing to model communication networks of different types.

(2) Once a network is modeled in DNI, multiple predictive models are generated automatically. This allows to conduct multiple performance analyses using different prediction tools without requiring knowledge and experience in performance prediction techniques.

(3) The automatically generated predictive models vary in their prediction accuracy and solution time. This allows to obtain less accurate prediction results

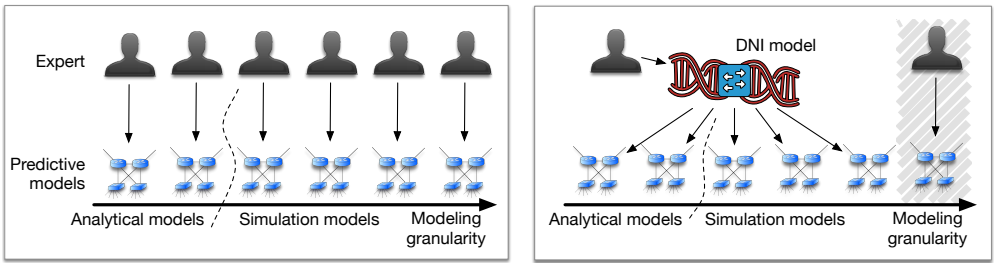


Figure 1.3: Core benefit of the DNI approach: DNI supports a range of models with different modeling granularities without requiring expertise in each of them.

in a shorter time or more accurate results at higher solution costs depending on the user preferences.

(4) The approach supports modeling virtualized networks based on SDN as a selected representative network virtualization technology. The generic nature of a node in the network allows to use the proposed modeling formalism for NFV-based networks as well.

In the following, I present in more detail the building blocks of the approach proposed in this thesis, grouping the thesis contributions into research (primary and secondary) and technical. The secondary research contributions are closely related to the approach, however they are validated using different case studies than the primary research contributions.

1.3.1 Primary Research Contributions

Novel Modeling Abstractions for Virtualized Network Infrastructures

The main contribution of this thesis is the DNI meta-model, which defines a modeling language designed to model virtualized data center network infrastructures for capacity management purposes. This involves the following main parts:

(a) the DNI meta-model—a new medium-detailed descriptive modeling language for data center networks,

(b) miniDNI meta-model—a minimal version of the modeling language for the coarsest modeling granularity,

(c) modeling elements for describing SDN-based networks and their performance-relevant aspects,

(d) a meta-model integration interface for integration between DNI and DML as a representative descriptive model for other data center domains.

DNI is designed to offer a flexible medium-detailed modeling granularity abstracting low level details of the network and focusing on the most important performance-influencing factors. The meta-model offers flexibility by allowing the user to specify parts of the system in different ways, as discussed in detail in Section 4.3.1.

The miniDNI meta-model is a minimal version of the DNI modeling language. miniDNI is useful in cases where a minimum amount of information about the system under study is available. miniDNI defines a lower bound on the modeling granularity for which feasible performance analysis is supported.

Regardless of which variant of the DNI meta-model is used, the modeling language provides generic modeling elements allowing to describe the majority of existing and future network technologies. In this thesis, I focus on SDN and NFV as examples of modern virtualization technologies. DNI provides support for SDN supports novel prediction scenarios (e.g., software SDN forwarding table), while at the same time abstracting factors that have low influence on the overall performance.

Another important part of the DNI approach is the network deployment meta-model—an interface between DNI and other meta-models that allows to define mapping between DNI and other descriptive models. The integration with other domain-specific models allows capturing behaviors that are not reflected in the DNI model, for example, software bottlenecks, server virtualization, and middleware overheads.

The contributions presented above were published in [RSK16, RKTG15, RK14b, RKZ13, RZK13]. The DNI modeling language was initially designed in 2012 [RZK13] and gradually extended until its final version in 2016 [RSK16].

Flexible Model Solving with Model Transformations

DNI models have a descriptive nature, that is, they store information about the network infrastructure, however, without providing any means to predict the network performance under different conditions. To enable performance prediction, a DNI model is transformed into predictive models using model-to-model transformations. Each model transformation (or a chain of multiple transformations) contributed in this thesis enables solving a DNI model by transforming it into a predictive model. The model transformations vary in size and complexity depending on the amount of data abstracted in the transformation process and provided to the solver.

In this thesis, I contribute *six* transformations that transform DNI models into various predictive models based on the following modeling formalisms: (a) *OMNeT++* simulation, (b) Queueing Petri Nets (QPNs), (c) LQNs. For each of there formalisms, multiple predictive models are generated (e.g., models with different level of detail): (a) two for *OMNeT++*, (b) two for QPNs, (c) two for LQNs. Moreover, some predictive models can be solved using multiple alternative solvers resulting in up to *ten* different automated solving methods for a single DNI model. As described in Section 1.2.1, the main incentive for supporting various modeling formalisms is the difference in their characteristics, which is a prerequisite for the flexible performance prediction approach. In this thesis, I focus mainly on the *OMNeT++* and QPN models and solvers as the applicability of the LQN formalism for data center networks is limited. For this reason, the LQN model transformation (described in detail in Section 5.5), it is considered as a secondary contribution.

The proposed list of performance modeling formalisms is by no means exhaustive. In this thesis, I contribute, selected model-to-model transformations, however, the general approach does not limit the amount and type of transformations that may be applied to a DNI model, so the set of supported transformations can be further extended.

The DNI solvers contributed as part of this thesis are evaluated in terms of their execution times and prediction accuracy for the metric network capacity. The network capacity is expressed as the maximum possible network traffic that can be sustained, that is, the amount of consumed network bandwidth. However, the solvers deliver other performance metrics as well, for example, end-to-end network transmission delays and packet losses. Nevertheless, given the medium-detailed nature of the DNI modeling approach, it is impractical to conduct model calibration for such performance metrics due to the inherent interferences between the network and the computing infrastructure. Software applications are typically sources of much higher delays than the network, so the analysis with the DNI approach is unfeasible unless the solvers for the integrated DNI and DML are used. On the other hand, pure point-to-point delays (e.g., port-to-port) are of limited practical value in the context of a complex data center, where usually the end-to-end processing time of a service is more interesting than a low-level switch-to-switch latency.

The model transformations contributed in this thesis are presented in Chapter 5 in an illustrative, graphical form. The graphical examples-based presentation of the transformations eases their understanding. Formally, all transformations are specified with code included as a part of this thesis. The contributions regarding model transformations were published in [RSK16, RKTG15, RK14a, RK14b, RKZ13].

1.3.2 Secondary Research Contributions

Traffic Model Extraction

As a secondary contribution of this thesis, I provide methods and tooling that support the modeler in the modeling process by automatically extracting and prefilling the DNI model with the network traffic data. Traffic profiles are difficult to extract manually due to the high amount of data transmitted over the network in a short period of time. Fortunately, traffic models can be extracted automatically from traffic traces captured on the network ports of the devices. The traffic profiles I capture are represented as time-series of the data volume for an interface. They contain only model-relevant data (simplified time series) without any payloads. Moreover, I provide a new traffic profile abstraction and optimization method in order to minimize the size of the traffic model while assuring the compactness and representativeness of the extracted traffic profiles. The method ensures balance between the size and the level of detail of the extracted profiles.

I consider the extraction process as semi-automatic because the contributed method expects captured traffic traces as an input and produces a partial DNI model. The approach was published in [RSS+16].

A Method for Selecting Feasible Solvers for DNI Model

A DNI model can be automatically transformed into multiple predictive models. As a secondary contribution, I conduct a trade-off analysis characterizing each transformation with respect to various parameters such as its specific limitations, expected prediction accuracy, expected run-time, required resources in terms of CPU and memory consumption, and scalability (based on the modeled traffic volume and network size). The analysis should enable the declarative and tailored performance prediction—as presented in [WvHK+16, GBK14]—based on the constraints and requirements specified by the user, for example, in the following way: *network size ≈100 servers; traffic volume: low; expected prediction accuracy: any; expected solving time: <10 minutes;*. The proposed method is presented in Section 5.6 (selection of solver based on the modeled features) and 7.4.3 (evaluation of the solvers based on the solving time and solver resource consumption).

Transformation to Layered Queueing Networks

I provide a model transformation to a coarse-grained predictive model that offers support for analytical solver to speed-up the solving process and propose the most coarse-grained representation of a network. In Section 5.5, I propose a transformation that transforms QPN models into LQNs. As QPN models can be automatically obtained based on two types of DNI models (DNI and miniDNI), chaining the transformations DNI-to-QPN and QPN-to-LQN may provide two LQN models representing the network with different granularity. The obtained LQN models can benefit from three existing solvers (two of which are analytical): LQNS [FMW+09], LQSIM [FMW+09], LINE [PC13]. The analytical solvers are expected to speed up the solving process for large models.

I characterized the QPN and LQN formalisms by comparing the differences. I characterized their incompatibilities and highlighted model fragments where the information could be lost due to the different nature of the formalism (e.g., loops or fork-join patterns).

The QPN-to-LQN model transformation was published in [MRSK16]. The evaluation of the transformation published in [MRSK16] was further extended by Müller in his master thesis [Mü16].

1.3.3 Technical Contributions

Additionally to the presented scientific and conceptual contributions, I provide tools, editors, and evaluation procedures that implement the approach. The technical contributions allow to use the conceptual contributions in practice without in-depth understanding of the internal specifics of the approach.

Implementation of the DNI Modeling Language Including an Editor

The DNI meta-model is the basis of the approach proposed in this thesis. The meta-model is the implementation of the modeling language. The DNI meta-model

is implemented using *Ecore*, which is "the defacto reference implementation of Essential Meta-Object Facility (EMOF)" [Eco10, MOF14]. The DNI meta-model is implemented using a derivative of Ecore called Xcore [Xco16], which allows to specify Ecore meta-models in textual form and adds programmability allowing automatic derivation of property values while maintaining full compatibility with Ecore. The implementation of the DNI meta-model includes an automatically generated and fine tuned editor, which supports the modeler in the process of manual DNI model building by suggesting feasible parameter values, applying defaults where applicable, and issuing warnings where relevant. The implementation of the DNI meta-model is publicly available in the DNI Git repository that is available under: `http://descartes.tools/dni`.

Implementation of the DNI Tool Chain

Once a DNI model is built, it can be processed in numerous ways. The model can be verified for correctness, duplicated and prepared for batch analysis, optimized for compactness, and transformed into predictive models, which can be solved providing performance predictions. Each of these steps requires a transformation, a script, or an application to be run in a specified order. The DNI tool chain is a wrapper implementation that encapsulates the transformations, model verification scripts, model manipulation scripts, adapters to run the solvers, and scripts to process the prediction results and display them in a desired form. The DNI tool chain allows, for example, conducting automated batch performance analysis, in which each copy of a DNI model represents a change in the original model, for example: analyze network capacity for a workload intensity originating from 1, 2, 3, ..., 10 deployed applications. The implementation of the DNI tool chain is publicly available in the DNI Git repository that is available under: `http://descartes.tools/dni`.

Evaluation of the Approach in the Context of two Realistic Systems

The validation of the DNI approach addresses several goals: applicability of the approach, multiple performance predictions with different accuracies and solving times, good prediction accuracy of network capacity, good usability of automatically extracted traffic models. I validate the approach using two realistic systems: *SBUS-PIRATES* message bus for traffic monitoring systems, and *L7sdntest* software that generates load by mimicking a cloud storage, for example, Dropbox. As part of the evaluation, I conduct a range of experiments to demonstrate and validate the approach. Based on the obtained results, I evaluate the approach with focus on such factors like: prediction of network capacity and interface throughput, applicability, flexibility in trading-off between prediction accuracy and solving time.

The validation results show that the proposed approach can provide a variety of performance predictions that can be applied in practice based on the required accuracy and solving time constraints. Despite not focusing on the maximization of the prediction accuracy, I demonstrate that in the majority of cases, the prediction

error is low—up to 20% for uncalibrated models and up to 10% for calibrated models depending on the solving technique. Moreover, the modeling abstractions included in DNI allow to properly represent the behavior of SDN-based networks with heterogeneous network devices enabling performance analysis of switches with different implementations of flow tables.

1.4 Application Areas

The DNI approach was designed to enable conducting performance analysis in various scenarios. In this section, I present several such scenarios and application areas for DNI.

Run-time Capacity Management

Data center networks are no longer static nowadays. To handle varying dynamic workloads, networks not only adapt in terms of their virtual logical structure but also the physical topology can be expanded by adding or replacing network devices and servers. Each change in the data center may have an impact on the overall system performance and thus various resource allocations may need to be applied to handle the varying workloads without exceeding the capacity of the network and violating the service agreements.

With DNI, data center operators can investigate the impact of changes in the allocated resources before such changes happen. They may also analyze the current state of the system and identify potential bottlenecks as candidates for reconfiguration or upgrade. Assuming that the model is calibrated with data collected at run-time, the prediction errors are usually lower than in design-time performance analysis. The DNI approach helps to answer the following questions that may come up during system operation:

- What is the average utilization of a given device/network interface for the current load conditions and network configuration?
- How many application services can be deployed in the data center without exceeding the available network capacity?
- Which maximal ratio of over-provisioning can be applied to handle the maximum number of customers?
- Which level of bandwidth over-provisioning is required to handle workload spikes without exceeding the maximal capacity of the network?
- Which hardware should be upgraded (which device encounters the most serious bottlenecks)?

The run-time capacity management consist of analyses that concern different parts of the data center network. In the following, I present three areas, for which the impact analysis of changes is supported with DNI. They include: workload profile and intensity, structure, and configuration of the network.

Impact Analysis of Changes in the Workload Profile and Intensity

Network workloads are usually highly dynamic, so the data center infrastructure must be able to handle them to satisfy the customer needs. With the DNI approach, a system operator may analyze the possible scenarios of workload profile and intensity in order to prepare the system for a load spike. DNI may support the operator in predicting the impacts of possible workload fluctuations on the system's capacity. DNI provides support in answering the following questions that may arise during workload analysis:

- How much bandwidth is needed between devices to handle the expected traffic workload?
- How many service requests must be refused to keep a safe buffer of free network capacity?
- How to load-balance the network traffic to not exceed the capacity of the network?

Impact Analysis of Network Structure Changes

Thanks to modern virtualization technology, it is much easier nowadays to replace data center equipment during run-time. Virtualized servers may be migrated to a different node while the hardware gets upgraded. Unfortunately, the virtualization of networks is more complex and less mature when compared against the virtualization of computing resources. Network hardware upgrades may lead to serious consequences for the system if there is no free network capacity on other devices. Similarly, scaling the network infrastructure of the data center (e.g., to support more servers) may require updating the topology. DNI supports network operators by allowing them to analyze the network with various devices and network topologies to select the best upgrade/maintenance plan. DNI provides support in answering the following questions that may arise by considering an upgrade in the data center network structure:

- Will the new hardware handle the usual traffic better than the old one?
- How robust is the network topology against link failures? Can it handle the traffic after a failure?
- Which topology is optimal for the expected future scaling of the offered services?

Impact Analysis of Network Configuration Changes

Proper network configuration plays an important role for sustainable performance of the data center. Especially in SDN-based networks, a misconfiguration is easy to overlook (e.g., a flow rule is installed into improper flow table) and the consequences may be severe for the performance of the entire network (as presented in, e.g., [KPK15]). Similarly the configuration of routing may overload selected links, so that redundant paths must be enabled to increase capacity (e.g., using Equal-Cost Multi-Path Routing (ECMP)). In the same way, improper configuration

of load balancing (e.g., implemented using SDN, as shown in [Sto16]) may impact selected services while their duplicates may run underutilized.

The DNI approach supports modeling of the most performance-influencing configuration options. This allows the network operators to analyze the impacts of a potential reconfiguration the system performance. The contributed approach aids the network operator in answering the following questions:

- How to load-balance the network traffic to not exceed the capacity of the network or overload the application servers?
- How to configure an optimal routing?
- Is the SDN flow table big enough for current and future SDN scenarios?
- What happens if an SDN device starts using the slow software flow table?
- What happens when the SDN flow table capacity is exceeded?

1.5 Thesis Organization

In Chapter 1, I introduced the reader to the topic and gave a compact overview of the most important aspects of this thesis. The rest is organized as follows.

In Chapter 2, I present the foundations. The chapter describes the basics of the research areas that this thesis spans including: (a) network and its virtualization including SDN and NFV in the context of data centers; (b) approaches to modeling of network traffic and performance; (c) run-time and design-time aspects of the performance prediction; (d) descriptive modeling using meta-models as domain-specific modeling languages.

Chapter 3 summarizes related work on performance analysis of SDN-based—or other virtualized—data center networks. I focus on approaches based on a gray-box modeling or using descriptive models.

Chapter 4 describes the main primary research contribution of this thesis: the DNI meta-model and its minimal version miniDNI. The DNI meta-model is presented in two parts: first, how it supports modeling of classical networks (Section 4.1) and second, how the language is extended to offer the support for SDN-based networks (Section 4.2). The description is enriched with examples that demonstrate various applications of the modeling language (including an NFV scenario). Next, in Section 4.3, I present the flexibility offered by the DNI and miniDNI meta-models. Section 4.4 describes the integration of DNI with DML. The integration is presented using examples and a compact deployment meta-model defining the mapping between the meta-models is proposed.

Chapter 5 describes how a DNI model is processed and solved to provide performance predictions. First, I describe the procedures used for model validity checking, the ways to parametrize transformations, and so-called in-place DNI transformations that transform the user-friendly DNI version into a transformation-friendly format. Next, in Sections 5.2–5.5, I describe six model-to-model transformations that automatically process descriptive models (DNI, miniDNI) and produce predictive models that can be solved with various solvers. Finally, in Section 5.6, I characterize the differences and semantic gaps between the transformations and

the solvers. I present a method that recommends a set of feasible solvers by evaluating the feasibility of transformations and solvers based on the network features included in a DNI model.

In Chapter 6, I present how DNI models can be extracted semi-automatically. I present a method for extracting the DNI traffic models out of *tcpdump* simplified traces. Moreover, I propose an optimization method to compact the extracted traces providing a trade-off between the accuracy of the representation and the model size. The proposed extraction method provides a valid partial DNI model that can be completed manually and solved using the presented approach.

Chapter 7 presents the validation of the DNI approach. In addition to the experiments, I present the experimental testbed used for experiments and the methods for practical model extraction, calibration, and solving. The approach is validated using two case studies: (1) *SBUS-PIRATES* case study from the project Transport Information Monitoring Environment (TIME) conducted at Cambridge University [BBE+08, Ing09b], (2) *Cloud file backup* based on the *L7sdntest* software that mimics a cloud backup/file exchange scenario. Additionally, I extend the validation focusing on the flexibility of the prediction process and presenting the trade-offs between prediction accuracy and solving time of the validated solvers. Moreover, in Section 7.5, I validate the traffic model extraction method using the *robot telemaintenance* case study, whereas in Section 7.6, I separately validate the *QPN-to-LQN* transformation.

Finally, in Chapter 8, I summarize the contributions of this thesis and formulate directions for future work.

Chapter 2

Foundations

In this chapter, I present the foundations of most relevant research areas that are used in this thesis. This includes: networks and network virtualization, modeling of network traffic, performance modeling and prediction in run-time and design-time phase, and the descriptive modeling languages.

First, in Section 2.1, I present generic data center network virtualization techniques following a coarse definition of virtualization. I identify techniques that constitute atomic building blocks for complex virtualization architectures. From the presented data center network virtualization architectures, I select two representative technologies—SDN and NFV—that are generic, promising, and have gained much attention from the industry and academia. I present the foundations of SDN and NFV in Section 2.2.

In Section 2.3, I briefly present general approaches to performance modeling of networks based on analytical models, gray-box models, and simulation. In Section 2.4, I present the differences between run-time and design-time aspects of performance modeling, and finally, in Section 2.5, I describe the basics of domain-specific modeling languages (DSMLs) and meta-modeling.

2.1 Generic Data Center Network Virtualization

In nowadays data centers, the use of virtualization techniques allows flexible assignment of resources to virtual machines (VMs). However, virtualization comes at the cost of increased system complexity and higher dynamics due to the introduction of an additional level of indirection in resource allocations and the resulting complex interactions between the applications and workloads sharing the physical infrastructure.

While sharing computational resources and main memory works relatively well in cloud computing, sharing of network resources is more problematic [AFG+10]. There are established mature solutions for system virtualization enabling efficient and fair sharing of computational and storage resources like, for example, virtualization platforms based on Xen [BDF+03] or VMware [SVL01]. Unfortunately, currently no such widely adopted standard approach exists for network virtualization in data centers. Software-Defined Networking (SDN) and Network Function Virtualization (NFV) are highly promising candidates for virtualization of enterprise data center networks, hence I discuss them separately in Section 2.2.

Due to the large amount of specific approaches to network virtualization, I focus on generic virtualization techniques that are typically used as building blocks for implementing concrete network virtualization solutions. I categorize various techniques and discuss their common aspects and different characteristics.

According to [FRZ14], "network virtualization is the separation of logical network structure from the physical network structure". In this section, I decompose the network into smaller parts and consider the virtualization aspect at the level of elementary network components: links, nodes, interfaces, protocols. I distinguish three categories of generic network virtualization techniques. Each category contains generic building blocks that enable the implementation of specific features as part of a virtualization solution based on these techniques. For the sake of completeness, I include the *Other Specialized* category to cover uncommon, highly specialized approaches as well. The categorization is depicted in Figure 2.1. The categories are discussed in the following sections.

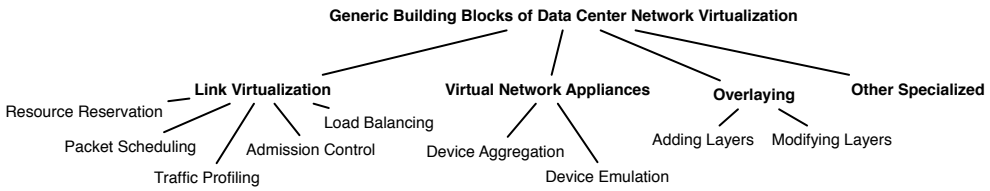


Figure 2.1: Categorization of generic data center network virtualization techniques.

2.1.1 Link Virtualization

Under the term *link virtualization*, I understand a way to transfer multiple separate traffic flows over a shared link (physical or emulated), in such a way that each traffic flow appears to be using a dedicated link referred to as virtual link.

Internet Engineering Task Force (IETF) developed the Integrated Services (IntServ) approach [RB94] to provide guaranteed bandwidth to individual flows. The guarantee is provided by a reservation of resources over an entire communication path using the Resource Reservation Protocol (RSVP). In modern packet-switched networks, traffic classification combined with packet scheduling algorithms is used to differentiate QoS levels. Based on the Differentiated Services (DiffServ) approach [BBC+98], probabilistic QoS guarantees are provided by classifying enqueued packets and dequeuing them according to predefined policies or advanced scheduling algorithms [GS08, GSR10]. In addition, a traffic profiling technique allows to limit the packet sending rate. In contrast to IntServ, the deployment of traffic profiling at the end hosts allows to avoid bandwidth reservation in switches if the cross traffic is not exceeding the capacity of a given path. Admission control techniques allow to drop incoming flows if the admission would exceed the available capacity or cause QoS degradation. Finally, load balancing mechanisms utilize multiple paths leading to a given destination by spreading the traffic over

separate routes and providing an illusion of a single link with increased capacity. Although some of these approaches can be treated as well known QoS mechanisms, they match the proposed definition of network virtualization by providing an abstraction of a separate network channel with defined performance parameters.

2.1.2 Virtual Network Appliances

A *virtual network appliance* is any networking device that does not exist in a pure physical form but acts like an analogous physical equivalent. I distinguish two types of virtual network appliances: (1) device aggregation where multiple networking devices act as a single logical entity, and (2) device emulation where an equivalent of a physical device is emulated by software. Emulation can apply to selected fragments or to an entire device.

Device Aggregation

VMware offers the vNetwork Distributed Switch [Zho10] that combines all virtual switches of a hypervisor into one logical centrally managed unit. Another example of device aggregation is the Juniper Virtual Chassis technology [Net11]. This feature allows up to ten switches to be interconnected and managed as a single virtual switch. Similar approaches are usually applied to provide a single point of management over the devices.

Device Emulation

Multiple VMs running on a single physical machine are normally communicating using a software switch (or a bridge) provided by a hypervisor. In this case, the functionality of a networking device is emulated by the virtualization software [SVL01]. Based on the incentives of NFV, a commodity server can be turned into a networking device using software emulation. There are several software solutions that provide such functionality, for example, Quagga [Sch09] or Open vSwitch [ope11], which implement functionalities of an SDN switch. Emulation of a physical device usually introduces additional performance overhead [CWD11]. I discuss more on NFV in Section 2.2.2.

2.1.3 Protocol Overlaying

An *overlay network* is a network resulting from a modification or expansion of a layer belonging to the ISO/OSI stack; in short, it is a method for building a network on top of another network [BCH+11]. The major advantage of overlay networks is their separation from the underlying infrastructure. Overlaying in networks consists mainly of *adding a new layer* to the existing stack of protocols by defining tunnels, or *modifying a layer*, for example, by introducing a new addressing scheme. Some virtualization techniques may use both approaches (adding and modifying a layer) simultaneously. Overlaying can be regarded as a link virtualization technique

however, I classify it separately to clearly separate different goals and a wider scope of these approaches.

Adding a Layer (Tunneling).

Tunneling consists of using one layer in order to transport data units of another layer—data units of one protocol are encapsulated in the data units of another protocol. The result of the tunneling is adding an additional layer to the default networking stack. For example, tunneling of layer 2 (L2) frames over L3 Internet Protocol (IP) creates a new layer between the L3 and L4 layers.

One of the most popular techniques that use tunneling in practice is Virtual Private Network (VPN). VPNs carry private traffic over a public network using encrypted tunnels. VPNs focus on security issues and do not provide QoS or performance isolation. Generic Routing Encapsulation (GRE) is a tunneling protocol that can encapsulate a protocol in an L3 protocol, for example, IPv6 over IPv4. Another example of an overlay network is Multi-Protocol Label Switching (MPLS) [ER01]. It operates between L2 and L3 of the International Standard Organization/Open Systems Interconnection (ISO/OSI) stack and is often referred to as layer 2.5 protocol. MPLS provides IP packet switching based on a short label instead of a long IP address. The subsequent classification and forwarding are based only on the label, accelerating the forwarding process. MPLS assures QoS similarly to the DiffServ approach. Due to mature traffic engineering functions of MPLS, it is mainly deployed by the Internet service providers as a replacement of ATM or Frame Relay protocols. However, deployment of MPLS in a data center requires compatible hardware. Overlay networks are often used to address limitations of the Internet or to provide new functionalities like, for example, provide local connectivity in distributed computing environments [GABF06].

Modifying a Layer

Modifying a specific layer consists mainly of providing a new protocol for that layer or changing the behavior of an existing one. The main goal of such a modification is to mend certain drawbacks of an existing technique or to provide a new functionality.

The VLAN technique [Soc05] provides logical isolation between broadcast domains in L2 by creating virtual subnets on top of a single physical subnet. It is a modification of the Ethernet consisting of adding additional fields to the Ethernet frame headers. In VLANs (802.1Q), the new addressing scheme consists of expanding traditional Ethernet frames by a VLAN ID which allows to create 4096 VLANs. The main limitation of the VLAN technique, namely a limit of 4096 VLANs in a network, is about to be eliminated by VXLAN which assumes 24-bit VLAN network identifier [vxl11]. Unfortunately, VLANs do not provide performance isolation and their QoS capabilities are limited to traffic prioritization (802.1p). Moreover, the Spanning Tree Protocol typically used in VLANs cannot

utilize the high network capacity of modern data center network architectures like, for example, fat-tree [AFLV08], DCell [GWT+08], BCube [GLL+09].

Another example of a layer modification is the networking infrastructure used at Facebook where the default L4 TCP has been replaced with a custom UDP transport layer to obtain lower latencies in their data centers [Rot09]. The main drawback of network virtualization based on overlaying is the performance overhead caused by processing packets in the additional layer or the lack of hardware support for modifications of layers.

2.2 Network Virtualization Technologies

In Section 2.1, I presented an overview of generic data center network virtualization technologies. In this section, I present in more detail two, that are the most relevant to this thesis: SDN and NFV.

2.2.1 Software-Defined Networking

In SDN, the network topology, the devices, and the functions are designed to be programmable similar to the software. The network functions are decoupled from the hardware and implemented as a software on the entities called controllers. Such architecture helps to overcome the limitations of network elements with fixed feature sets. New protocols, additional functionalities and highly dynamic adaptations are delivered as software with the advantage of flexibility and short release-cycles. SDN follows the following four principles.

Separation of control and data planes is the major novelty of SDN. In traditional networking devices, the control plane, containing the logic like routing algorithms, and the forwarding plane are implemented in one hardware device. This normally does not allow to modify any algorithms implemented in the control plane as it is programmed on a very low level, embedded in hardware and optimized for performance. This limits the flexibility of the network itself as implementing new algorithms require replacement of the device. In SDN, the intelligence is extracted into a separate control plane, so that each device can communicate with the controller over well defined API and protocol (usually, OpenFlow [MAB+08b]). The controller is a commodity hardware server (or a VM), so it can be flexibly programed by the user without cumbersome and expensive hardware upgrades.

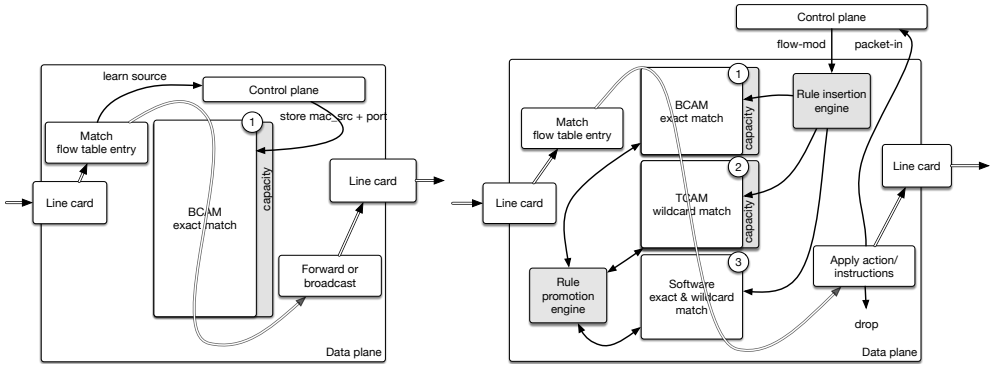
Logically centralized control enables configuration and management of the network from a single SDN controller. For reliability and load balancing purposes, the controller may be duplicated and may run on multiple physical or virtual instances.

Open interfaces and standard protocols are required to use SDN network devices from different vendors. Especially the communication between control and data planes relies on open protocols to facilitate different switch models and a vendor-independent controller.

Programmability means to influence the network logical topology and the forwarding behavior of packets by external software or applications in an flexible (i.e., changeable on-demand) fashion, so that the network functions can be purchased separately from the hardware. This allows the network applications run on top of the network and offer services regardless of the hardware model or vendor.

Performance-Relevant SDN Elements

SDN-enabled devices have undergone architectural and structural changes due to the numerous features that SDN offers. Here, I investigate the most important performance-relevant changes in the design of an SDN forwarding device. As this thesis aims at medium modeling granularity, I analyze the structure focusing on selected performance-relevant aspects.



(a) Classical (non-SDN). (b) SDN.

Figure 2.2: Comparison of typical forwarding pipelines of classical (non-SDN) and SDN devices.

The typical processing pipelines of a non-SDN and an SDN forwarder are presented in Figure 2.2. Typically in a non-SDN device (presented in Fig. 2.2a) the processing is conducted as follows. First, a network packet (frame, packet, or segment depending on the protocol) arrives to a line card and is passed to the device. Next, the device matches the destination address of the packet against the addresses stored in a flow table. The flow table is the local representation of device knowledge. It contains mapping of already seen destinations to the identifiers of the line cards of the device. Once a match is found, the packet is send to the outgoing line card and transmitted to its destination. Otherwise, the

device issues broadcast awaiting the response of the destination node, so that the information about the proper line card can be stored in the flow table. The architecture of classical flow tables is relatively simple. Data is usually stored in a binary content-addressable memory (BCAM) [PS06] which uses data search words consisting entirely of ones and zeros. In classical switches, the flow table maps an address of known length to the line card identifier. The memory has limited but rather high capacity and is optimized for performance.

The general processing in an SDN device is similar. However, an SDN device has complexer internal structure (as shown in Fig. 2.2b) to support more sophisticated features. In contrast to a classical device, SDN-enabled forwarder may have multiple logical flow tables that may by implemented using different physical memories [KPK15]. SDN devices use usually a ternary content-addressable memory (TCAM) memory which (in contrast to BCAM) may store a third state *don't care* denoted as $*$ or x. This allows the SDN devices to conduct wildcard matching. The matching of packet fields in an SDN device is more complex than in an classical one. A single flow rule located in a flow table may define many fields (e.g., ten in OpenFlow v1.0) against which the matching can be done. Additionally, a rule may specify an action that is executed and a set of statistics that are updated after a successful match. This complexity requires more advanced memory chips that are usually characterized with high energy consumption and price, so their capacity is strictly limited.

Multiple hardware vendors has addressed the limited capacity of hardware TCAM chips by implementing, so called software flow tables. The software flow tables do not have a designated chip in the forwarder but are usually placed in the SDRAM (synchronous dynamic random-access memory). This increases the maximal capacity of such flow table, however, the performance of forwarding using rules placed in software tables is drastically decreased [KPK15, RSK16, RSKK16].

An SDN device may contain two additional entities to manage the flow rules in the flow tables: rule insertion engine and rule promotion engine (marked in gray in Fig. 2.2b). These engines are responsible for selecting proper memory chip where an incoming rule should be installed and for moving the already installed rules between the flow tables respectively. Additionally, the rule insertion engine may return an error in case of incompatibility of a given flow rule with a memory implementing a flow table.

Finally, if no flow rule can be matched, the devices issues a *packet-in* message and sends it to the controller. The controller may respond with a *flow-mod* message, which contains instructions about new rules to be installed to handle future occurrences of packets from the same flow. The SDN controller is implemented using a dedicated software and deployed on a commodity server, so the modeling of its performance should be conducted using the techniques devoted for software performance modeling, for example, as presented in [RBB+11, KHBZ16].

Typical Use Cases for SDN

There are many application areas where the SDN concept may be used in practice [Jar14, SSHC+13]. The programmability and open interfaces allow programmers to develop new network software that can be deployed in an SDN controller and offer new network functions without hardware upgrade. This has lead to explosion of customized services that previously required a specialized and often expensive hardware to run.

From a general point of view, a network based on SDN can work in three modes: proactive, reactive, and hybrid. In a proactive mode, the SDN devices receive flows rules from an SDN controller *before* the affected traffic appears in the device. In the reactive mode, no preconfiguration takes place. Once an unknown flow arrives to a device, the SDN controller is queried for a rule and the rule is applied to the flow. Finally, the hybrid mode profits from the benefits of both modes. The network my be partially precofigured, whereas the unknown flows are forwarded to the controller that decides how to handle the flow. The controller may store a rule on an SDN device either for a predefined period of time (by setting the timeout parameter > 0) or permanently (timeout $= 0$). These possibilities enable, among others, scenarios that were previously nearly impossible to implement without a dedicated hardware, for example:

- centralized network management [KF13],
- firewall and traffic filtering [AMX15, HHAZ14],
- load balancing [KK12],
- policy-based routing [Fin13],
- application aware quality-of-service (QoS) [HsSL+14],
- network virtualization (e.g., Flowvisor [SGkY+09] or OpenVirteX [ASLG+14]),
- service insertion and chaining (typical to NFV) [QG14].

2.2.2 Network Function Virtualization

NFV is a new paradigm in networking, that moves network functionality out of proprietary hardware devices into virtual machines, running on industry-standard high-volume servers, switches, and storage [CCW+12, Tay14]. These functions include the typical—basic routing, switching, forwarding—and the complex—load balancers, firewalls, network address translators [Tay14]. In this paradigm, the aforementioned functions are implemented as virtualized network functions (VNFs) running in virtual machines on commercial off-the-shelf (COTS) hardware taking advantage of the economies-of-scale of the latter.

Any network functions that can be implemented with COTS equipment can be encapsulated and offered as VNFs. However, not all network functions are suited for virtualization. For example, it is challenging to virtualize high-bandwidth fiber-optic gateways, as these would require special equipment and logic for dealing with the unique hardware. The specialized hardware itself is the main feature to these function are not targeted for execution on commodity hardware. However,

the vast majority of network functionality is not hardware-dependent and can be virtualized [HGJL15].

NFV data plane. The data plane is responsible for the forwarding of data. Processing may involve modifying the data, for example, encrypting it. This usually stresses the input/output (I/O) or central processing unit (CPU) of the node. Because the device should be able to saturate the bandwidth of the network cards, this processing is sometimes done in dedicated application-specific integrated circuit (ASIC)or digital signal processing (DSP) components. Finding the destination for the forwarding is done via a forwarding table, managed in the control plane; the packet is passed up to the control plane if the processed traffic has yet unknown destination [SSHC+13].

NFV control plane. The control plane is responsible for setting up the routing/forwarding table on the device [SSHC+13]. The forwarding table is what the data plane uses to determine where packets should go. Additionally, the control plane might dictate what operations the data plane should perform on the data. This plane is not very I/O intensive and does most of its work in the CPU. Jobs performed include, for example: starting and stopping sessions, authorizing network access, and registering new devices in the network.

NFV management plane. The management plane represents the interaction with the device by network operators, including configuration and monitoring. The management is mostly manual, but it may also be automated. Here, configuration of the control and data plane are performed. In general, this concerns the initialization and fine-tuning of the device, but not the standard operation.

NFV in Relation to SDN

In traditional devices, the data and control planes are tightly coupled in the network device. (The management plane, as it plays more of an auxiliary role, is somewhat separate.) The data plane is bound to the hardware, and the control plane is intertwined. The goal of SDN is to decouple these two planes, so that the control plane may, for example, even be situated on a completely different device. This allows network intelligence and state to be logically centralized and consolidated [SSHC+13, FRZ14].

Neither NFV nor SDN are dependent on each other; they do, however, complement each other and are mutually beneficial. While NFV goals can be achieved without the separation of the data plane and control plane, the usage of SDN can simplify NFV, providing better configurability, integration, and performance. NFV benefits from SDN by providing infrastructure upon which the SDN software can run.

SDN shares some common ideas with NFV, for example, that SDN software should run on commodity hardware [CCW+12]. The two developments come

from different sources: SDN was proposed by university researchers, while NFV by network operators. Both had the goal of simplifying complex networking processes [Pat13].

Performance-Relevant Aspects of NFV

From the performance perspective, NFV setups depend on the performance of the underlying commodity hardware. There are two main hardware factors influencing the performance.

First, the performance of the data plane is dependent on the line cards of an NFV node. Commodity servers offer usually typical 1Gbps Ethernet network cards that matches a typical top-of-the-rack (ToR) switch setup. However, obtaining a typical port density (i.e., number of ports per device) is much more challenging than in network devices. The port density is usually limited not only by the physical size of the hardware, but also by the bandwidth of the bus with which are they communicating with the CPU and the system.

Secondly, the performance of the I/O–CPU subsystem of an NFV node defines the performance of the control plane and the network functions. This part is almost identical to any computing use case as the virtual functions and the control plane are delivered by software applications. Usually, the designers of an NFV node leverage specialized operating systems and thin middleware to optimize the performance and minimize bottleneck. The leveraging of commodity allows to model NFV system using usual performance modeling approaches from the computing and software architecture domains (e.g., Descartes Modeling Language (DML) or Palladio Component Model (PCM)).

2.3 Performance Modeling of Networks

Modeling of networks for performance analysis purposes is a well established field of science. Since the discovery of a telephone and emergence of telephony operators in 19th century, the capacities of circuit-switched telephone exchanges were challenging to model. A. K. Erlang modeled the arrivals of the telephone calls and provided the formulas for call loss and waiting times to provide optimal service to the customers [Erl09, Erl17]. Since that time, many models were proposed—analytical and later also simulations—for circuit and packet-switched networks. In this section, I briefly present the foundations of relevant models for network traffic and network structure.

2.3.1 Network Traffic Models

Network traffic models are usually expressed as a sequence of arrivals of discrete entities, such as packets, connections, calls, etc. They are usually represented mathematically using counting processes or interarrival time processes. A counting process is a stochastic process and defines how many arrivals were observed since

the beginning of the observation, whereas the an interarrival time processes is a random sequence that defines the length of the interval separating consecutive arrivals.

Analytical traffic models are usually required to represent specific characteristics of the traffic observed in real, for example, burliness or self-similarity. The oldest analytical model of traffic is the Poisson model [Erl09, GSTH08]. It assumes that packet arrivals are independent and their interarrival times are exponentially distributed with rate parameter $\lambda : P\{A_n \leq t\} = 1 - e^{-\lambda t}$. Unfortunately, the Poisson model is unable to represent burstiness of the traffic and is unsuitable for representing self-similarity [Man65], which are dominant phenomena in many nowadays data networks. Originally, Erlang proposed the Poisson model for legacy telephone circuits, which were not affected by these phenomena.

Many other traffic models were proposed to address the issues of the Poisson model, so that the realistic network traffic could be accurately modeled [WP98, FM94, QKW+04, RK96, SJLW11]. All of them focus on representing the statistical features of the originally observed traffic.

2.3.2 Black-Box and Gray-Box Performance Models

Black-box and analytical performance models abstract the modeled system to the highest degree. Such models require a defined set of inputs—which is usually a set of parameter values required in a formula—and return values of performance metrics as the output. I distinguish black-box and gray-box models based on the degree to which they include internal structure of the modeled system.

Black-Box Models

Black-box models are the most coarse performance analytical models. They do not consider the internal structure of the modeled system and do not allow to change any system parameters that do not belong to the input variables of the model (e.g., deployment of services, topology). Usually, such a black-box model is trained based on experimental data, so that the underlying mathematical model fits the characteristics observed in reality. According to [Lju01], black-box modeling (also known as system identification) should be rather called curve fitting. To examples of such techniques I account: statistical regression, linear and non-linear interpolation, differential equations, neural networks, Classification and Regression Trees (CART), Multivariate Adaptive Regression Splines (MARS), Kallman filters and others (some surveyed in [WHKF12, HLT09, SCBK15a]). Such models usually properly represent a concrete aspect of a system allowing rapid performance analysis of its features. On the other hand, such models are usually fitted to the measurement data and provide no guarantee to represent the general system behavior. Moreover, changes in internal structure of the system are also not allowed as the structure is completely abstracted—if the modeled system changes, the model must be newly built from new data.

Gray-Box Models

Gray-box models partially represent the structure of the modeled system. The degree to which the structure is included in the model is variable and depends on the level of detail included in modeling formalism. The most popular and well established formalism for modeling networks are **Queueing Networks (QNs)** [LZGS84].

QNs provide means to represent the resource contention in the modeled system. The formalism uses queues that consist of waiting lines and a service stations. The requests wait in a queue to be serviced in the service station which represents a limited resource. The service stations may represent one or multiple servers. The QN models may be parametrized in numerous ways, for example: the structure of the connections between queues in the QN, queueing disciplines, scheduling at a server, server processing rates.

An unification of the queue characteristics was proposed by Kendall [Ken53]. He proposed to describe each queue using a tuple of three parameters A/S/c which was later extended into six: A/S/c/K/N/D, The parameters have the following meaning:

— A is the arrival process (interarrival time distribution),
— S denotes the service time distribution,
— c denotes the number of servers,
— K denotes the capacity of the queue (i.e., maximal number of requests), equals to ∞ if omitted,
— N is the size of population from which the customers come, equals to ∞ if omitted,
— D is the queueing discipline, which defines the order in which the requests are dequeued form the waiting line.

The commonly used values denoting distribution parameters (A,S) are: M for exponential (Markovian); D for deterministic; E_k for Erlang-k; and G for general (i.e., unknown).

QNs wide applicability has lead to emergence of multiple analysis methods and laws which apply for selected QNs. Examples include: the Utilization Law, the Little's Law, the Response Time Law, the Forced Flow Law, the Pollaczek-Khinchine formula [LZGS84]. However, analytical solving of QNs face multiple limitations.

The most efficient analytical analysis of QNs is enabled for so-called product-form QNs [Bal00]. There exist efficient polynomial time algorithms for analysis providing a good balance between high accuracy of performance analysis and the efficiency of analysis. Unfortunately, the product form requires that several assumptions hold. The most important requirements are quasi-reversibility and partial balance [Bal00]. This applies mainly for models that can be represented by a set of QNs containing *M/M/1* queues. More complex QNs are generally difficult to solve analytically (either by exact or approximate algorithms); this applies to models with unknown distributions and queueing disciplines other than first in first out (FIFO). Simulation-based approaches are usually used for solving such complex cases (see Section 2.3.3). Further details on QNs have been presented in [BGdMT98, GSTH08].

Perti Nets (PNs) [Pet62] were proposed by Carl Adam Petri in 1962 with a main purpose to support modeling of concurrency, for example: synchronization, blocking, and software contention. A PN is a bipartite directed graph that consist of: (1) a finite set of places, (2) a finite set of transitions—separate from the set of places, (3) a finite set of forward and backward incidence functions, and (4) an initial marking. In an ordinary PN, places are interconnected with transitions using directed weighted arcs that define incidence functions. The initial marking defines the initial assignment of tokens to the places, whereas the further states of the PN are defined by the transitions and incidence functions that define the behavior of the net. A transition fires if all preceding places contain the respective number of tokens defined by the weights of the arcs. Once a transition fires, the tokens from the preceding places are consumed and new tokens are placed in the succeeding places in the quantity defined by the weight of outgoing arcs.

The ordinary PNs are missing the notion of time, so multiple extension to PNs were introduced. They include: colored Petr nets (CPN), which introduce multiple classes of tokens (colors); stochastic Petri nets (SPN), which add a firing delay to the transitions; generalized stochastic Petri nets (GSPN), which mix PNs with SPNs; hierarchical Petri nets (HPN), which allow to define subnets; and combinations of those, for example, colored generalized stochastic hierarchical Petri net (CGHSPN). Queueing Petri Nets (QPNs)—discussed in Section 5.2—mix QNs with CGHSPNs by adding a queue to each place.

Available tools for the analytical solution of QPNs are based on the analysis of underlying Markov chains [BK98]. However, the underlying Markov chains are usually too big for efficient analysis. In [Kou05], Kounev claims that "QPN models of realistic systems are too large to be analyzable using currently available analysis techniques".

Layered Queueing Networks (LQNs) [FAOW+09] are performance models that are an extension of regular QNs. Compared to ordinary QNs, LQNs introduce the concept of layers, software servers, and they allow the modeling of simultaneous resource possession. LQNs are usually used to model software and hardware contention in a uniform way, as well as scheduling disciplines, simultaneous resource possession, synchronization, and blocking [WNPM95]. LQN formalism has been developed as a DSML that covers wide range of computer systems with a special focus on software and hardware systems. In contrast to that, QPNs are general-purpose models and are not bound to any particular domain.

Woodside et al. [WNPM95] claim that "LQNs have a great advantage over the competing models (Petri nets, Markov chains, timed process algebras) that they scale up to large systems with dozens or hundreds of cooperating processes." Achieving such speed-ups in the solving is usually connected with abstracting selected data or limiting the modeling capabilities.

Heimburger [Hei07] analyzed the differences between the solvers for QPNs (*SimQPN* [KB06]) and LQNs (LQNS [FMW+09]) in the context of the performance prediction of Java Enterprise Edition (Java EE)-based software. Here, I generalize the comparison of the formalisms and briefly summarize the key differences in Table 2.1.

Table 2.1: Selected key differences between QPN and LQN formalisms.

Feature	QPN	LQN
Unit of flow	Colored tokens	Calls
Workload	Open and closed	Open and closed
Hierarchy	Yes, subnets, can be flattened.	Yes, layers, cannot be flattened.
Direction of flow	Any place with any transition	An activity to an entry, higher layer to lower layer only.
Loops	Yes, any type (including infinite), loop iterations can be modeled probabilistically or deterministically.	Yes, most of deterministically modeled loops (exceptions see Section 5.5.3), number of loop iterations must be known and finite.
Starting point	No explicit starting place or transition. Transitions that fire first can be calculated.	Top layer

In QPNs, the colored tokens represent the behavior of the model—they are deposited in places and are moved from place to place by firing the transitions. In LQNs, this function is realized by calls denoted normally as arrows pointing to an entry. Both formalism support modeling of open and closed workloads, however LQNs can be claimed to provide less support for closed workloads due to the limitations concerning loops that span multiple layers. Layers are used to represent the hierarchy in LQNs, whereas in QPNs, nets can be nested using subnet places. QPN tokens can be moved from a place to another place when a transition fires. A transition can connect any two places at a given level in the hierarchy. The tokens, however, can be forwarded (via input and output places of subnets) to any place or transition disregarding the hierarchy.

In contrast to QPNs, the LQN calls can connect only the layers that are non-higher than the layer from which the call originates. This limits the direction of the calls and narrows the modeling capabilities. The hierarchy of LQN layers cannot be flattened, whereas in QPN it does.

Another difference is the way the loops are modeled. The LQN formalism allows to explicitly model simple loops where the loop iterations need to be specified by a constant, finite value. QPNs do no support loops directly, however loops can be build easily using few places, tokens, and transitions. A loop built in this way can iterate over a defined number of times (also infinite) or the number of iterations can be specified probabilistically. Finally, QPNs do not have a predefined single starting point, whereas LQNs have so-called top layer where the execution starts.

Other formal, analytically solvable performance formalisms include: process algebras [Hil96, HHK02], Markov chains [BGdMT98, BH02, Her01].

2.3.3 Simulation Approaches

Simulation helps analyzing the models for which no efficient analytical or numerical analysis methods exist. Simulation offers the user a controlled environment in which a system can be investigated in more detail than using the analytical

methods. It often provides visualization and batch analysis tools (e.g., for a range of parameter values) to help understand the internal behavior of complex systems.

Numerous simulation tools were proposed for solving performance models. Complex QNs, for which no analytical method apply, can be solved with various simulation frameworks and tools. To examples of such tools I account: Java Modeling Tools (JMT) [BCS09], General Purpose Simulation System (GPSS) [Gor78], or SimPy [MV16].

QPNs can be efficiently solved with SimQPN [KB06], which is a simulator for QPNs that I use in this thesis. Its features and limitations have been characterized in Section 5.2 and 7.4.

In the area of network simulators, the most popular simulation frameworks are *OMNeT++* [Var01] and ns-2/3 [RH10]—both discrete event simulators. In [RH10], ns-2 is considered as the standard simulator for academic network research as it supports modeling of the most popular network setups and protocols. Examples of further similar simulators include: openWNS, OPNET, GTNetS, and IKR simulation library [WGG10]. All these simulators focus on medium-detailed modeling of the popular TCP/IP protocol stack and therefore are limited in their scope. Moreover, there exist simulators that are tightly bound to a given protocol or even to a concrete implementation of that protocol. An example is the Venus simulator [DLWJ08] where its authors use the TCP implementation extracted from the FreeBSD kernel for maximum simulation accuracy.

Despite more accurate modeling and higher prediction accuracy simulation approaches have also drawbacks. Simulation of large, complex models may require long solving times and consume high amount of resources—mainly CPU power and memory capacity. The complexity of a simulation-based performance analysis depends on the size and features of the analyzed model but also on the simulation implementation. Simple simulation scenarios may exhibit short solving times, however, the benefits come at the price of a trade-off between prediction accuracy, model size, or modeling granularity. For some scenarios (e.g., small and simple models), simulation may overlap with analytical methods providing similar results but with higher solving times when compared against analytical approaches. Therefore, simulation-based performance analysis is recommended for such application scenarios where high prediction accuracy is more important than the minimization of solving times.

2.4 Run-Time and Design-Time Aspect of Performance Prediction

The characteristics and differences between design-time and run-time performance prediction have been thoroughly investigated by Brosig in [Bro14a]. In this section, I summarize the main findings of Brosig and position his findings from the perspective of data center networks as the application area.

Goal

Run-time and design-time performance prediction approaches aim at estimating the performance of a system based on different data. Despite the aim, the goal of the modeling is different. Modeling in design-time targets the evaluation of various design alternatives before implementing and deploying the modeled system. Run-time analysis may benefit from the additional data available due to the measurements on the running system and thus, the goal of run-time analysis is to predict the impact of changes applied to the running system on its performance.

Structure of the Model

Run-time models benefit form the automatic extraction methods as the modeled system is available and can be represented in a model without involvement of a human operator. Although user-usability is an important factor, the major parts of a run-time model may be prefilled by a script and later tuned by an operator. Design-time models, on the other hand, cannot be extracted in most of the cases (e.g., with exception where partial models are extracted from code stubs or early Unified Modeling Language (UML) designs). Instead, the design-time models are targeted for manual modeling with separation of modeler roles, where domain experts model a part of the system in which they are specialized. This leads to emergence of many sub-models that are tuned for the specific needs of respective domain experts.

Availability of Input Data

Performance models require data to correctly represent the modeled system. Workloads, structure, configuration, hardware specification of the system components are inevitable for building a performance model. In design-time, only selected required parameters may be available (e.g., planned structure, deployment, or hardware specifications), whereas the rest must be estimated usually using analytical models or the data from similar existing systems. Unfortunately, even similar systems may perform differently and the vendor performance data does may not always accurately represent the real operation conditions (e.g., such extreme cases as the temperature of operation or different performance offered by different instance of the same model of hardware). This causes that the design-time models are usually coarser than the run time models as they need to incorporate possible performance estimation errors.

In contrast to this, run-time models are usually fed with realistic monitoring data which originate directly from the modeled system. The user models a concrete system instance deployed in a known environment. This allows to calibrate the model properly and minimize the possible errors caused by uncertain data sources.

However, some run-time scenarios may be still partially affected by uncertainty, for example, impact analysis of hardware upgrade. Moreover, the precise design documentation may be missing in some cases and the system needs to be abstracted, whereas the design-time models are assumed to have full knowledge about the

structure of the designed system. This shows that not only availability of the modeling data differs, but also the type of modeling data is different.

Solving Restrictions

A modeled system may be solved using performance solvers. Both approaches to performance modeling—run-time and design-time—use similar solvers as the purpose of modeling—performance—is the same. However, solving a design-time model experiences much looser time restrictions, as there is usually enough time for conducting fine-grained low-level simulations.

In contrast to this, run-time modeling scenarios may face near-real situations. An example of such scenario may be a forecast or a trigger caused by a workload spike that forces the system to reconfigure (e.g., scaling or migration of some components) in order to avoid service level agreement (SLA) violation. In such situations, the impact of possible reconfigurations of the system must be analyzed in a timely manner sacrificing the prediction accuracy and benefiting from the short solving times. Naturally, in other situations the time restriction can be looser and more accurate solvers can be used.

This difference in application scenarios leads to another difference between modeling approaches. Design-time approaches aim at minimizing prediction accuracy errors (assuming the uncertainty of input data), whereas the run-time approaches should provide flexible means to adapt the prediction accuracy and solving time based on the situation.

Differences in Change Impact Analysis

Finally, the art of changes which are subject of analysis may differ for the run- and design-time modeling scenarios. At the design-time, the spectrum of analyzed system configurations is wider. The designers may investigate various structures and configurations that are not available in run-time (e.g., including various network topologies or hardware types). The scope of run-time analysis is shifted towards reconfigurations, hardware upgrades, or redeployments as major reconfigurations including long system down-times are usually undesired. This allows to divide the running system into static and non-static parts and conduct the analysis for the latter assuming that the former remains unchanged. Therefore, the run-time analysis is more constrained and focused in contrast to design-time where almost any parameter of the system may be varied significantly increasing the number of possible designs to analyze.

2.5 Modeling Languages, Meta-Models and Descriptive Models

In this thesis, I often speak of modeling languages, formalisms, meta-models, descriptive models. In this section, I briefly introduce the concept of a meta-model and a descriptive modeling language.

There exist multiple definitions of models. In [Pid03], Pidd provides a general definition of a model as "a representation of reality intended for some definite purpose". If a model (M1) is representation of reality (M0), then the representation of a model (M1) is called meta-model (M2, model of model), and the representation of a meta-model (M2) is a meta-meta-model (M3), and so on. In practice meta-meta-models describe them selves and do not require a further level of modeling.

There are multiple definitions of a meta-model; I provide an exemplary one from Seidewitz [Sei03]: "A meta-model makes statements about what can be expressed in the valid models of a certain modelling language." In other words, a meta-model defines a modeling language by defining a valid set of models.

In this thesis, I distinguish descriptive and predictive models. Both types of model have their respective meta-models, however, a descriptive model *describes* a given domain without providing any other means of analysis. In contrast to this, a predictive model can be solved by at least one solver that provides performance prediction as the result of solving. In other words, a descriptive model is a structured form of data representation, whereas predictive model is an input to a solver that can conduct performance prediction.

I use *Ecore* for meta-model implementation in this work. *Ecore* is "the defacto reference implementation of OMG's (Object Management Group) Essential Meta-Object Facility (EMOF)" [Eco10, MOF14]. EMOF is one of two compliance points for Meta-Object Facility (MOF), whereas MOF is a closed meta-modeling architecture that defines a meta-meta-model (M3), which conforms to itself. MOF can be viewed as a standard to write meta-models.

In order to conduct model transformations, both input and output models must have their meta-models defined in MOF. However, some models may miss the definition of their underlying meta-model or the definition is provided using other meta-modeling standards. For example, *OMNeT++* defines its syntax using a Backus–Naur form (BNF) grammar, whereas LQN solvers described in Section 5.5 provide XML Schema Definition (XSD) to formally describe the format of their input files. Both formats—BNF and XSD—were transformed into MOF compatible meta-models for the need of this work.

Chapter 3

Related Work

I present the related work in two parts. First, in Section 3.1, I focus on the primary research contributions (see Sec. 1.3.1), whereas in Section 3.2, I present the related approaches to traffic model extraction, that is considered as secondary research contribution of this thesis (see Section 1.3.2).

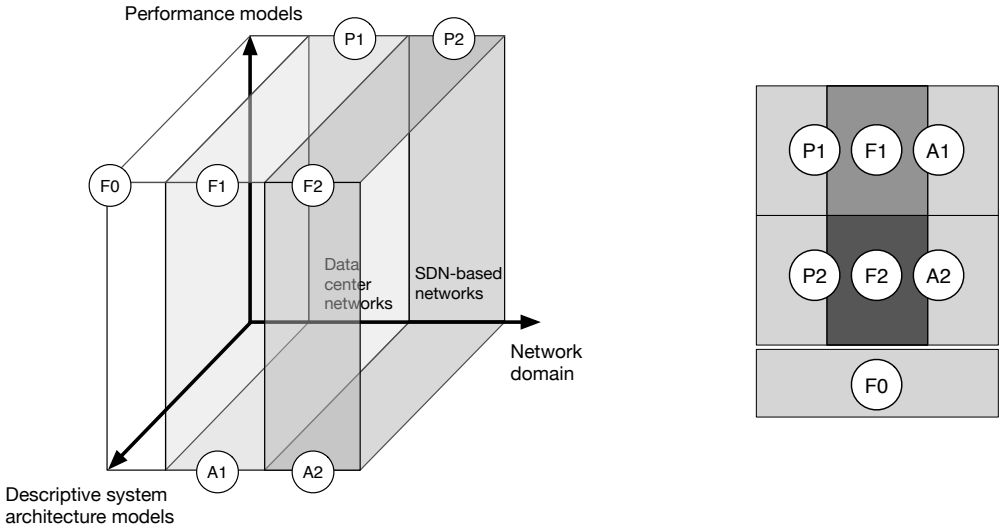
3.1 Related Areas

I review the state-of-the-art in the areas related to the main contributions of this thesis. To better structure the review, I consider related work along three different dimensions illustrated in Figure 3.1a. The first dimension is the *network domain*, distinguishing between approaches targeted at classical data centers and SDN-based networks. The second dimension concerns the *descriptive system architecture models* to represent the end-to-end system architecture including the computing infrastructure and the software running on top of it. Finally, the third dimension distinguishes the types of *performance models* and their scope in terms of the aspects they capture.

A detailed literature review on classical approaches to performance modeling of communication networks in general can be found, for example, in [Pui03, HP93]. In the following sections, I describe the state-of-the-art for each of the most relevant topics that are marked in Figure 3.1a as:

- F0, F1, F2: F like **F**ocus of this thesis,
- A1, A2: A like system **A**rchitecture models,
- P1, P2: P like **P**erformance models respectively.

In Sections 3.1.1 (P1) and 3.1.2 (P2), I review related work contributing performance models of classical or Software-Defined Networking (SDN)-based data center networks, however, without maintaining a link to the end-to-end system architecture (software, servers, etc.). Sections 3.1.3 (A1) and 3.1.4 (A2) describe contributions that provide descriptive architecture-level models, but do not model performance aspects. Finally, Section 3.1.5 (F0) describes architecture-level performance models that do not include networks, whereas Sections 3.1.6 (F1) and 3.1.7 (F2) review related work addressing all three aspects, that is, performance, architecture models, and SDN-based data center networks. The work presented in this thesis contributes directly to the areas F2 and F1.



(a) Three dimensions of related work.

(b) Venn diagram.

Figure 3.1: Related work divided into domains and relations between them.

The relations and intersections between the work presented in this chapter follows the Venn diagram presented in Figure 3.1b. I consider the performance and architecture models of non-SDN data center networks separately from the performance and architecture models of SDN-based networks (i.e., $P1 \cap P2 = \emptyset$, $A1 \cap A2 = \emptyset$). Analogically, to the area F0, I account architecture-level performance models that do not focus on networks.

3.1.1 Performance Modeling of Data Center Networks (P1)

There exists a lot of literature on performance modeling and evaluation of data center networks (e.g., [HP93]). Existing modeling approaches are mostly based on stochastic models such as classical product-form queueing networks, layered queueing networks, stochastic Petri nets, stochastic simulations models, and so on. I distinguish specific performance models (network simulators) and general purpose models that can be used to model data center networks.

The most popular network simulation frameworks are *OMNeT++* [Var01] and ns-2/3 [RH10]—both discrete event simulators. In [RH10], *ns-2* is considered as the standard simulator for academic network research. *ns-2* supports modeling the most popular network setups and protocols; modeling of custom networks—for which no models are provided in the standard libraries—requires manual implementation. Modeling custom network setups using *ns-2* requires extensive knowledge of programming in Tcl and C++ and thus it requires a significant time investment to learn how to build models. Similar observations apply to *OMNeT++*—

almost all non-standard protocol-specific models need to be implemented by hand at the required modeling granularity. Changing the modeling granularity (e.g., abstracting some highly-detailed information) requires changing the code of the custom model. Further similar simulators include: *openWNS*, *OPNET*, *GTNetS*, and *IKR* simulation library [WGG10]. All these simulators focus on medium-detailed modeling of the popular TCP/IP protocol stack; extensions to support further popular protocols are often available as add-ons. There are also simulators that are tightly bound to a given protocol or even to a concrete implementation of that protocol. An example is the Venus simulator [DLWJ08]; the authors use the Transmission Control Protocol (TCP) implementation extracted from the FreeBSD kernel for maximum simulation accuracy.

On the other hand, there exist many general-purpose performance modeling formalisms that can be applied to data center networks. To such formalisms I account, for example, queueing networks, layered queueing networks, stochastic Petri nets, stochastic process algebras, Markov chains, analytical estimation methods (e.g., bounds analysis). In [Pui03], Puigjaner reviewed selected general-purpose performance modeling formalisms and their applicability in the networking area. Each of these formalisms requires extensive knowledge and experience in order to be able to apply it to a specific domain—this limits the use of such formalisms to experts having experience in the given formalism and the respective domain. I described selected methods in more detail in Section 2.3.

3.1.2 Performance Modeling of SDN-based Networks (P2)

I now review some related work on performance models applicable especially in the area of SDN-based networks. I consider the general SDN networks as well as SDN-based data center networks.

According to [ANP+13], "in spite of active research around SDN and OpenFlow in particular, there are very few works to either address the performance issue of the SDN or OpenFlow as one of its early implementation or evaluation frameworks". The authors of [ANP+13] propose an analytical performance model based on network calculus. Additionally, they claim that "this is the first time that a network calculus-based analytical study is presented to model the behavior of SDN." The modeling approach presented in the paper is focused on performance evaluation of SDN switches and controllers. In contrast to my approach, the proposed model does not cover the computing infrastructure, the software architecture and the hosted applications.

The authors of [JOS+11] propose a performance model based on queueing theory to evaluate SDN-enabled switches. The evaluation focuses only on OpenFlow-enabled switches and controllers; the scope of the complete data center architecture is missing. Similarly, the authors of [BBGP10] focus only on a selected part of SDN-based networks—the data plane of a switch. The authors of [KJ13] proposed an SDN extension to *OMNeT++* simulation based on INET library. The proposed simulation models in detail the communication between an SDN switch and a single SDN controller. Nevertheless, given the simulation at the protocol-level,

their approach focuses on the specific network protocols supported by the INET library and misses the scope of the entire data center.

Other works, for example [TGG+12, KWS+13, BKLG14], focus on simple setups spanning an SDN switch and a controller, thus their scope is also limited and no end-to-end models of the complete system architecture are considered. In other works (e.g., in [GYG13]), the authors evaluate the influence of SDN on the network performance, however, no explicit models are provided.

3.1.3 Architecture-Level Modeling of Data Center Networks (A1)

I briefly review related work on descriptive architecture-level models applicable in the area of data center networks, however, without providing explicit support for performance analysis.

In [ITU00], the authors propose a formally complete language Specification and Description Language (SDL) that was originally designed for telecommunication systems. Its current areas of application are wider and also include process control and real-time applications in general. The language was also combined with Unified Modeling Language (UML) to extend its specification capabilities. However, the scope of the modeled systems is general with no specific consideration of performance aspects. The Common Information Model (CIM) [Bd05] is an open standard that defines objects of an IT environment and relations between them. Similarly to SDL, CIM does not provide any means to describe the performance aspects of network infrastructures. In [PTC12], the authors propose an architecture-level modeling approach designed for specifying the behavior of autonomic networks. The authors follow a similar approach to Descartes Network Infrastructure (DNI) (i.e., leveraging a descriptive model coupled with model transformations), but again the proposed models do not cover performance aspects.

The authors of [GRSS12] proposed LARES (LAnguage for REconfigurable Systems)—a formalism to model and analyze system dependability. LARES allows to model any system element by describing the probabilities failures and their consequences. The work described in [BCR+09] is driven by similar goals. The authors propose a modeling approach called SLIM to express the possible states of a system component in order to predict component failures and the possible propagation of errors. While the techniques used in LARES and SLIM are similar to the DNI approach (meta-models, model transformations), the goals of these approaches differ from my goals (dependability of systems vs. network performance).

3.1.4 Architecture-Level Modeling of SDN-based Networks (A2)

In [HHSDK14], Haleplidis et al. propose a model for SDN and Network Function Virtualization (NFV)-based networks. The authors consider SDN and NFV as being part of a bigger network picture and provide a common abstraction model. Their model supports “interoperability and homogeneity, as well as one protocol for control, management and orchestration of the network data path and the network

functions, respectively". The authors focus on modeling the network architecture using elements such as forwarding element, control element, and logical function block, however, without explicitly considering performance relevant aspects.

3.1.5 Architecture-Level Performance Modeling (F0)

In this part, I briefly discuss related work on architecture-level performance models that do not explicitly model the network infrastructure or that represent the network in a highly abstract and simplified manner.

The wide variety of existing performance modeling approaches makes it hard to select an appropriate model for a given scenario. A common approach is to use a descriptive architecture-level model (capturing the performance-relevant aspects of a system in a descriptive manner) and to then apply model-to-model transformations to automatically generate different predictive models (e.g., queueing networks or stochastic simulation models) that can be used for performance analysis.

Approaches based on model-to-model transformations originate mainly from the software engineering community. UML models are used to analyze various software-related metrics [BdMIS04]. Software architecture models are annotated with performance-relevant information and automatically transformed into predictive models. Examples include: UML with Stochastic Well-formed Nets [BM07], Performance by Unified Model Analysis (PUMA) [WPP+05], or Kernel LAnguage for PErformance and Reliability analysis (KLAPPER) [GMS07]. Similar work was surveyed by Koziolek in [Koz10]. Software architecture models have been extended to also include information about the hardware resources and the deployment of software components. Among the modeled hardware-related aspects also very simple network models have been considered (e.g., [RBB+11, CPSV08]).

In [Zsc09], the author discusses semantic concepts for the specification of non-functional properties of component-based software. He proposed a new specification language quality-modelling language for component-based systems (QML/CS) that can be used to model non-functional product properties of components and component-based software systems. As the modeling focus is put on software, the hardware resources and their non-functional properties are modeled in a black-box manner. The author proposed to represent the network resource demands in terms of required bandwidth. However, the network infrastructure and network traffic are not considered in this work. Moreover, the proposed approach focuses on the modeling language and lacks tooling required for analysis and solving of the models.

End-to-end performance analysis requires taking into account multiple performance-influencing factors such as computing resources, deployment middleware, storage, and networks. Networks are usually abstracted in models such as the above and represented as black-box statistical models. For example, in [BKR09], the authors use Palladio Component Model (PCM) to model software architecture at design-time. PCM represents the network as a linking resource, which represented as a black-box analytical function abstracting the network configuration, topology and traffic patterns.

3.1.6 Architecture-Level Performance Modeling of Data Center Networks (F1)

I now consider architecture-level performance models that provide support for modeling data center networks as part of the system architecture. In [dWK05], the authors propose to extend the SDL and UML languages with performance annotations. The obtained models are used for automatic generation of simulation models based on a proprietary simulator. The models are evaluated at the protocol level by simulating data center networks but also broadband links and mobile access networks. On the one hand, the focus is put on the protocol level, on the other hand, the scope is not limited to a given type of networks (e.g., mobile access networks or data center networks).

In [DDSG07], the authors proposed a modeling approach named Syntony. Syntony is used to model the Ad-hoc On-Demand Distance Vector protocol and compare the model-based analysis to the native *OMNeT++* implementation. Similar to [dWK05], the modeling is focused on the protocol-level. Similar approach is presented in [MTMC99].

The authors of [KO01] present a stochastic model for the window dynamics in TCP and investigate the throughput performance of TCP-Tahoe. According to the authors "the overall purpose is to investigate the impact of packet loss probability on the resulting TCP throughput". They use the ns-2 simulator to evaluate the performance of the TCP protocol, however, the modeling and performance prediction is limited to a particular version of the TCP protocol.

The I/O path model (IOPm) [KL12] was designed do model the architecture of parallel file systems. Although the focus is placed on the I/O system, IOPm is able to model simple storage networks in a data center. The network models are only used to detect bottlenecks and the scope is limited to storage networks.

3.1.7 Architecture-Level Performance Modeling of SDN-based Networks (F2)

Finally, I target the architecture-level performance models that provide support for modeling SDN-based networks (with and without the data center in their scope). To the best of my knowledge, there exist no works that model the performance of SDN-based data center networks at the architecture level, including the link to the computing infrastructure, the software architecture, and the applications deployed on it. The results of this thesis directly contribute to the areas F2 and F1.

3.2 Traffic Model Extraction

In Chapter 6, I describe a secondary contribution that aims at extracting network traffic models from *tcpdump* traces. Here, I present the related approaches to network traffic modeling and extraction.

As noticed by Adas, "Traffic models are at the heart of any performance evaluation of telecommunications networks" [Ada97]. On the other hand, the authors of [KSG+09] claim that "there is not much work on measurement, analysis, and characterization of data center traffic" suggesting that more focus should be put to modern data centers and the intra-data-center traffic characterization.

There exist many related work on modeling general network traffic [FM94, QKW+04, GS08, SJLW11]. Most of the works focus on probabilistic models that were meant to approximate the characteristics of network traffic when aggregated or to preserve self-similar nature of the traffic. However, the goals of the extraction proposed in this thesis are different. Here, I aim at representing the traffic deterministically to analyze the traffic exactly from the time it was recorded and not to generalize the model to larger time scales. To achieve this, I envision the following steps: (1) I decompose the traffic profile into a set of generators (on-off traffic sources) with defined start and end of their activity, (2) I flexibly compress the model of the network traffic at the same time being able to control the loss of the characteristics of the original trace (because some solvers do not accept large detailed inputs, for example: *SimQPN* [SKM12a]), (3) the approach supports any traffic aggregation interval, whereas the trace driven simulations use usually packet as a smallest unit of traffic and due to that produce fine-grained models with predefined, constant granularity.

The authors of [vKHZ+15] propose a similar approach to the method proposed in Section 6.2. They propose a tool that extract workload profiles (not network traces but rather service requests) and decompose them into patterns. The decomposed traces are stored in an Ecore-based models [SBPM09] and are used mainly for workload forecasting and replaying modified traces in benchmark environments (e.g., replaying an original trace but with amplified burstiness).

Although the approach is similar, the details clearly separate their work from this one. First, the workload model that LIMBO extracts is different to the DNI Traffic model, so the extraction procedure cannot be applied. I extract sets of traffic generators whereas LIMBO looks for patterns like, for example: seasonal, trend, burst. Second, I focus on network traffic models extraction by considering time series of the transferred data; LIMBO defines workload at the level of requests that can be mapped to various data sizes. And finally, LIMBO depends strongly on seasonality of the workload as the first step of their extraction procedure searches for data seasonal patterns (e.g., sine-shape). The DNI approach also supports seasonal patterns in the form of a set of ON-OFF traffic generators (see network traffic generator model in Section 6.2.1), however any other traffic characteristic can be modeled as well using the traffic generator representation.

Regarding the approaches to model extraction, the most of the approaches calculate traffic statistics from the traces and represent the traffic statistically, for example, using packet size distributions and packet interarrival times (as described in Section 2.3.1). Such approaches cannot be applied in the context of this thesis, as I aim to discover relatively compact set of traffic generators to represent the trace deterministically. Other works, for example [VV06], do model the structural information about the traffic, but this is usually represented as users,

sessions, connections and packets causing the approach to be application-specific. Additionally, in [VV06], there is no intention for flexibility of representation of the traffic, so that the trade-off between model size and accuracy of representation cannot be selected.

Chapter 4

Network Performance Abstractions

Managing the end-to-end performance in modern virtualized data center infrastructures is a challenging task and may be difficult to be addressed manually. Multiple management-related questions may appear, as presented in Section 1.2.1. Answering such question requires to consider the system in a changed state with respect to the current one. However, introducing the changes on a running system may cause service outages and lead to service level agreement (SLA) violations, therefore it is required to predict the impact of such changes considering the system run-time. I refer to this as *online performance prediction*, that is, I assume, that the user can observe the running system and build a model based on the observed data. The model can be later used to conduct a *what-if* analysis without reconfiguration of the real system. The requirements for *online performance prediction* are presented in Section 1.2.1.

In this chapter, I propose **new data center network performance abstractions** for use in online scenarios in the form of a **descriptive meta-model called Descartes Network Infrastructure (DNI)**. The DNI meta-model provides a new approach to model: (1) performance-relevant aspects of a network, (2) the performance of networks based on Software-Defined Networking (SDN) and Network Function Virtualization (NFV), and (3) load-balancing scenarios in data center networks.

Defining the scope of the proposed modeling language is challenging (see Section 1.2.1). I aim at finding a balance between the ability to model all important performance-relevant factors and the generic character of the meta-model. Including more detail in the model requires more time, data, and user experience to build fine-grained network models. On the other hand, abstracting too much information causes degradation of prediction accuracy.

The presented entities of the DNI meta-model constitute a definition of a new modeling formalism for run-time performance prediction in the context of data center networks. In the contrast to other models (as discussed in Chapter 3), DNI is not separated from the computing and software contexts, so it can be used for end-to-end performance analysis of modern IT infrastructures (when used together with Descartes Modeling Language (DML)). DNI is divided into three logical parts: structure, traffic, and configuration, so that the slightly different parts of the network infrastructure can be modeled using various tool or using the knowledge of various experts. The major criteria for the design of DNI are the following.

- Separation of concerns: I envision that DNI models can be created (or semi-automatically extracted) step-wise by engineers having expertise in different areas. To this end, the DNI meta-model is divided into three parts: (a) structure (modeled, e.g., by technical personal of a data center) describing the network topology, servers, and virtual machines; (b) configuration (modeled, e.g., by a data center manager) describing the protocols, algorithms, and routes of the network; and (c) traffic (modeled, e.g., by software engineers) describing the deployment of the software components on servers and specifying the characteristics of the traffic sources.
- Wide applicability: the modeling approach should be applicable to the major types of network infrastructures used in today's data centers. Finding a balance between the generic nature of the meta-model and the accuracy of the provided performance predictions is one of the main challenges addressed in this work.
- Support for network virtualization: the DNI meta-model should support describing virtualized network infrastructures. To demonstrate this ability, I apply the approach in the context of SDN. I include SDN-specific modeling elements to increase the expressiveness of the meta-model in the SDN domain without sacrificing its generic character.
- Integration with DML: the DNI meta-model should be integrated with DML to enable end-to-end performance analysis of the software, computing, and network infrastructure in a data center. DML is a sister-language for DNI that targets software architecture and servers virtualization.

The remainder of this chapter is organized as follows. In Section 4.1, I introduce the DNI meta-model focusing on the modeling of classical, non-virtualized network infrastructures. In Section 4.2, I extend the meta-model for modeling classical networks and present new entities used for virtualized networks (based on SDN and NFV). Section 4.3 discusses the flexibility of building DNI model with various level of detail to balance between the fine and coarse granularity of the model. Next, in Section 4.4, I present an approach to integrating the DNI meta-model with the DML meta-model, so that the models can jointly represent a data center. Finally, Section 4.5 summarizes this chapter.

4.1 Modeling Classical Networks

In this and the following sections, I present the DNI meta-model. To efficiently represent its elements in text, I use the following notation: the meta-model entities are presented using `verbatim` font face, (e.g., `Node` or `End Node` as combination of two entities: `Node` and `End`). Class attributes and objects are presented using *italics*, for example: *DatabaseServer* is a `Node` having `NetworkInterface` named *eth0* that has *interfaceThroughput* of $100Mbps$. On the other hand, in the diagrams, I use the

classical notation similar to Unified Modeling Language (UML) (stemming from Essential Meta-Object Facility (EMOF)) where italics denote abstract entities.

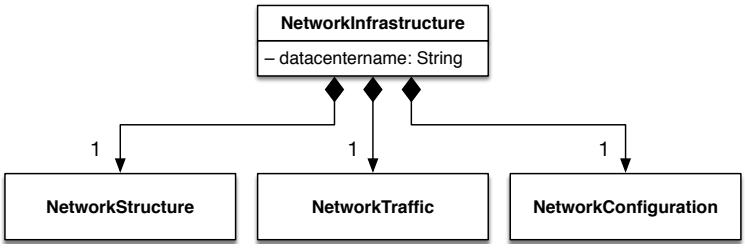


Figure 4.1: Root of the DNI meta-model.

The root element of the DNI Meta-model (`NetworkInfrastructure`) connects three main parts: network structure, traffic and configuration (see Fig. 4.1). To analyze the performance of any network infrastructure, one must know how the network is physically built (`NetworkStructure`), how it is configured (`NetworkConfiguration`) and how it is used (`NetworkTraffic`).

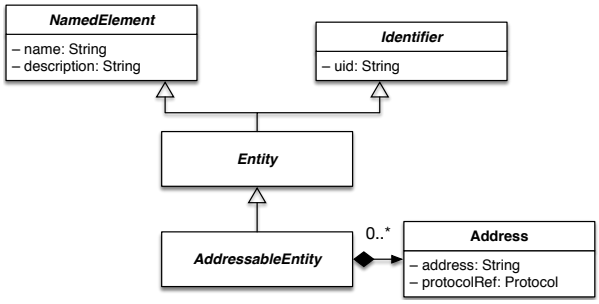


Figure 4.2: Core of the DNI meta-model: Entities.

DNI contains various kinds of entities. An `Entity` is named (`NamedElement`) to allow the user to assign an understandable name for each element and optionally provide additional description. From the modeling perspective, the entities are identifiable (using `Identifier`), so that each object can be distinguished from the others even if the set of object names is non-unique. Moreover, some entities can be additionally addressed using network `Address` as some users may prefer to identify the objects in this way. The core part that describes entities in the DNI meta-model is presented in Figure 4.2.

Every numeric value in the model is modeled as a `Dependency`—presented in Figure 4.3. A `Dependency` represents a `Variable` (constant or random) or a `Function`. Additionally, each `Dependency` can be accompanied with a `DNIUnit` that represent processing performance (`SpeedUnit`), data size (`DataUnit`), or time (`TimeUnit`). Examples of a `Dependency` can be the following parameter value descriptions: *"exponentially distributed with mean value of* $100ms$*"*, or just *"5Mbps"*.

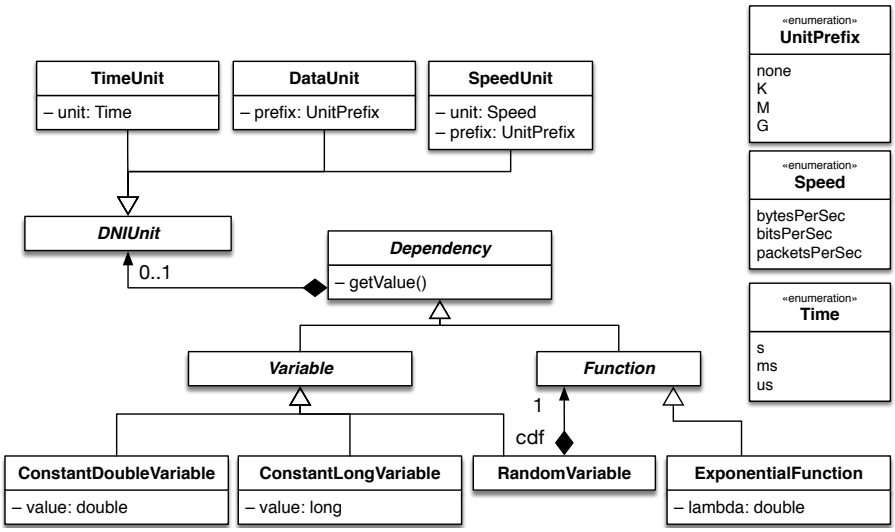


Figure 4.3: In the DNI meta-model, Dependencies are used to model numeric values and functions.

4.1.1 Network Structure

`NetworkStructure` represents the physical and logical network structure of a data center. The user should start building a DNI model by defining the topology of the network. The simplest representation of a network is a `Node`. A `Node` represents any computing, storage, or networking device as long as it has `NetworkInterfaces` or can host applications. To enable prediction, the solvers used by DNI require to use at least a single `NetworkInterface` via which traffic is traversing.

The DNI meta-model representing the network structure is depicted in Figure 4.4. For now, I abstract the SDN-related entities; they are discussed in Section 4.2.

Node

The `NetworkStructure` is a graph consisting of `Nodes` and `Links` connected through `NetworkInterfaces`. I assign a `IPosition` to a `Node` to indicate its role in the topology. An `End Node` represents a device (physical or virtual) that produces or consumes network traffic but does not forward it. On the other hand, an `Intermediate Node` forwards traffic but does not produce nor consume any (an exception to this rule is described in Section 4.2). A `Node` can be also have both `IPositions` simultaneously (i.e., `End Intermediate Node`) to represent a node that forwards and produces or consumes the traffic. I distinguish `End Nodes` (e.g., virtual machine, server) and `Intermediate Nodes` (e.g., switch, router), because their performance descriptions are different, for example, end nodes do not utilize information about forwarding performance. Moreover, only `End Nodes` can host `CommunicatingApplications`.

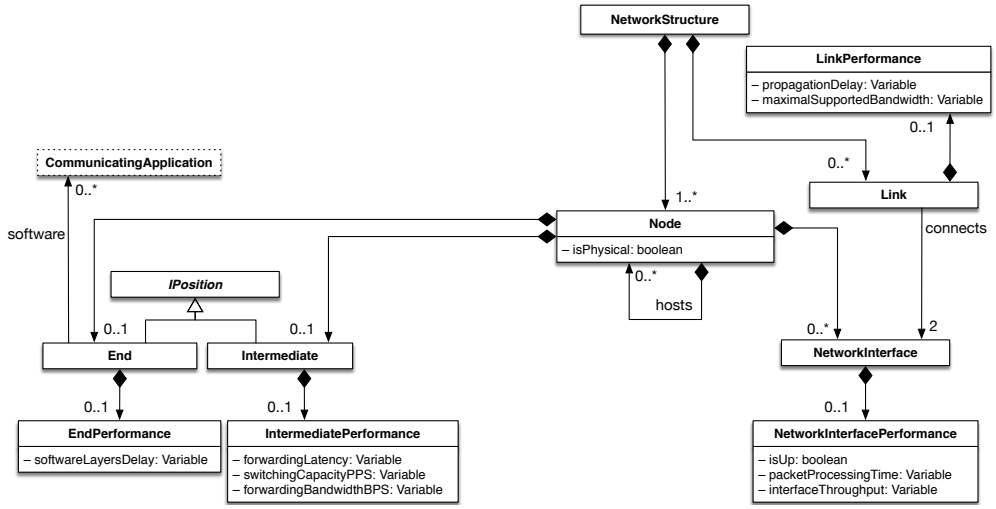


Figure 4.4: DNI meta-model of network structure.

To represent virtual and physical `Nodes`, I propose the *hosts* relation, that allows a `Node` to be hosted on another `Node`. The `Node`'s property *isPhysical* is automatically derived and tells if a node is virtual or physical. According to the Figure 4.4, any two `Nodes` can be connected with a `Link` over `NetworkInterfaces`. I limit the valid connections of virtual in the following way: *A virtual Node can be connected only to the physical not that hosts it or to other virtual nodes that share the same host*. This constraint is implemented using Object Constraint Language (OCL) to disallow building invalid models.

The performance description of an `End Node` and an `Intermediate Node` differ from each other. The `EndPerformance` describes coarsely the delay that is incurred by the software layers (parameter *softwareLayersDelay*), whereas the `IntermediatePerformance` concerns the performance of forwarding. The forwarding performance is described using the following parameters:

- *forwardingLatency*, which is added to every forwarded data unit;
- *switchingCapacityPPS*, which describes how many data units can be processed in an unit of time (expressed in units `packetsPerSec`);
- and *forwardingBandwidthBPS*, which describes how much data the forwarding engine can handle (expressed in units `bitsPerSec` or `bytesPerSec`).

The *PPS* and *BPS* suffixes serve as suggestions for the modeler which units should be used. The meta-model itself does not forbid setting incorrect units. Enforcing to do so at the meta-model level would make it more complex, so the allowed units are defined using OCL constraints.

The total forwarding delay of an `Intermediate Node` is calculated in the model transformations. The calculations are suggested to follow the following recommendations. Let $d_f(msg)$ denote the total forwarding delay of a message msg of

size $size(msg)$ on a node, $f_latency$ denote *forwardingLatency*, $capacity$ *switchingCapacityPPS*, and $bandwidth$ the *forwardingBandwidthBPS*. Assuming, that the unit of $size(msg)$ and $bandwidth$ is consistent (bits and bitsPerSecond or bytes and bytesPerSecond), the total forwarding delay is expressed by the Equation 4.1. The forwarding delay does not include the delays incurred by `NetworkInterfaces`.

$$d_f = f_latency + max(\frac{1}{capacity}, \frac{fb}{size(msg)}) \tag{4.1}$$

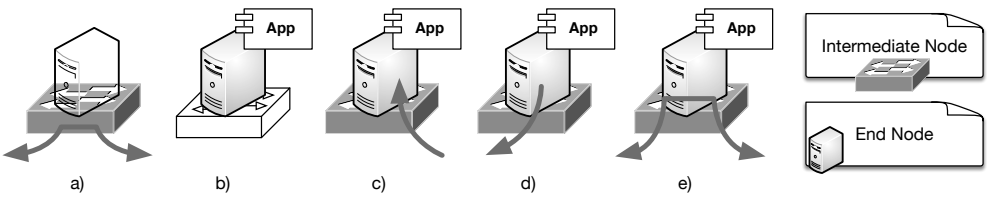


Figure 4.5: Example of various `Node IPositions` combinations: a) intermediate, b) end, c) end+intermediate (receiver), d) end+intermediate (sender), and e) end+intermediate (forwarder).

The forwarding delay is calculated differently for a node that is an `End` and `Intermediate` at the same time. Possible combinations of `Node IPositions` are depicted in Figure 4.5. For example, a switch based on NFV (Fig. 4.5e), the forwarding delay is calculated as in Equation 4.2, where $s_latency$ is the value of the *softwareLayersDelay* parameter.

$$d_f = 2 * s_latency + f_latency + max(\frac{1}{capacity}, \frac{fb}{size(msg)}) \tag{4.2}$$

On the other hand, in case b) and c) the forwarding does not happen, so the total node delay equals $s_latency$, whereas in case d) the total delay is the sum of $s_latency$ and d_f as presented in Equation 4.1.

Note that all performance parameters are specified as `Variable` and in some cases might be specified as random variables or functions. This allows various modeling approaches and gives the modeler freedom to select the most suitable one. Moreover, the model-to-model transformations that use DNI as input may interpret the performance parameters differently or even do not support some combinations of the parameters.

Network Interface

DNI `Nodes` are connected using `NetworkInterfaces` and `Links`. The performance description of a `NetworkInterface` (entity `PerformanceNetworkInterface`) includes: *isUp* flag, *interfaceThroughput*, and *packetProcessingTime*. The parameters are understood as follows:

- *isUp*: if set to *false*, all traffic directed to this interface will be dropped;
- *interfaceThroughput*: defines the throughput of the interface, usually specified using *bps* units;
- *packetProcessingTime*: adds a constant delay to every data unit traversing this interface. The delay is applied additionally to the delay that is calculated based on the *interfaceThroughput* parameter.

Moreover, the traffic traversing a `NetworkInterface` is decapsulated or encapsulated using the *dataPayload* information from the `NetworkProtocols` that the `NetworkInterface` implements. The connection between `NetworkInterface` and `NetworkProtocol` in the meta-model is depicted in Figure 4.6. Every `NetworkInterface` is a `AddressableEntity`, which has a reference to the `NetworkProtocol` that provides the address and thus sets the *dataPayload* and *packetOverhead* values. If a `NetworkInterface` has more than one address, the *packetOverhead* is calculated as a sum of the *packetOverhead*s provided by the protocols in the `ProtocolStack` that is referenced by the addresses. As a `NetworkProtocol` may belong to various `ProtocolStacks`, the relevant protocol is referenced directly from the `NetworkInterface` using the *usedProtocolStack* parameter.

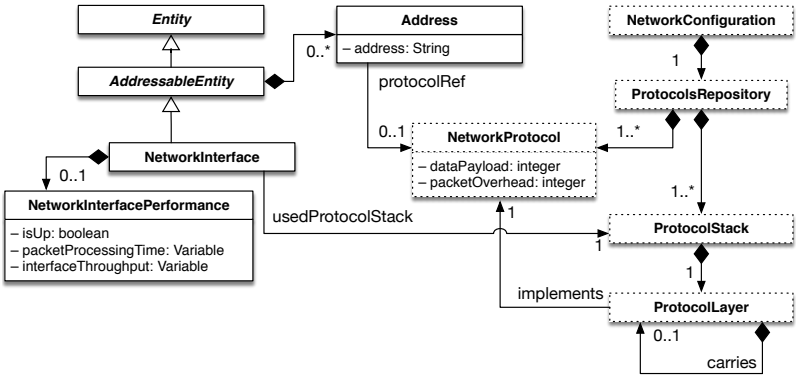


Figure 4.6: Relation between `NetworkInterface` and `NetworkProtocol` in the DNI meta-model. Dotted entities originate from the NetworkConfiguration part of the DNI meta-model (presented in Section 4.1.3).

Link

In DNI, `Link` represents a physical or virtual medium over which the traffic is transfered. A `Link` is virtual if it connects `NetworkInterfaces` that belong to at least

one virtual Node; otherwise, the Link is considered physical. Physical Link have the performance constrained by the *propagationDelay* that is limited by the speed of causality (also known as the speed of light). The *maximalSupportedBandwidth* is provided for the comfort of the user and is supposed to be less exact than the *propagationDelay*. The *propagationDelay* and *maximalSupportedBandwith* may be propagated back (or forward) to the connected NetworkInterfaces in a model-to-model transformation. For example, the *NetworkInterface.packetProcessingTime* is increased by the value of *Link.propagationDelay* and the *Link.propagationDelay* is set to 0.

4.1.2 Network Traffic

In a data center, most of the network traffic is generated by deployed applications. As depicted in Figure 4.7, in the DNI meta-model, network traffic is generated by TrafficSources that originate from CommunicatingApplications. CommunicatingApplications are deployed on End Nodes. Each TrafficSource generates traffic Flows that have exactly one source and possibly multiple destinations. The Flow destinations are located in CommunicatingApplications, so they can be uniquely identified. Each TrafficSource can generate a set of Flows. The information about the precise transmission time of a flow is modeled in the workload model.

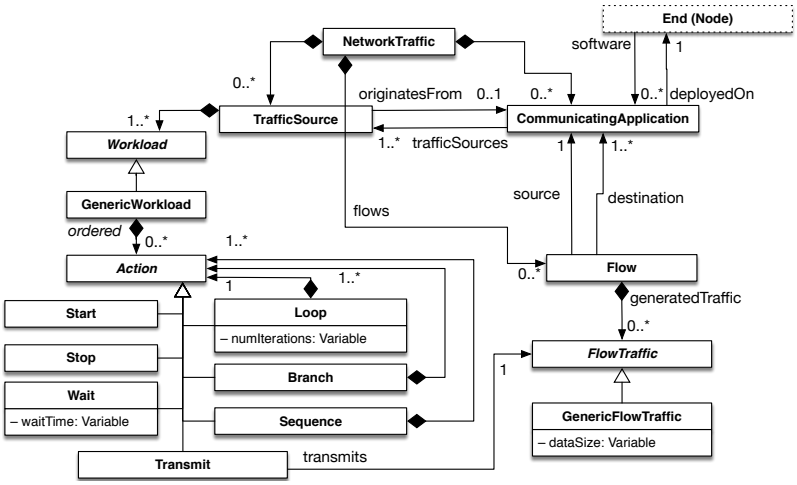


Figure 4.7: DNI meta-model of network traffic.

DNI supports modeling of open workloads. Currently, DNI supports a GenericWorkload but the set of supported workload models can be extended to support more compact (and thus probably more abstract) alternative descriptions (e.g., as presented in [vKHK14, Ada97]). A GenericWorkload consist of Actions that include Wait, Transmit, and additional meta-actions and containers: Start, Stop, Loop, Branch, Sequence. The actions Start, Stop, and Sequence play important

role for modeling languages that do not provide containers with *order* relation. An example of a workload is presented in Figure 4.8 using object diagram.

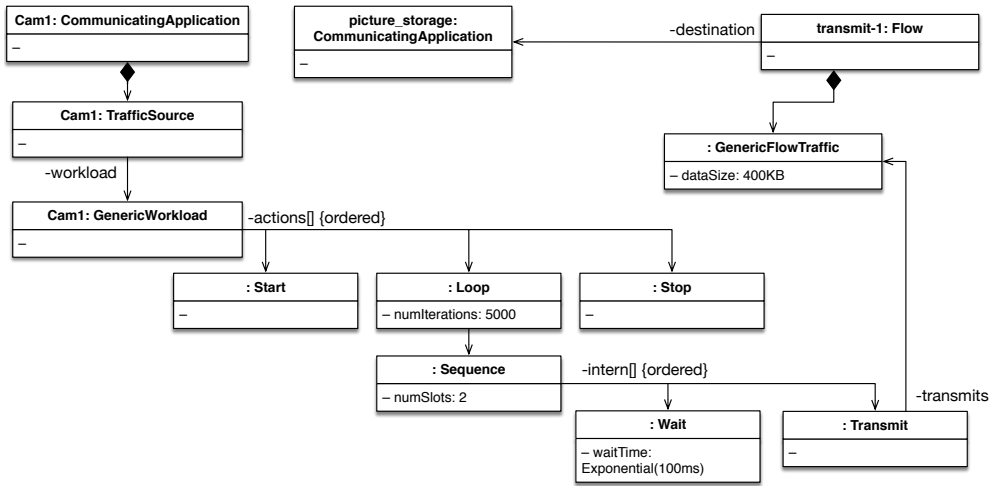


Figure 4.8: Example of the DNI model presenting the workload. Depicted using the UML object diagram.

The example in Figure 4.8 represents a fragment of a system where cameras take pictures and transmit them to the storage. The *Cam1* application takes 5000 pictures, each every 100 milliseconds on average (the *wait-time* is exponentially distributed), and sends it to the *picture-storage* application. Each picture is modeled deterministically as data of constant size $400KB$. Note that the traffic model does not define the path (or route) which is used for transmission.

Moreover, the `Flow` entity is a pair of source and destinations. `Flow` does not include any other traffic characteristic. Thanks to this, it is a flexible modeling element, as all other descriptions of the traffic characteristics (data size and the workload pattern) are modeled orthogonally. Each flow can be described by means of various flow descriptions; currently DNI supports a `GenericFlowTraffic` description that captures the size of transferred data. The meta-model can be extended to support other traffic models from the literature, for example, presented in [FHH02, KMF04].

4.1.3 Network Configuration

The third part of the DNI meta-model—`NetworkConfiguration`—contains the information about network paths, protocols, and protocols stacks. The model represents a snapshot of the current network configuration. This assumes that, for example, the routes in the network are modeled statically for a given moment of time. DNI does not support dynamic routing as this would require including more

detailed information in the model (e.g., routing algorithms), which is intentionally abstracted here.

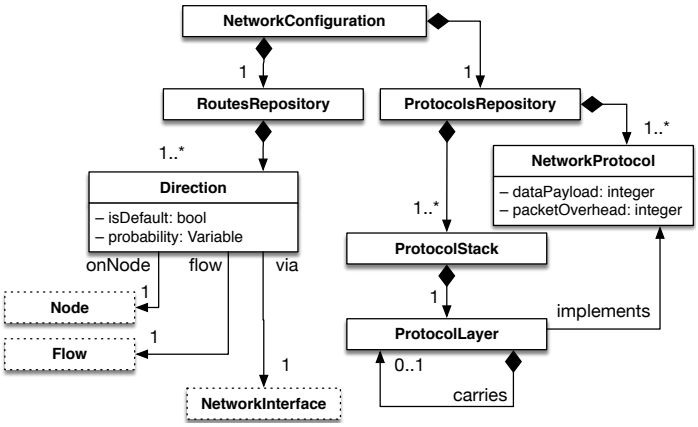


Figure 4.9: DNI meta-model of network configuration. Entities in dotted boxes represent the entities form the other parts of the DNI meta-model.

The network-configuration meta-model is presented in Figure 4.9. In the meta-model, a route or network path is represented as a set of `Directions`. Each `Direction` consists of three references to: `Node`, `NetworkInterface`, and `Flow`. It should be read as: *On node `onNode`, the flow `flow` should be forwarded over network interface `via` with probability `probability`.* Note that the flow includes the information about its destination, so it does not need to be specified in the `Direction`. The probability defines the behavior of load balancing on the node. It is required that the sum of destination probabilities for a given node and flow is larger than 0. The probabilities may be normalized in the model transformations (presented in Chapter 5) if they do not sum up to 1.

An exemplary fragment of the DNI model is presented in Figure 4.10. The traffic flows from *App1* to *App2* via nodes *N1*, *N2*, *N3*, to node *N4*. The included object *d: Direction* presents a fragment of the path configuration that says that the flow f on node *N2* should be forwarded via port *p1* with probability 0.5. This rule affects the half of the forwarded traffic. It is not said what happens with the other half; by default it will be dropped. Some model-to-model transformations may normalize the probabilities and interpret this as the entire traffic will be forwarded via *p1*. It is important that each transformation clearly specifies the way in which this parameter is handled.

A `ProtocolStack` is an ordered set of `ProtocolLayers`, where each `ProtocolLayer` references a single `NetworkProtocol`. The `NetworkProtocol` itself is described in a minimal way by specifying the overhead introduced by the protocol (parameter *packetOverhead*) for each portion of data (parameter *dataPayload*). The overhead may also include other factors as, for example, the retransmissions of lost data units if the modeled protocol offers delivery guarantee (e.g., the Transmission

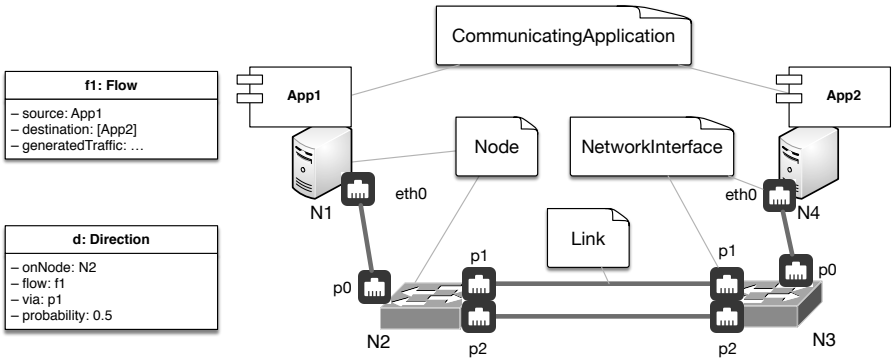


Figure 4.10: Example presenting a fragment of a DNI instance using both: unofficial graphical notation and object diagram elements. The Direction *d* defines that half of the traffic from *App1* to *App2* should be routed via port *p1* on node *N2*.

Control Protocol (TCP)). For example, a `NetworkProtocol` with *dataPayload*=1500B and *packetOverhead*=42B adds 42B overhead to each 1500 bytes of data payload block. At the lowest layer of the protocol stack, the *packetOverhead* should include not only the data unit headers (e.g., 22 bytes for Ethernet II) but also all interframe gaps, preambles, and start-of-frame delimiters (20 bytes in total for Ethernet II). The calculation of the total overhead of a protocol stack is conducted in the model transformations.

4.2 Modeling SDN Networks

In this section, I present the SDN extensions to the DNI meta-model. Each of the three parts: structure, traffic, and configuration is presented again in a wider context. I focus in the description on the newly added entities that are relevant for SDN-based networks.

4.2.1 Processing in an SDN Node

The DNI representation of network traffic processing in the SDN mode approximates the behavior of a real SDN node. In this section, I present the main differences between the SDN and the native traffic processing. The foundations of SDN processing are described in Chapter 2, Section 2.2.1. For each `Node` in DNI, I define four processing possibilities.

1. Native: when SDN is disabled or not supported by the node.
2. Hardware SDN switching: takes place when the modeled node contains a *hardware flow table* and the SDN rule that matches the incoming flow is located in this table.

3. Software SDN switching: takes place when the modeled node contains a *software flow table* (usually implemented in SDRAM of the node) and the SDN rule that matches the incoming flow is located in this table.
4. Processing in the SDN controller: no SDN rule can be found in the flow tables of the node, so the forwarding decision has to be made by the SDN controller.

The native behavior is described in detail in Section 4.1. In the following sections, I describe how the proper SDN processing mode is selected and how the forwarding is conducted.

4.2.2 Network Structure

The `NetworkStructure` part of the meta-model was extended by the `IType` entity that defines the type of `Node` either as `Common` or `SDN`. In this section, I focus on the `SDN Node`. The `SDN Node` can process traffic using hardware- and software flow tables, for which the performance is described using *hardware-* and *softwareSwitchingPerformance* parameters respectively. Lack of any of these performance properties means that the respective switching mode is not supported by the node. In an extreme case when both parameters are not set, a warning shall be returned to the user and, by default, an infinite performance should be assumed. The meta-model including the SDN-relevant entities is presented in Figure 4.11.

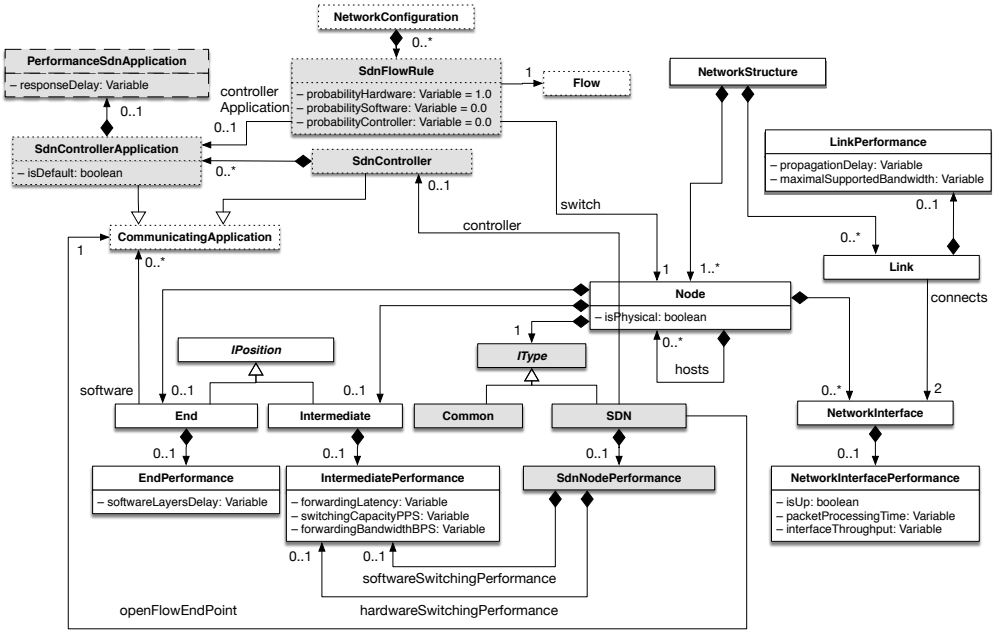


Figure 4.11: Meta-model of network structure with SDN entities included (in gray). Dotted entities represent other parts of DNI.

Traffic arriving to an `SDN Node` should be forwarded using one of three SDN forwarding modes: hardware, software, or via the controller. The selection of the forwarding mode is based on the `SdnFlowRules`. An `SdnFlowRule` defines the probability for the given flow, on a given node to be forwarded using the: software switching mode, hardware switching mode, or the SDN controller.

The probabilistic modeling of the SDN node behavior is an abstraction of a complex real behavior. In DNI, the probabilities define the chance that *a message* (defined by the `GenericFlowTraffic` in `NetworkTraffic`) is processed using the given mode. In the reality, the node may forward first packet of each flow to the controller to request a decision regarding the forwarding mode. The decision is later applied for all packets that belong to the same flow and arrive before a timeout occurs. In DNI, this behavior is modeled more coarsely as a result of finding a balance between the extremes of modeling granularity.

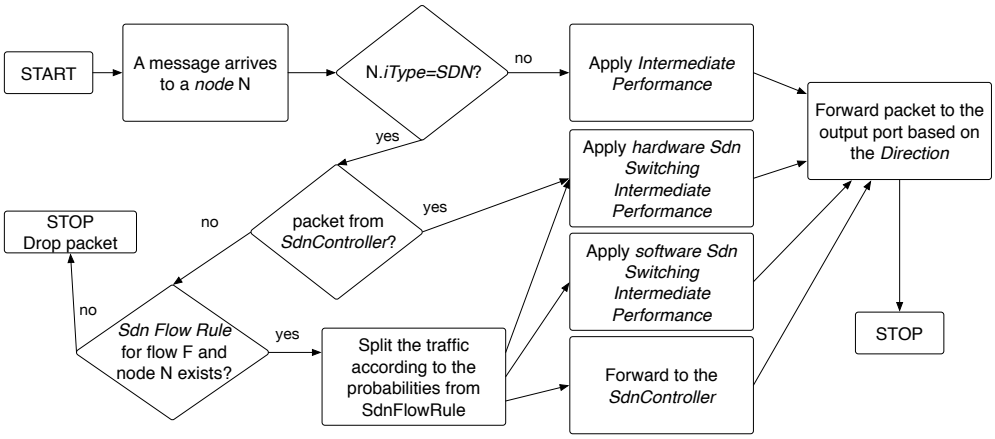


Figure 4.12: Flow diagram for forwarding in a DNI Node.

The processing of incoming traffic in an `SDN Node` is presented using a flow diagram in Figure 4.12. After a message arrive to the node, the type of the node is analyzed. For a `Common Node`, the normal `IntermediatePerformance` is applied and the message is forwarded to the output port. For an `SDN Node` the processing continues. In the next step, the node checks if the message originates from the SDN controller. If no, the node makes a rule lookup and selects a `SdnFlowRule` that matches the flow and the node. If the node is unable to find any `SdnFlowRule` the message is dropped. In the other case, the `SdnFlowRule` is read and the probabilities are used to select the processing mode. Then, the message is forwarded according to hardware, software, or SDN controller processing. If the message is forwarded to the controller the processing is paused and a *packet-in* message is forwarded to the controller. The processing in the node is paused until the controller responds with a *flow-mod* message. If a *flow-mod* message arrives back to the node, the paused message is processed further using the arbitrarily

chosen hardware switching mode. Arbitrarily selecting the hardware processing mode is an approximation of the real behavior.

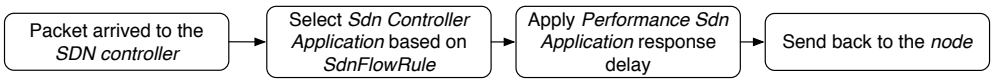


Figure 4.13: Flow diagram for processing a *packet-in* request in an SDN controller.

The processing of *packet-in* messages in the `SdnController` is approximated as presented in Figure 4.13. Once the *packet-in* message arrives to the controller, the proper `SdnControllerApplication` is selected (based on the `SdnFlowRule`), the processing delay is applied (defined in `PerformanceSdnApplication` entity) and the *flow-mod* message is returned to the node. The transmission of the *flow-mod* and the *packet-in* messages is conducted according to the topology and directions specified in the rest of the DNI model.

4.2.3 Network Traffic

The `NetworkTraffic` part of the SDN DNI meta-model includes three new entities: `SdnController`, `SdnControllerApplication`, and `PerformanceSdnApplication`. Both `SdnController` and `SdnControllerApplication` inherit form `CommunicatingApplication` and are deployed in `End Nodes`. `SdnController` is a container for `SdnControllerApplications`. `SdnController` normally does not generate traffic by itself (although it is not forbidden in DNI). `SdnControllerApplications` represent the behavior of the SDN Controller, which can handle each flow differently. In DNI, the behavior of the controller is modeled in a *black-box* manner by coarsely specifying the delay between receiving a *packet-in* messaged and responding with a *flow-mod* message. The `PerformanceSdnApplication` entity is depicted in Figure 4.14 using dashed line, because it approximates the more detailed software description offered normally by DML.

Each `SDN Node` may communicate with the `SdnController` using the same infrastructure as the other DNI `nodes`. A `Flow` must be defined for the communication between an `SDN Node` and the SDN Controller. Defining a `Flow` requires both nodes to host `CommunicatingApplication`, but only `End Node` is allowed to do so. To allow communication of any `SDN Node` with the `SdnController`, I introduce the *openFlowEndPoint*. The *openFlowEndPoint* allows to "deploy" a `CommunicatingApplication` on an `SDN Node` disregarding its `IPosition`. In this way also the exchange of *packet-in flow-mod* messages can be conducted over a path (defined by `Direction` entities) that is defined in the configuration part of DNI.

4.2.4 Network Configuration

The SDN version of DNI's `NetworkConfiguration` includes a set of `SdnFlowRules` that were discussed in Section 4.2.2. An `SdnFlowRule` connects a `Node` and a `Flow` with an `SdnControllerApplication`. It defines also the probabilities that specify

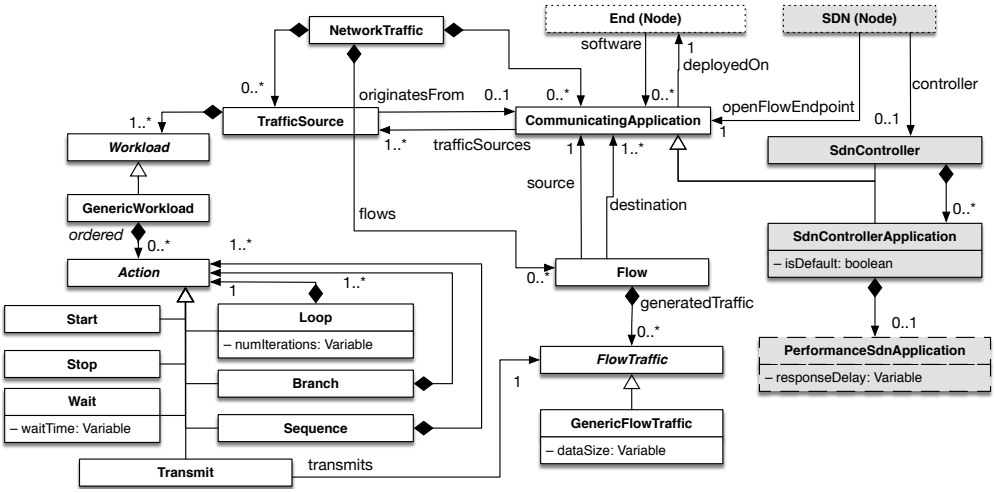


Figure 4.14: The DNI meta-model of network traffic with SDN entities included (in gray). The dashed entity `PerformanceSdnApplication` is a simplified representation of software component performance description that can be modeled using DML.

the forwarding mode of an `SDN Node`. `SdnFlowRule` do not apply to `Common Nodes`. The SDN version of DNI's `NetworkConfiguration` is depicted in Figure 4.15.

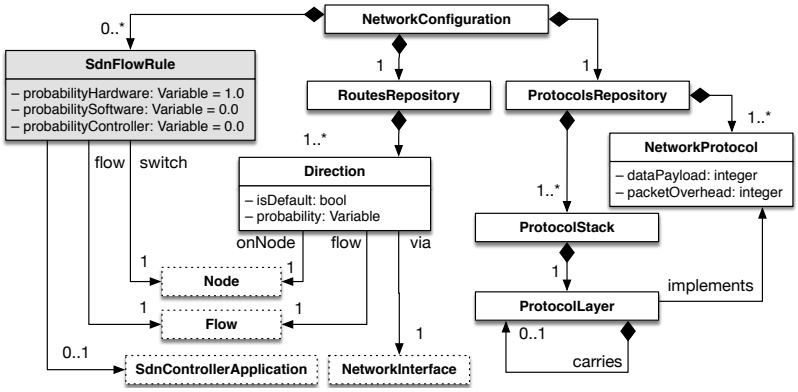


Figure 4.15: The DNI meta-model of network configuration with SDN entities included (in gray).

4.3 Flexibility of Modeling

DNI meta-model offers high degree of modeling freedom. Many performance-related parameters can be left unspecified without violating the validity of a model. I present the flexibility of instantiating of the DNI meta-model in Section 4.3.1. Despite the high level of abstraction of DNI, still many parameters are required to build a DNI model. To reduce the amount of the required input data, I propose a smaller version of the DNI meta-model called *miniDNI*. I present the *miniDNI* meta-model in Section 4.3.2.

4.3.1 Flexibility in Building DNI Models

A valid instance of the DNI meta-model can be build in many ways. Providing a valid model guarantees that the model transformations (described in Chapter 5) will be able to analyze and transform it. The set of model transformations that support DNI is unbound, so it cannot be guaranteed that each valid DNI model will be transformed or solved if a given transformation does not support a part of the DNI model. This aspect is discussed in more detail in Chapter 5.

The DNI meta-model offers a lot of flexibility already in the modeling phase. Several parts of the meta-model give freedom to the modeler to pick the proper level of details. The options from which the modeler can choose have been depicted as a tree in Figure 4.16.

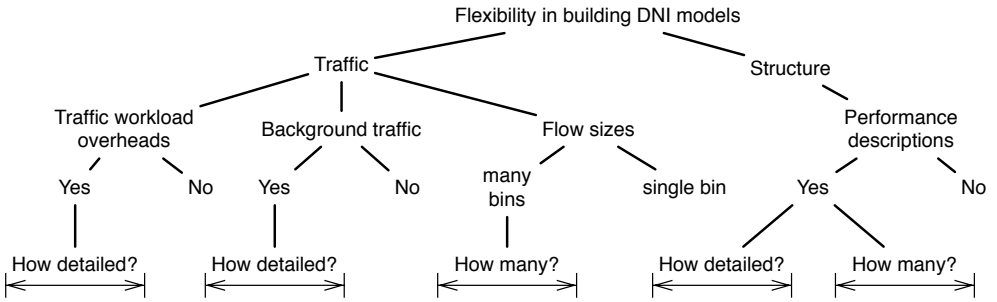


Figure 4.16: Flexibility in building DNI models.

The modeler may choose the level of detail for the DNI model based on the answers to the following questions. (1) How detailed the traffic workload overheads may be modeled (usage profile versus usage profile with overheads)? (2) Shall model background traffic be modeled? If yes, then how detailed? (3) How to group the sizes of flow messages? (4) How many performance descriptions to include in the DNI network structure elements? (5) How detailed the structure performance descriptions should be (e.g., vendor-provided data versus measurement)? In the following, I present the places in DNI where the user may decide about the granularity of modeling.

Skipping Performance Descriptions

Assigning performance description for DNI entities is optional. Performance descriptions can be assigned to the following DNI entities: `End Node`, `Intermediate Node`, `SDN Node`, `NetworkInterface`, `Link`, and `SdnControllerApplication`. I assume, that DNI models can be build step-wise and the elements without performance annotation should be treated as structural information. For this reason, an element without a performance description is assumed to incur 0 delay and deliver infinite throughput. Combining elements with and without the performance description creates a space for the modeler to vary the level of detail of the model.

Accuracy of Modeling Workloads

A workload in DNI is understood as the amount of data that is created in a given moment of time in a given `CommunicatingApplication`. `Workload` (including its `Actions`) is the only entity in DNI that represents a time span with contrast to the other entities, which represent a snapshot of a system's state in a single moment of time. The `Workload` can be specified flexibly, that is, including or abstracting the following factors: the overheads of the application and the operating system, background traffic, the exact flow sizes, exact timing of transmission actions.

`Workload` is an important part of DNI regarding the sensitivity (as demonstrated in Chapter 7). It means, that excluding selected elements of the workload in the model may decrease the representativeness of the model and thus degenerate the performance prediction accuracy delivered by the solvers.

In an ideal case, the modeler defines Workload from the *usage profile* context. For example: "*Application A generates one message of size 2.47MB every second and transmits it to application B.*". In an ideal case, such a natural language description may be directly translated into DNI workload as a `Loop` that contains two actions: `Transmit` a flow of size 2.74MB and `Wait` one second. In reality, prior to sending, the messages will be processed by the application, the operating system of the node, and the network interface. This may reshape the traffic workload of the application and result in imprecise representation of the traffic, which in result may lead to lower performance prediction accuracy. Analyzing the overheads incurred by the application and the operating system (and possibly virtualization layers) lies not in the scope of DNI and is normally handled by DML. Integration of DNI and DML will be discussed in Section 4.4.

Preferably, the DNI traffic workload should be captured at the respective network interface. In this way the DNI's traffic workload represents the real traffic profile with the highest modeling accuracy which implies also higher performance prediction accuracy in the end. The modeler decides which elements of the traffic profile will be included in the DNI's traffic workload model and which will be abstracted.

Another example that concerns the flexibility of modeling of the traffic workload is the background traffic. It is difficult to model the background traffic if it cannot be precisely measured. For scenarios where the background traffic cannot

be abstracted, its traffic profile should be automatically extracted. I describe automated extraction of the DNI traffic workloads in Section 6.2.

Binning of Flow Message Size

The DNI traffic workload is composed of `Transmit` and `Wait` actions. Each transmit action refers to a `FlowTraffic`, which specifies the size of the transferred message. Each DNI `Flow` contains a set of discrete values of its message sizes. The granularity of the message sizes set can be freely defined by the modeler. One modeler may measure the message sizes and bin them into predefined set of bins, for example, 200KB, 210KB, 220KB, whereas another may round up the sizes, treat the messages as identical approximating their sizes to 210KB.

4.3.2 miniDNI Meta-Model

In the *miniDNI* meta-model (depicted in Fig. 4.17), I abstract selected parts of DNI. Regarding the DNI's `NetworkStructure`, I removed the `NetworkInterface` entity, which is merged now with the `Link`. The performance of a `Link` in miniDNI is described using *throughput* and *delay*. The performance descriptions of various `Node` aspects are merged into a single object `NodePerformance`. The `NodePerformance` includes *softwareLayersDelay* and four throughput specifications, each for the following forwarding modes: native mode (non-SDN), software SDN, hardware SDN, and via SDN controller. The performance of an SDN controller is now represented using a single parameter that abstracts DNI's `SdnControllerApplications` and the path between a node and the controller.

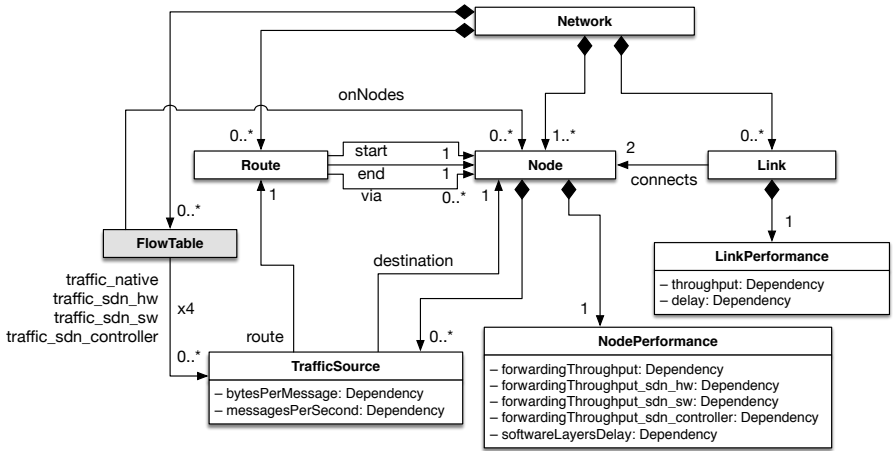


Figure 4.17: miniDNI meta-model with SDN entities included (in gray).

The `CommunicatingApplications` are abstracted and represented as `TrafficSources`. However, miniDNI's `TrafficSource` is less detailed than the DNI's

`TrafficSource`. A `TrafficSource` in miniDNI generates workload that uses a concept of "messages" without enforcing specific interpretation of the message. As a possible interpretation of a message, I understand, for example: a picture, a database transaction. A workload generated by a `TrafficSource` is described by using two parameters: *bytesPerMessage* and *messagesPerSecond*.

The configuration of a network is modeled using two entities: `Route` and `FlowTable`. A `Route` is an ordered list of `Nodes` with distinguished *start* and *end* (which can be derived automatically for the convenience of the modeler). Each `TrafficSource` follows a `Route` that it refers to. A `FlowTable` is used only in SDN setups and represents the mapping of a `TrafficSource`, a `Node`, to one of four processing modes: native (non-SDN), SDN with hardware flow table, SDN with software flow table, and SDN over the controller. miniDNI does not support specifying the probabilities that describe the chance of being forwarded using a given mode. The traffic from a `TrafficSource` is forwarded deterministically using the predefined forwarding mode that is specified in the `FlowTable`.

Table 4.1: Comparison of the differences in modeling granularity between the DNI and the miniDNI meta-models.

Modeling detail	DNI	miniDNI
Traffic patterns	yes	no, flat traffic
Packet-level traffic	yes	no, coarse messages
Network Protocols	yes	no
SDN support	yes, probabilistic	yes, deterministic

In Table 4.1, I present the main differences in modeling granularity between the *DNI* and the miniDNI meta-models. In contrast to *DNI*, miniDNI models the traffic at the level of messages. Using messages instead of packets flattens the traffic profile by aggregating the traffic in one-second intervals and thus destroys any bursts or peaks in the traffic profile. The `NetworkProtocols` are not modeled in miniDNI, thus their overheads need to be aggregated into other performance-relevant parameters. miniDNI supports SDN networks, however only using deterministic mapping of flows to predefined flow tables in the nodes. I present miniDNI model-transformations in Chapter 5 and evaluate the performance prediction accuracy of the generated predictive models in Chapter 7.

4.4 Integration of the DNI Abstractions with Descartes Modeling Language

DNI extends DML in a natural way. Both meta-models represent complementary parts of a data center: DNI models the networks, DML models the computing and software parts. DNI and DML can be used separately as well as together to jointly represent a data center.

In this section, I present the relation between DNI and DML. First, I present an overview of the DML in Section 4.4.1, next I analyze the overlapping parts and

propose the mapping of the data center structure in Section 4.4.2, the applications in Section 4.4.3, and the network workload in Section 4.4.4. In Section 4.4.5, I summarize the mappings between the models and propose a *Network Deployment* meta-model that binds DML and DNI meta-models. Next, in Section 4.4.6, I demonstrate the integration on an example.

4.4.1 Overview of Descartes Modeling Language

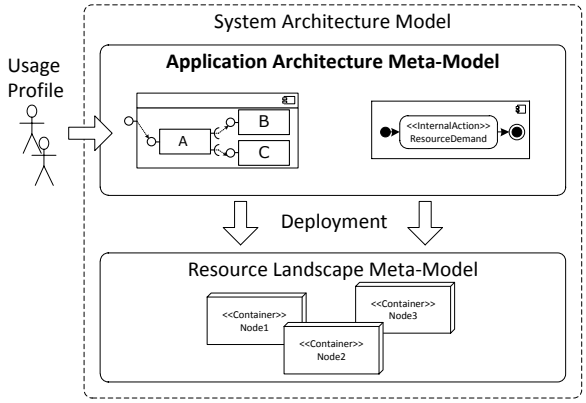


Figure 4.18: Overview of DML's structure. Excerpted from [Bro14b].

Brosig [Bro14b] describes DML as follows. "The system architecture meta-model consists of the application architecture meta-model and the resource landscape meta-model. The resource landscape meta-model allows modeling the physical and logical resources (e.g., virtualization and middleware layers) provided by modern dynamic data centers [HBK12]. The application architecture meta-model allows modeling the performance-relevant service behavior of the applications executed in the resource landscape [BHK14, BHK12]. These two meta-models are connected with the deployment meta-model, which can be used to describe how software components are deployed. The usage profile meta-model can be used to describe how users access the hosted applications." An overview of DML is presented in Figure 4.18. In the following sections, I present how I extend the *Deployment* part of DML to allow integration with DNI.

4.4.2 Data Center Structure

DML and DNI are both located within the data center context. Both meta-models represent selected aspects of physical and virtual elements of a data center structure. DML uses the *Resource Landscape* to represent computing nodes, storage nodes, and virtual containers. DNI uses the *Network Structure* part to describe physical and virtual nodes (including network nodes) that are connected with network links.

DML's *Resource Landscape* meta-model is presented in Figure 4.19 and 4.20. It describes a `DistributedDataCenter` that consist of `DataCenters`. A `DataCenter` contains `Compute-` and `StorageNodes` that can be organized hierarchically in a `CompositeInfrastructure`. Some DNI and DML entities overlap with each other.

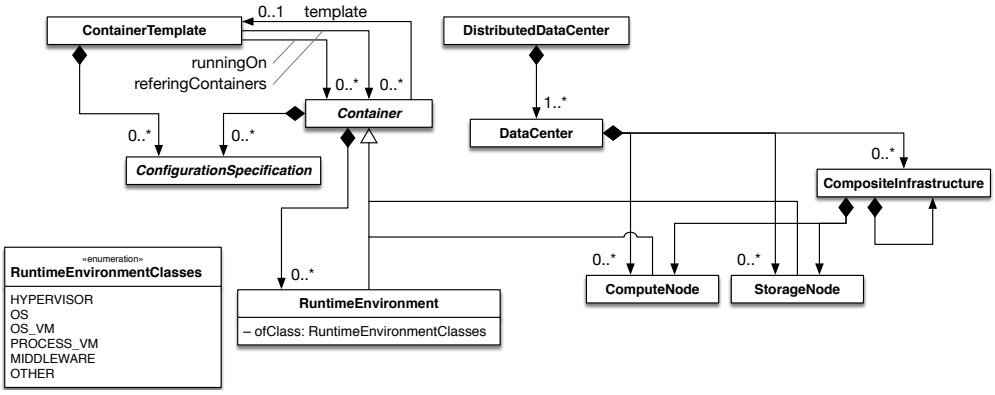


Figure 4.19: DML's Resource Landscape meta-model. Updated and redrawn based on [Hub14].

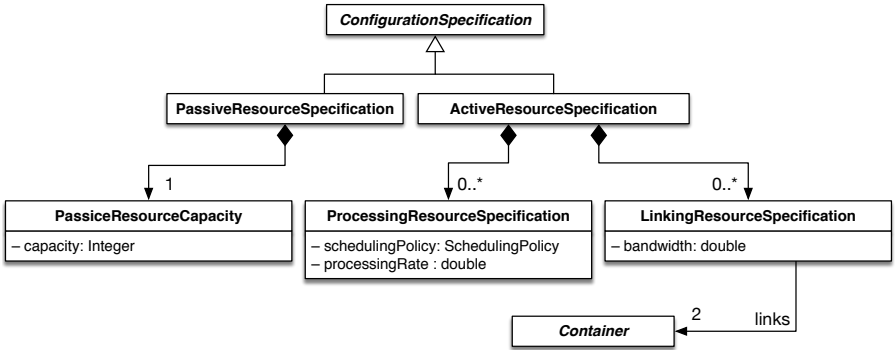


Figure 4.20: DML's Resource Landscape meta-model: ConfigurationSpecification. Updated and redrawn based on [Hub14].

The overlapping entities and the degree to which they overlap is presented in Table 4.2.

4.4.3 Data Center Applications

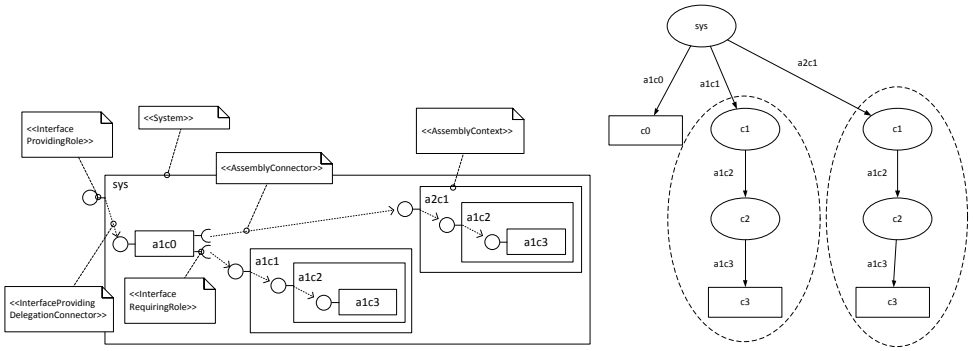
The business applications are represents in DML using `AssemblyContexts`. An `AssemblyContext` binds DML's software components together to form a `Composite-Component` or a `System`. The software component instances in DML are represented

Table 4.2: Overlapping entities in DNI and DML—data center structure.

Real entity	Server, Switch, Storage, virtual machine (VM)
DNI	`Node`
DML	`ComputeNode`, `StorageNode`, `RuntimeEnvironment`
A server, switch, and VM can be represented in DNI using the `Node` entity. A server is represented by an `EndNode` (`Node` with `IPosition=End`), a switch as `IntermediateNode`, and a VM as `Node` that is hosted on another `Node`. It is impossible to distinguish an instance of DML's `ComputeNode` form a `StorageNode` in DNI—both will can be represented with `End Node`. Only specific DML's `RuntimeEnvironments` can be mapped to DNI's `Nodes`. DNI requires a `Node` to be able to have network interfaces, so only a `RuntimeEnvironment` *ofClass=OS_VM* can represent a `Node` that is hosted on another `Node` (a virtual machine). `RuntimeEnvironments` of other classes are ignored by DNI.	

Real entity	Network Connection, Network Cable/Medium
DNI	`Link`, `NetworkInterface`, `Node`
DML	`LinkingResourceSpecification`
DML represents the network as a black-box using the `LinkingResourceSpecification` that binds two `Containers`. The network path between two containers can include a single or multiple network links. This can be expressed as a chain of the following DNI entities `NetworkInterface-Link-NetworkInterface-Node`. So a `LinkingResourceSpecification` abstracts the network topology and represents a `Link` or the `Network structure` depending on the case.	

as *chains* of assembly contexts, because a single component can be deployed multiple times. In Figure 4.21, I present an example from [Bro14b] to illustrate this. In the example, there are three component instances—as seen in Figure 4.21b–composed using two components: $a1c0$ and $a1c3$ (Fig. 4.21a).



(a) Example system instance.

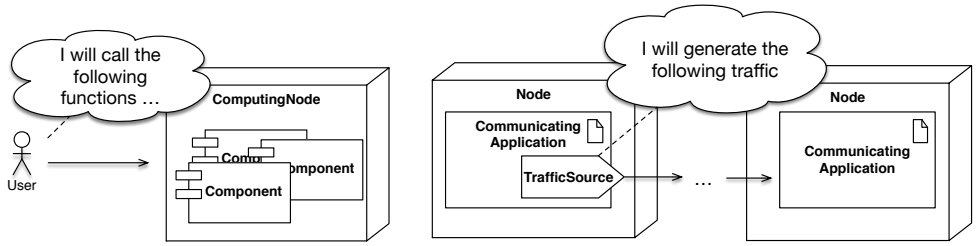
(b) Example system instance as composition tree.

Figure 4.21: Assembly Context in DML. Excerpted from [Bro14b].

In DNI, the business applications are represented using the `CommunicatingApplication` entity. A `CommunicatingApplication` represents a chain of DML's `AssemblyContexts`, that is, a component instance. Component instances that do not communicate over network are modeled using `CommunicatingApplication` but have no `TrafficSource` in DNI.

4.4.4 Traffic Workload

Both DML and DNI use the term "workload" but the meaning is different. DML defines workload as the unit of work representing request to the modeled system. The workload in DML is represented by *UsageProfile* meta-model. A workload in DNI represents the amount of traffic that needs to be transmitted over network. So DNI's workload is indirectly caused by DML's workload. A graphical comparison of the workloads is depicted in Figure 4.22. Unfortunately, the traffic `Workload` in



(a) DML. (b) DNI.

Figure 4.22: Different meaning of workloads in DML and DNI.

DNI cannot be directly mapped to selected DML entities (and vice versa) as the discrepancy between the representations is too big. Fortunately, the DNI's traffic `Workloads` can be *extracted* from a DML instance.

Extracting Traffic Volume

Network traffic emerges when a software component calls an interface (a software interface) of another software component under the assumption that the components are deployed on separate nodes. The signature of the called software component contains call parameters and return parameters. The size of the call parameters and the return parameters defines the traffic flow sizes in both communication directions respectively.

An example is presented in Figure 4.23. The *WebShop* component instance deployed on the *ApplicationServer* requires the interface of the *SQLDB* component. By calling the interface, the *WebShop* transmits data over the network (abstracted in Fig. 4.23). After processing the call, the *SQLDB* component returns data to the caller and thus traffic in opposite direction emerge.

A traffic `Workload` of DNI's `TrafficSource` can be extracted if the following factors are known: (1) size of the data parameters in the call, (2) size of the returned data, and (3) moments of the call and the returning of data. This information is partially included in DML. Selected time aspects may need to be simulated (solved) to obtain the exact time stamps of the events. I explain how the three required parameters are represented in DML in the following.

The software components in DML are specified using three levels of granularity. The `ServiceBehaviorAbstraction` is presented in Figure 4.24. Each `ServiceBehav-`

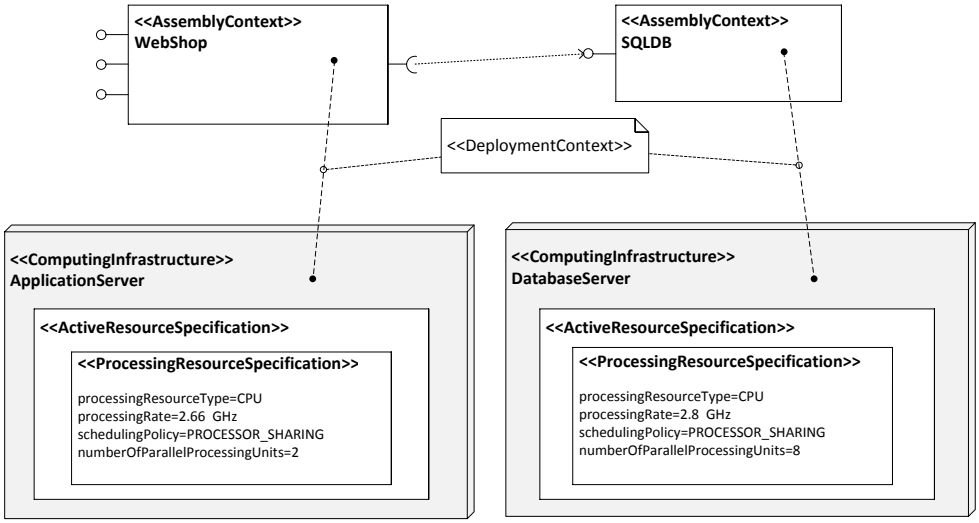


Figure 4.23: DML Example. Excerpted from [Bro14b].

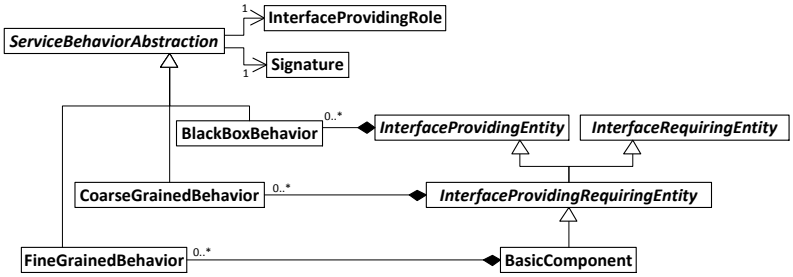


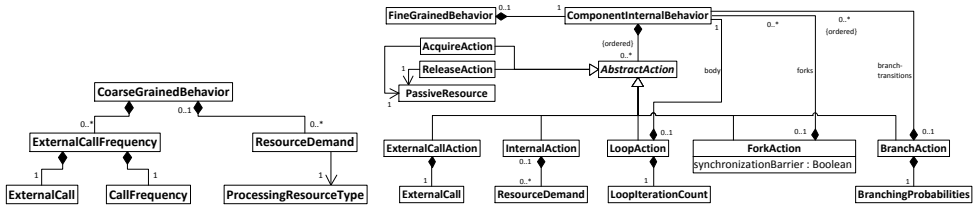
Figure 4.24: DML's FineGrainedBehavior. Excerpted from [Bro14b].

`iorAbstraction` may be described using `BlackBox-`, `CoarseGrained-` (Fig. 4.25a), and `FineGrainedBehavior` (Fig. 4.25b).

The `BlackBoxBehavior` contains only the specification of *response time* of a service. Based only on a `BlackBoxBehavior`, the information about outgoing traffic workload cannot be extracted because the `BlackBoxBehavior` abstracts external calls. I can extract the incoming calls as long as they originate from a service described with `Coarse-` or `FineGrainedBehavior`.

In contrast to the `BlackBoxBehavior`, the `CoarseGrainedBehavior` and the `FineGrainedBehavior` represents the external calls (see Fig. 4.25), so the originating traffic workload can be extracted from services described with these entities.

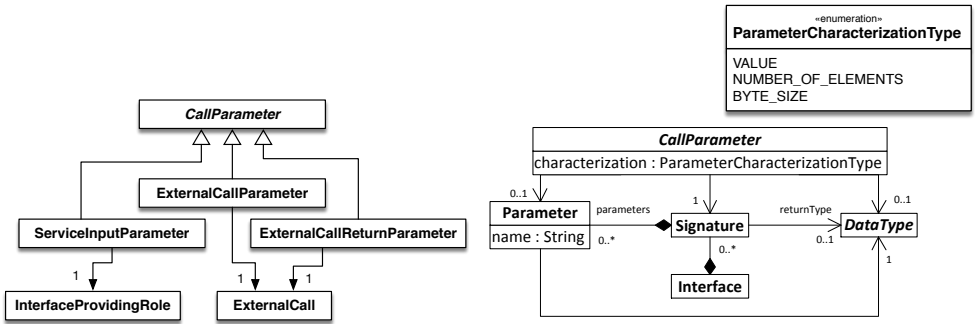
Each `ExternalCall` is parametrized with `CallParameter` and `ParameterCharacterizationType` that contains information about `BYTE_SIZE` (see Figs. 4.26a and 4.26b). Calculating the total size of each `ExternalCall` and combing it with the respec-



(a) CoarseGrainedBehavior. (b) FineGrainedBehavior.

Figure 4.25: Service behaviors in DML. Excerpted from [Bro14b].

tive behavior descriptions (`CallFrequency` for `CoarseGrainedBehavior` and the `ComponentInternalBehavior` for `FineGrainedBehavior`) allows to specify the volume of traffic originating and ending in a given software component instance. The calculated traffic volumes shall be used to calculate the DNI's *dataSize* of a `GenericFlowTraffic`.



(a) CallParameter hierarchy. (b) CallParameter.

Figure 4.26: `CallParameter` in DML. Updated and redrawn based on [Bro14b].

Extracting Traffic Profile

The traffic profile in DNI includes time series containing the information about the moments in which traffic is generated by a service. The service generates traffic by being called or by calling other services. DML uses the *usage profile* model to represent the user interactions with a system. The DML's *usage profile* model is based on Palladio Component Model (PCM) [BKR09]. I present the *usage profile* meta-model in Figure 4.27.

The DML's *usage profile* is similar to DNI's `Workload` model. Both model contain control-flow entities (e.g., loop, branch), delay entity, and a workload generating entity. In DML, the workload generating entity is represented with `SystemCallUserAction`. The `SystemCallUserAction` generates load by demanding the

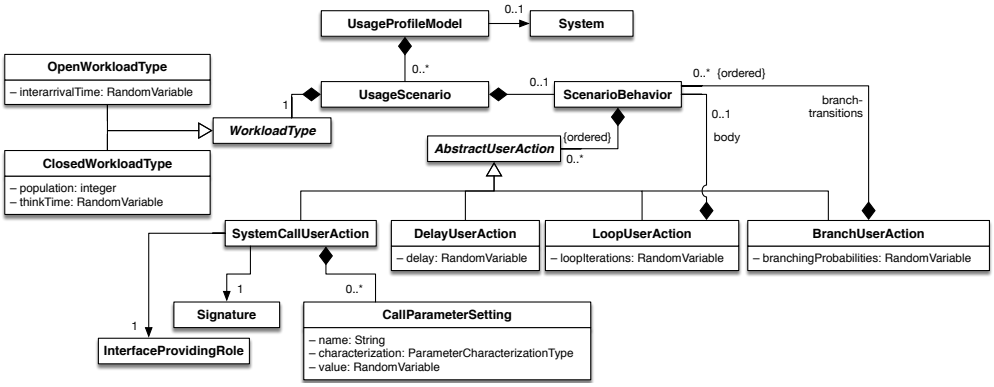


Figure 4.27: Usage Profile Model in DML. Updated and redrawn based on [Bro14b].

service from a given component instance. In DNI, an equivalent of the `SystemCallUserAction` is the `TransmitAction`. A `TransmitAction` generates load in the form of transmitting a given volume of data over the network. As described in Section 4.4, the `SystemCallUserActions` are the causes of `TransmitActions` and the latter can be obtained based on the former.

DML allows to represent a usage profile in the form of open and closed workload. In contrast to this, DNI supports only open workloads, yet according to [SWHB06] closed workloads can be transformed into open ones.

To build DNI's traffic `Workload`, I propose the following procedure.

1. Convert DML's closed workload into an open workload for the required time frame (if needed).
2. Identify the chain of `SystemCallUserActions` and `DelayUserAction` contained in the *usage profile* model.
3. For each `SystemCallUserAction`
   a) Unfold the chain of `ExternalCalls` of the callee in a recurrent manner.
   b) Calculate the response time of the called services based on the `ServiceBehaviorAbstractions`, represent each response time as a DNI's `WaitAction`.
   c) Build a `TransmitAction` for each `ExternalCall` that originates and ends in different nodes.
   d) Use *ParameterCharacterizationType* `BYTE_SIZE` to calculate the size of the transmitted volume.

The procedure is discussed in more detail in Section 4.4.6, where I demonstrate it using an example.

4.4.5 Network Deployment Meta-Model

The mapping of DML and DNI entities is presented using the *Network Deployment* meta-model that extends the *Deployment* meta-model of DML. In Figure 4.28,

I present the *Network Deployment* meta-model. The original DML *Deployment* model (presented in gray in Fig. 4.28) contains `DeploymentContext` entity that binds `AssemblyContext` in DML *Application Architecture* model (in yellow) with a `Container` from DML *Resource Landscape*.

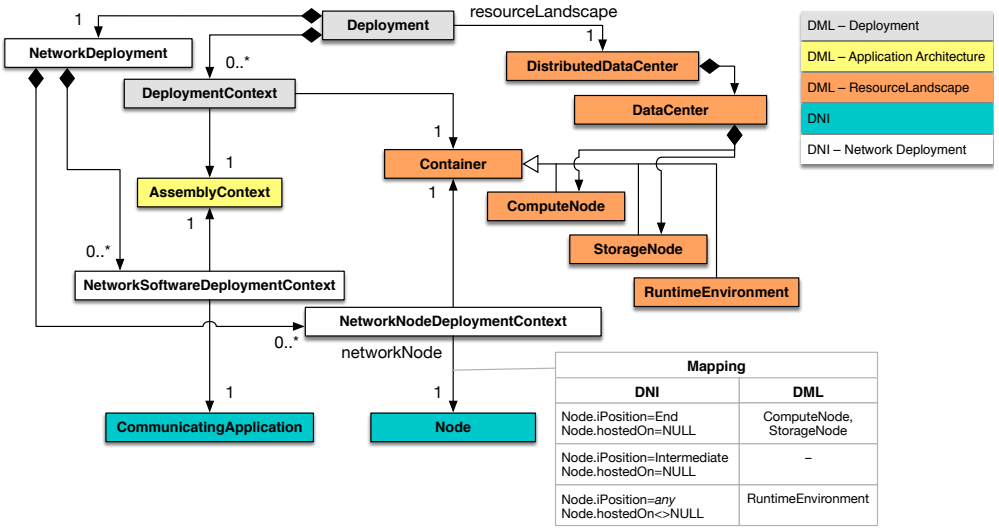


Figure 4.28: *NetworkDeployment* meta-model as extension of DML's *Deployment*. Respective parts are presented using distinguished colors.

The entities marked in white in Figure 4.28 bind DNI with the current DML deployment model. The `NetworkDeployment` contains two types of mappings: DNI `Node` to DML `Container`; and DNI `CommunicatingApplication` to an `AssemblyContext` in DML. The mapping between DML `Container` and DNI `Node` is annotated with additional context information regarding the types of nodes that can be mapped to each other. The relation between DML `Container` and DNI `Node` is discussed in Section 4.4.2. Neither the DNI `NetworkDeployment` nor the DML `Deployment` meta-models define mappings of workloads—they need to be extracted at the model level.

4.4.6 Example

I present the usage of the combined deployment models and the workload extraction process using a running example. As a basis, I use the DML example presented in Figure 4.23.

Mapping of Nodes and Applications

The mapping begins with identifying the nodes. The *ApplicationServer* and *DatabaseServer* are modeled using the `ComputeNode` entity (orange color in Fig. 4.29). This

maps onto DNI's `End Node`. It is unknown if the *ApplicationServer* and *DatabaseServer* are configured to forward traffic, so I do not add the `Intermediate` entity to the DNI nodes. The mapping is formalized with `NetworkNodeDeploymentContext` entities.

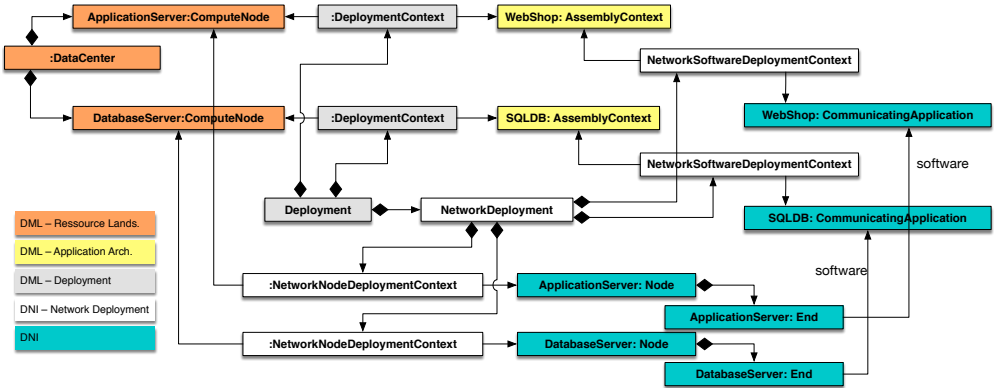


Figure 4.29: Example DML model (from Fig. 4.23) mapped onto DNI. Mapping presented using object diagram.

The *ApplicationServer* and *DatabaseServer* host business applications modeled in DML using `AssemblyContext` (yellow color in Fig. 4.29): *WebShop* and *SQLDB* respectively. For each DML object representing an `AssemblyContext`, I create a `CommunicatingApplication` in DNI and deploy it on the respective DNI equivalent of DML's node: *WebShop* on the *ApplicationServer* `End Node` and *SQLDB* on the *DatabaseServer* `End Node`. The mapping of `AssemblyContexts` to `CommunicatingApplications` leverages the `NetworkSoftwareDeploymentContext` entities. The DNI's `TrafficSource` entities are not created in this step as it is unsure if the applications generate any network traffic. I demonstrate the workload extraction in the following.

Extracting Traffic Flows

The example DML model from Figure 4.23 was extended by adding *Service Behavior Description*. The extended fragment of the example is presented in Figure 4.30. The *WebShop* component is described using `CoarseGrainedBehavior`, in which one external call is made. The *WebShop* calls the *SQLDB* component to fetch data from a database. The call is specified using the `ExternalCall-` and `ExternalCallReturnParameter`, in which *WebShop* sends 3500-byte request to *SQLDB* and the *SQLDB* responds with a data block of size 4 194 304 bytes (≈4MB).

The `CoarseGrainedBehavior` allows the modeler to extract the information about DNI `Flows`. The modeler knows that the information is exchanged in both directions (call + reply) and the size of the data is known. Based on these data, I build two DNI `Flows` as shown in Figure 4.31. To each newly built *Flow*, I attach

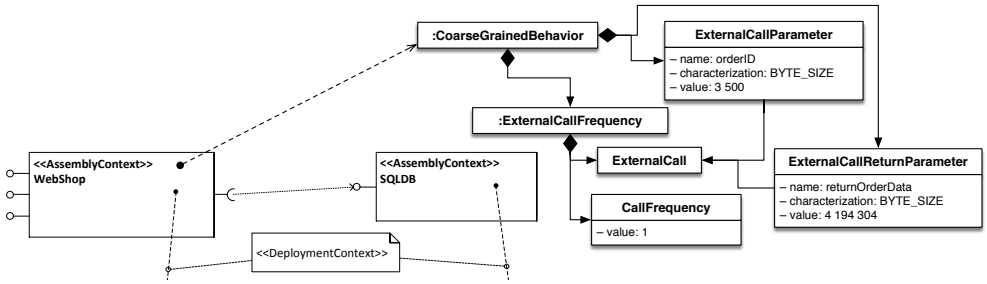


Figure 4.30: Example DML model (from Fig. 4.23) extended by defining *Service Behavior Description* using `CoarseGrainedBehavior`. The deployment part of the example is abstracted for brevity.

a `GenericFlowTraffic` that defines the size of a single message exchanged in the flow. The *SQLDB-WebShop* flow has 4MB, whereas the opposite *WebShop-SQLDB* 3 500 bytes. In this step, one can also add `TrafficSources` to the respective `CommunicatingApplications`.

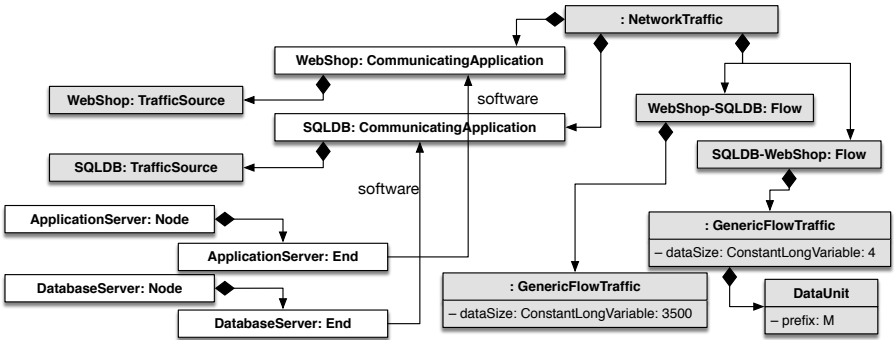


Figure 4.31: Extracting DNI Traffic Flows based on DML model. Object diagram. Gray entities are highlighted to show the new objects with respect to the previous step (Fig. 4.29).

Extracting Traffic Workload

In the last step of the traffic workload extraction, I add `Workload` objects to the `TrafficSources`. The workload profiles are extracted based on the DML's usage profile. I add a simple usage profile to the DML example presented in Figures 4.23 and 4.30. The extended DML example is presented in Figure 4.32.

In the example presented in Figure 4.32, I add a user that calls the *fetchData* functionality provided by the *WebStore* component. I represent the call using the dashed arrow as the respective part of DML model is abstracted from the Figure for brevity. The user calls the function with parameter *oderID* valued as 1 or

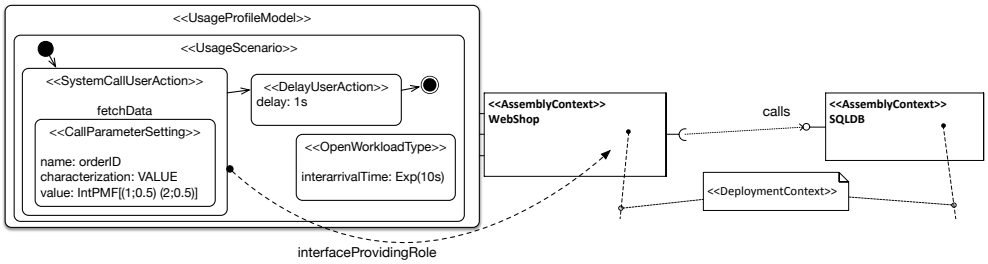


Figure 4.32: Example DML model (from Fig. 4.23 and 4.30) extended by defining *Usage Profile Model* (presents using graphical notation based on [BKR09]). The deployment and the Service Behavior Description parts of the example are abstracted for brevity.

2, each with probability 0.5. Next, the user waits exactly one second and its actions are over. The complete process repeats in an open-workload-manner with interarrival time expressed as exponential distribution with mean of 10 seconds.

The usage profile presented in Figure 4.32 can be transformed according to the procedure presented in Algorithm 1.

Algorithm 1 A sketch of procedure for extracting DNI traffic workload from DML usage profile.

```
1: function ProcessSystemCallUserAction(action)
2:     assemblyContext←action.interfaceProvidingRole
3:     TWA ← new Ordered Set                          ▷ TWA: Traffic Workload Actions
4:     return ProcessAssemblyContext(assemblyContext, TWA)

5: function ProcessAssemblyContext(ac, TWA)           ▷ ac is DML.AssemblyContext
6:     if ac.serviceBehaviorDescription.hasExternalCalls=true then
7:         callee ← ac.calls
8:         ta1 ← new DNI.TransmitAction
9:         ta1.transmits ← Flows.filter(f | f.source=ac and f.destination.includes(callee)).generatedTraffic
10:        ta2 ← new DNI.TransmitAction
11:        ta2.transmits ← Flows.filter(f | f.source=callee and f.destination.includes(ac)).generatedTraffic
12:        TWA.add ← ta1
13:        TWA.add ← ProcessSystemCallUserAction(callee, WTA)
14:        TWA.add ← ta2
15:    else
16:        waitAction ←new DNI.WaitAction
17:        waitAction.waitTime ← ac.getProcessingTime()
18:        TWA.add ← waitAction
19:    return TWA
```

In Algorithm 1, I make the following assumptions: (1) it is known if an *AssemblyContext* has external calls to components deployed on another node, (2) the *AssemblyContext* has single `ExternalCall` that can be identified with *ac.calls* (line 7 in Algorithm 1), (3) components with `ExternalCalls` have negligible low processing

time, and (4) the execution time of each component call is known beforehand. Not all of the assumptions hold a priori (before solving the DML model), yet all are known a posteriori.

In Algorithm 1, each `SystemCallUserAction` points at a component represented by *AssemblyContext*. The *AssemblyContext* is then processed recurrently. If the *AssemblyContext* calls external components, then two *TransmitAction* objects are build, one for the request and second for the reply. The DNI actions are added in the order of calling the respective components in DML.

I do not provide procedures for processing other DML `AbstractUserActions` as the mapping is trivial. The `DelayUserAction` may be transformed directly, whereas the container actions (e.g., `Loop-`, `BranchUserAction`) have the same semantic as in DNI, so the transformation procedure is intuitive. In Figure 4.33, I present the example after extracting the DNI traffic workloads based on DML usage profile. I mark added entities in gray for brevity.

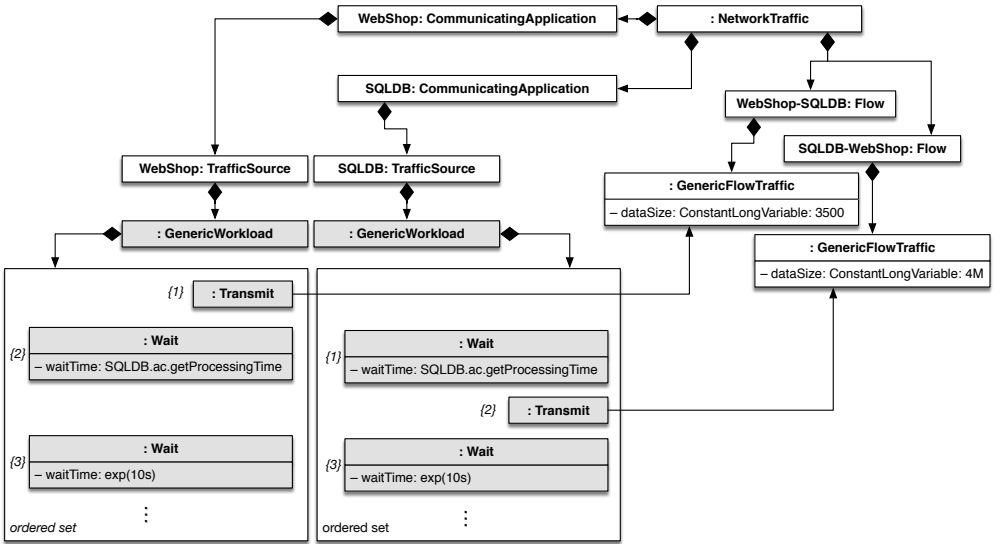
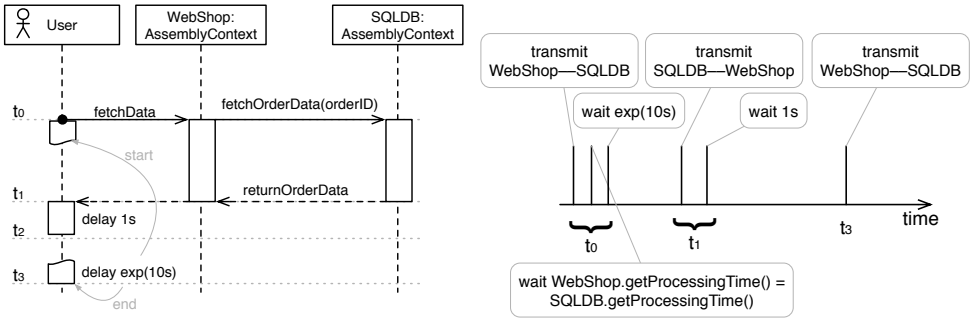


Figure 4.33: Extracting DNI Traffic Workload based on DML model. Object diagram. Gray entities are highlighted to show the new objects with respect to the previous step (Fig. 4.31).

In the last phase of the DNI traffic workload extraction, a `GenericWorkload` object for each `TrafficSource` is built. Next, an ordered set of actions is attached according to the execution order in the DML usage profile. In the example, the *WebShop* calls the *SQLDB* component by transmitting the request to it. This is represented by the `Transmit` action executed as first (order denoted: *{1}*). Next, the *WebShop* `TrafficSource` idles for the time of execution of the *SQLDB* component. As the DML usage profile contains no further actions, the same sequence is repeated for the time frame of the measurement.

The *SQLDB TrafficSource* executes analogous scenario. First, it idles while it gets called and executed—for simplicity of this example, I assume that *SQLDB* is called immediately. Next, *SQLDB* generates a response based on the `ExternalCall-ReturnParameter` that war transformed into the *SQLDB-WebShop* `Flow`. Finally, *SQLDB* idles *exp(10)* seconds as defined in DML. The process repeats until the experiment is over.

The example is additionally depicted using a sequence diagram in Figure 4.34a with the respective time series representing DNI Traffic Workload actions in Figure 4.34b. At time t_0 the user sends a request to the *WebShop* component.



(a) DML call sequence diagram.

(b) DNI `Workload`.

Figure 4.34: Extraction of the DNI traffic workload model from a DML model.

Immediately an `ExternalCall` is issued and the *SQLDB* component is called. In this moment the network traffic is produced. The *SQLDB* component replies within time $t_1 - t_0$ and the result is immediately passed to the user. Next, the user waits for 1 second and her actions end. After a time specified as *exp(10s)* the process begins again with another user. The exponentially distributed inter arrival time defines the duration period $t_3 - t_0$. In the diagrams in Figure. 4.34, I assume that $t_2 - t_0 < t_3 - t_0$.

The user actions and component calls are mapped to the DNI traffic workload actions depicted in Figure 4.34b. In DNI traffic workload, I assume that a `Transmit` action happens immediately because the processing and transmission delays are applied in other places (e.g., a `NetworkInterface` entity or a *SoftwareLayersDelay* parameter in the `End Node`). Thus, the first transmit action happens at time t_0 (WebShop→SQLDB) and the second in t_1 (SQLDB→WebShop). Also in t_0 starts the delay action that takes as long as the DML call *fetchData* needs. Similarly, in t_0 start the 1s and the *exp(10s)* wait actions. In moment t_3 the process is repeated until the complete workload is extracted. This ends the extraction of the traffic workload.

4.5 Summary

In this chapter, I described network performance abstractions for performance prediction. The proposed performance abstraction cover classical, SDN-, and NFV-based network infrastructures, network configuration, and the traffic workloads. The introduced modeling formalism may be used jointly with the DML to cover wider scope of a data center than the DML or DNI alone.

In Section 4.1, I introduced the DNI meta-model focusing on the modeling of classical, non-virtualized network infrastructures. In Section 4.2, I extended the meta-model for modeling classical networks and presented new entities used for virtualized networks (based on SDN and NFV). Section 4.3 discussed the flexibility of building DNI model with various level of detail to balance between the fine and coarse granularity of the model. Next, in Section 4.4, I presented how to integrate DNI meta-model with the DML meta-model using the *Network Deployment* meta-model as an interface, so that the models can jointly represent a data center. The integration of DML and DNI solvers opens several challenges that are discussed as a part of future work in Section 8.2.1.

Chapter 5

Model Transformations and Solving

Descriptive models organize information about a domain. Instances of Descartes Network Infrastructure (DNI) meta-model have descriptive nature and do not provide any means of performance prediction. To leverage the descriptive domain information in the performance prediction process, a DNI model needs to be transformed into a predictive model that can be solved. In this chapter, I describe the transformations and solution techniques available for DNI models.

The performance of network infrastructure has important influence on the performance offered by a data center to the end user. Being able to analyze new network topologies, configuration settings, Software-Defined Networking (SDN) configuration, or applications deployment without interrupting the operation of a network is of high value to a network operator. The network operator can use DNI to simulate different settings of the network before applying the changes to the productive system. The approach based on DNI offers coarse to medium detailed performance predictions focused on network throughput. The offered performance predictions vary in accuracy an the solving time depending on the simulated network and the selected solver.

The approach proposed in this thesis aims at run-time performance analysis for capacity management purposes. It aims at analyzing and adapting the network configuration and resource allocation dynamically. To dynamic adaptations, I account any reconfigurations that are justified with anticipated changes in the environment that lead to changes in the offered quality of service—in extreme cases to service level agreement (SLA) violations. To examples of changes in the data center that may trigger adaptations I account: a network traffic load spike, traffic flows previously unknowns to an SDN controller, changes in traffic characteristics (e.g., less “elephant” and more “mice” flows).

A typical scenario for run-time performance analysis and dynamic adaptation may be presented as follows:

1. Anticipation of an event that may influence the performance of the system,
2. Analysis of the impact of that event on the system,
3. Building a set of candidate corrective reconfigurations to minimize the impact of the event on the performance,
4. Analysis of the implications of applying each corrective action,
5. Selection of an optimal corrective action and applying it to the system.

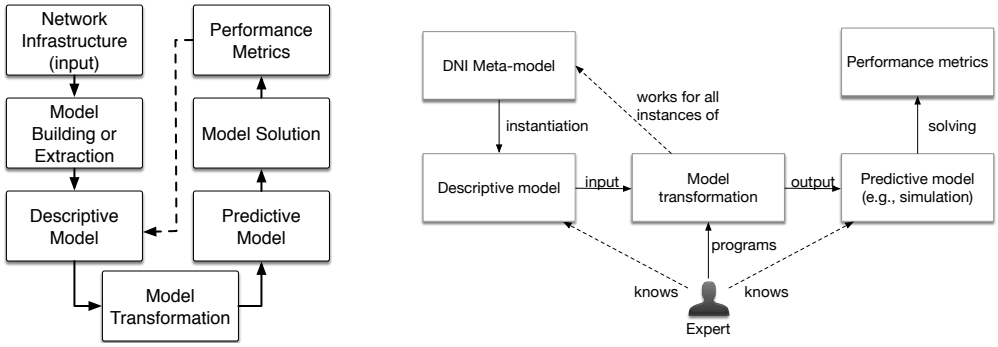
The proposed approach contributes directly to the steps two and four of the presented scenario. Anticipating future events (step 1) can be conducted using forecasting techniques [HHKA14]. Moreover, a data center operator may analyze the impact of a planned event (e.g., replacement of a device) on the system's performance. In step two, the modeler builds a DNI model including the anticipated or planned change and uses the proposed approach to evaluate the influence of the change on the performance. If the impact of the change is low, no action is required, however, if the event causes significant changes in the offered capacity or performance, a corrective action needs to be applied to prevent or minimize the negative outcomes. In step three, an expert proposes a set of possible corrective actions, for example, reconfigurations, new resource allocation, or scaling the system. Each proposed corrective action needs to be evaluated to select the optimal one. In step four, the proposed approach allows to build a DNI model for each candidate corrective action and evaluate its impact on the system. Finally, based on the performance predictions, an optimal corrective action is selected and applied to the system *before* the anticipated or planned event occurs.

Data center network operators may be challenged with scenarios other than the typical scenario described in the previous paragraph. The network operators may face the following questions for which the answers can be found using the proposed approach: (1) How a new network configuration influences the throughput offered to an application? (2) Which SDN rules should be installed to not violate SLA? (3) Which switches should be replaced to increase capacity of the network? (4) How to redeploy virtual machines (VMs) onto servers to avoid bottlenecks? (5) To which degree the SDN-based load-balancing may increase the capacity of the system? (6) Which network links are overloaded and should be load-balanced?

In this chapter, I describe how to conduct run-time performance analysis of data center networks using the modeling formalism introduced in Chapter 4. The approach to performance prediction I propose is based on model transformations [SK03].

Kleppe et al. [KWB03] defines a model transformation as follows. "A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

The approach is presented in Figure 5.1a. First, a descriptive model is extracted or built manually based on the existing network infrastructure. Next, a model transformation is executed to obtain a predictive model. The predictive model is solved using a compatible solver and performance predictions—in form of performance metric values—are delivered as an output. The performance predictions may be used for recalibrating the descriptive model and the process may be repeated until the required accuracy is reached.



(a) Prediction process.

(b) Building a transformation.

Figure 5.1: Model transformations in the process of performance prediction.

Model transformations are developed for a pair of languages defined by their meta-models. Correctly developed transformation can consume any model (an instance of a meta-model) at the input model and produce an equivalent representation as the output model. In the model-based approach proposed in this thesis, the output model that a transformation produces is a descriptive model that can be directly solved by a proper solver. The output model of a transformation must be also described using a meta-model in order to build and execute the transformation. The schema presented in Figure 5.1b depicts the general process of building a transformation (the meta-model of the predictive model is abstracted in the figure).

The model transformations are often judged based on their *correctness* and *completeness*. The terms *correctness* and *completeness* can be defined differently in various communities, so I provide the definition and interpretation used in this work.

A model transformation is *correct* if for any valid input model, the transformation builds a valid output model. A model is valid if it complies to its meta-model. A model transformation is *complete* if it can transform all entities of the input meta-model. Other properties of model transformations are defined in [LAD+14, CH06, MVG06].

It is impossible to *prove* for a transformation that the semantics of the input model and the output model is identical after the transformation. I intentionally propose model transformations that produce descriptive models with different level of detail (i.e., slightly different semantics) to highlight the flexibility in network performance prediction. The transformations contributed in this thesis in fact approximate the input DNI model with the output models. Lucio et al. [LAD+14] define approximation transformations as an approximation of model m_1 using m_2 where m_1 is equivalent to m_2 up to a certain error margin.

For selected cases (e.g., stochastic simulation models), one could show that input and output models are semantically equivalent by comparing the underlying

Markov chains. This applies, however, only to predictive stochastic models and to non-approximate model transformations (i.e., transformations that do not abstract any data between the input and output models).

An overview of the DNI model transformations is presented in Figure 5.2. DNI and miniDNI models are presented using rounded rectangles. The models can be processed by model-to-model transformations denoted using gray rounded rectangles. A model-to-model transformation may produce a descriptive or a predictive model on its output. Predictive models (i.e., such models that can be directly solved with a solver) are presented using a "document" shape. The predictive models can be directly solved using solvers denoted using parallelograms. Models and model transformations that support the SDN annotations of DNI are marked with a star.

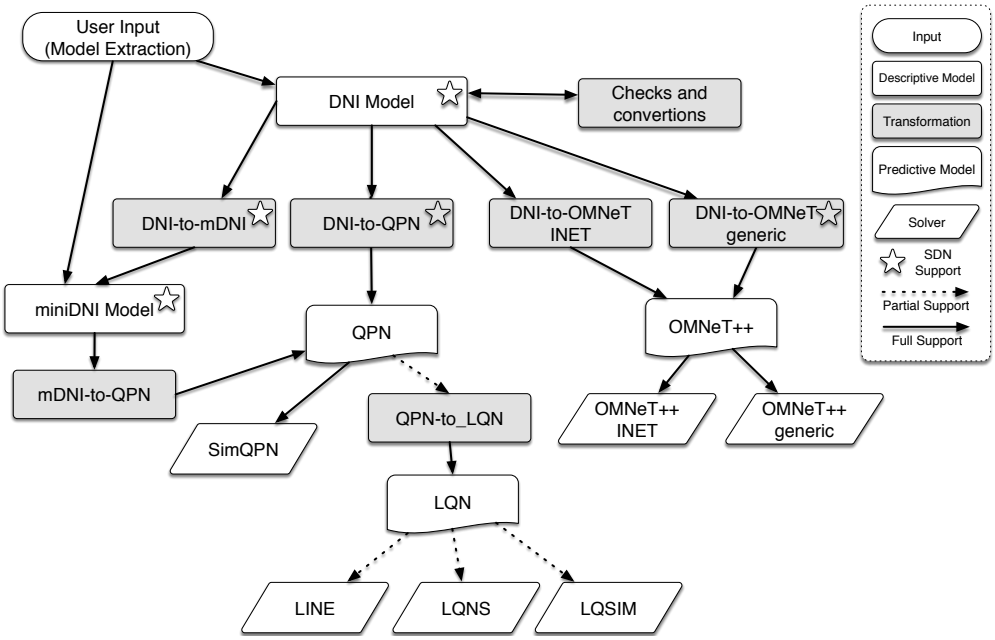


Figure 5.2: Overview of models, transformations, and solvers.

In Table 5.1, I present currently available transformations and their support for classical and SDN-based network scenarios. Currently available set of transformations allows to automatically generate up to ten various performance prediction methods:

1. DNI→ DNI-to-QPN→ solver *SimQPN*,
2. DNI→ DNI-to-QPN→ QPN-to-LQN→ solver LINE,
3. DNI→ DNI-to-QPN→ QPN-to-LQN→ solver LQNS,
4. DNI→ DNI-to-QPN→ QPN-to-LQN→ solver LQSIM,
5. DNI→ DNI-to-mDNI→ mDNI-to-QPN→ solver *SimQPN*,

Table 5.1: Matrix of DNI support of transformations and solvers.

Transformation	Solver	DNI	DNI+SDN	Published and presented in
DNI-to-QPN	*SimQPN*	yes	yes	[RK14a, RK14b, RKTG15, RSK16] Section 5.2
DNI-to-miniDNI	—	yes	yes	[RKTG15] Section 5.3.1
miniDNI-to-QPN	*SimQPN*	yes	no	[RKTG15] Section 5.3.2
DNI-to-*OMNeT++INET*	*OMNeT++INET*	yes	no	[RKZ13, RZK13] Section 5.4.1
DNI-to-*OMNeT++generic*	*OMNeT++generic*	yes	yes	unpublished yet Section 5.4.2
QPN-to-LQN	LINE, LQNS, LQSIM	partially supported	partially supported	[MRSK16] Section 5.5

6. DNI→ DNI-to-mDNI→ mDNI-to-QPN→ QPN-to-LQN→ solver LINE,
7. DNI→ DNI-to-mDNI→ mDNI-to-QPN→ QPN-to-LQN→ solver LQNS,
8. DNI→ DNI-to-mDNI→ mDNI-to-QPN→ QPN-to-LQN→ solver LQSIM,
9. DNI→ DNI-to-*OMNeT++INET* → solver *OMNeT++INET*,
10. DNI→ DNI-to-*OMNeT++generic* → solver *OMNeT++generic*.

The proposed set of transformations offers up to 10 solvers for a single DNI model. Unfortunately not all solvers support every instance of the DNI meta-model, so the number is lower in the practice. I elaborate more on transformation and solver feasibility in Section 5.6.2. In general, the set of the DNI transformations is not limited and may be extended as a part of the future work.

The rest of this chapter is organized as follows. In Section 5.1, I describe the transformation preparation steps including: DNI validity checking, in-place transformations, and transformation parametrization that enables specific transformation behavior. Next, in Section 5.2, I present the *DNI-to-QPN* transformation that transforms DNI models into Queueing Petri Net (QPN) models. I describe the transformation including its rules, features, examples, and limitations. Similarly, I present the *DNI-to-mDNI* and the *mDNI-to-QPN* transformations in Section 5.3. In Section 5.4, I describe two *DNI-to*-OMNeT++ transformations, whereas in Section 5.5, I present the *QPN-to-LQN* transformation that aims at providing analytical solvers based on Layered Queueing Network (LQN) for DNI. In Section 5.6, I compare the transformations and discuss the semantic gaps between the predictive models obtained in the transformations. Finally, in Section 5.7, I conclude this chapter.

5.1 Model Parametrization and Validity Checking

Each DNI model must be *valid* before it is passed to a model transformation to produce a predictive equivalent. In Section 5.1.1, I explain the *validity* and the necessary steps to ensure that a DNI model is valid.

A valid DNI model can be transformed using a model transformation, however the transformations can be additionally parametrized to produce slightly different model for each setting. I discuss the parametrization of model transformations in Section 5.1.2.

The DNI meta-model was designed to meet multiple criteria, among others the ease of use by non-experts. Optimizing the meta-model for the modeler's ease of use makes it more difficult to read for transformation developers. Fortunately, the DNI model provided by a modeler can be preprocessed using *in-place* transformations (so called *DNI-to-DNI* transformations) to adapt it to the needs of the transformation developers without loosing any information. I describe selected *in-place* DNI model transformations in Section 5.1.3.

5.1.1 Model Validity Checking

A *valid* DNI model shall satisfy the constraints specified in the meta-model, that is, the cardinality of required features must be met and the Object Constraint Language (OCL) constraints cannot be violated. Basic constraints are defined in the meta-model, however not all constraints can be defined in the meta-model without bloating the meta-model definition with complex OCL invariants. Multiple complex OCL constraints can also cause building the model more difficult and this contradicts my aim to make DNI easy to use by non-experts. Instead of adding multiple OCL constraints, I decided to use only one (i.e., entities must have unique IDs) and provide a validation script to provide simple pre-transformation model analysis. The validation script is implemented using Epsilon Validation Language (EVL) and contains few rules that analyze model structure. I briefly discuss selected validation rules in the following.

- **Connections between Virtual and Physical Entities.** In reality, it is impossible to directly connect two virtual machines with a link if the machines are hosted on different nodes. The same holds for DNI. I report a DNI model as invalid if at least on the following conditions hold. Assume *nodeA* is connected directly with a `Link` to *nodeB*. Moreover, *nodeA.hostedOn<>null* and *nodeB.hostedOn<>null*. The model is invalid if *nodeA.hostedOn <> nodeB.hostedOn*. Moreover, this rule disallows connections between the `Nodes` that are not on the same virtualization level, whereas the virtualization level is understood as the number of `Nodes` on which a `Node` is hosted.

- **Completeness of Routing Information.** It is required that the traffic generated by DNI traffic sources can reach its destinations. This means that there must exist at least one complete route (i.e., set of `Direction` entities) between each pair of nodes that communicate with each other. Additionally, for each

`SDN Intermediate Node` there must exist at least one route to a `Node` that hosts the `SdnController` assigned to the `SDN Intermediate Node`.

In reality, it is allowed to run a network with incomplete or erroneous routing configuration. Network devices are able to pick a default destination or drop the traffic that cannot be forwarded. In contrast to this, DNI requires that all sources and destinations of a `Flow` have at least one valid route defined in the model, where the route is specified using a set of `Direction` entities.

5.1.2 Transformation Parametrization

Model transformations interpret the information included in the input DNI model and produce an output model using a set of rules. The way in which a transformation interprets the DNI information can be parametrized, so that the users receive the predictive models according to their expectations.

Most of the DNI parameters and entities are not obligatory. For example, a performance description of a `Intermediate Node` can be omitted if it is unknown. The assumed interpretation of missing model parameters (as long as the model is valid) is as follows: the unspecified parameters or entities are not limiting the performance, that is, the performance is infinite and there are no processing delays. However, the user might want to omit the performance to indicate that, for example, a given `Node` does not process any traffic and drops all incoming messages. This could be achieved by setting the respective bandwidth or capacity parameter to zero, however such transformation behavior must be known to the modeler. I propose two interpretations of missing performance-related parameters to either *infinite performance* (default) or *drop all traffic*. The user may configure the expected behavior for the transformations.

Similar parametrization rules are offered for the SDN rules in the *DNI-to-QPN* transformation. An `SdnFlowRule` allows to specify three probabilities to define the forwarding mode in which an `Intermediate Node` operates: software SDN, hardware SDN, or non-SDN forwarding. If three probabilities add up to one, there is no ambiguity. Disambiguation arises if the sum of the probabilities is less than one. Modeling *probabilityHardware* as 0.5, *probabilitySoftware* as 0 and *probabilityController* as 0.1 opens two possible interpretations: (a) the probabilities should be normalized, so that the sum is 1.0, or (b) the probability of dropping a packet on the node is $1.0 - 0.5 - 0.1 = 0.4$. Selecting one of the two modes defines another possibility for the user to configure the transformation behavior.

I assume that the obtained predictive models can be solved without any other parametrization. This means that some fine-grained predictive models may need to be solved with default parameter values. For example, *OMNeT++INET* defines default values for TCP configuration. Other solvers require instructions regarding solving methods. For example, default configuration of the LQNS solver offers

simulation-based solution; a proper change in the configuration enables an analytical LQN solver. *SimQPN* requires selecting a solving method and parameterizing the warm-up phase and stopping criterion. On the other hand, *OMNeT++*-based solvers do not require (despite the possibility) any extra parametrization regarding the solving method. To guarantee that the solving process can be started automatically straight after transforming a DNI model, the respective transformations need to pass the solver-specific parameters to the solvers (or generate default values of these parameters). The configuration of the solver is considered as another possible parameterization of the model transformations.

The transformation parameters are defined in the DNI tooling. The tooling technically consist of a Java-similar application or *Apache ANT* scripts that run the respective transformations followed by starting the solvers where possible.

5.1.3 In-Place DNI Transformations

To ease the development of the transformations, some DNI entities are processed using in-place transformations. I briefly discuss selected in-place DNI transformations.

Routing Directions Aggregation

Based on DNI `Directions`, `FlowRoutes` are build. The `FlowRoute` represents a network path using an ordered list of network interfaces that need to be traversed to reach the destination of a `Flow`. Each `FlowRoute` references single `Flow` and specifies the *probability* that describes the chance of selecting a given path if multiple are available. An example of the routing in-place transformation is presented in Figure 5.3.

The example demonstrates two directions that define load-balancing. In fact, the DNI model contains two additional `Direction` objects: first for node *N1* and second for node *N3*, but they are abstracted in the example. Based on the `Directions`, two `FlowRoutes` are built—each with probability 0.5 for each redundant connection between nodes *N2* and *N3*.

`FlowRoute` representation is compacter and easier for the transformation developer to work with—one can query the model to return all `FlowRoutes` that reference a given `Flow`. This returns the information about all available traffic paths in the network along with the probabilities of selecting given path by the traffic.

Flattening Workload Branches

DNI `Workloads` may contain `Branches`. A `Branch` defines that a given traffic source spawns a new thread and starts behaving as two independent traffic sources. Although it is easier for the modeler to work with branches, it is more challenging to process a workload including branches in the transformations. Thus, the second

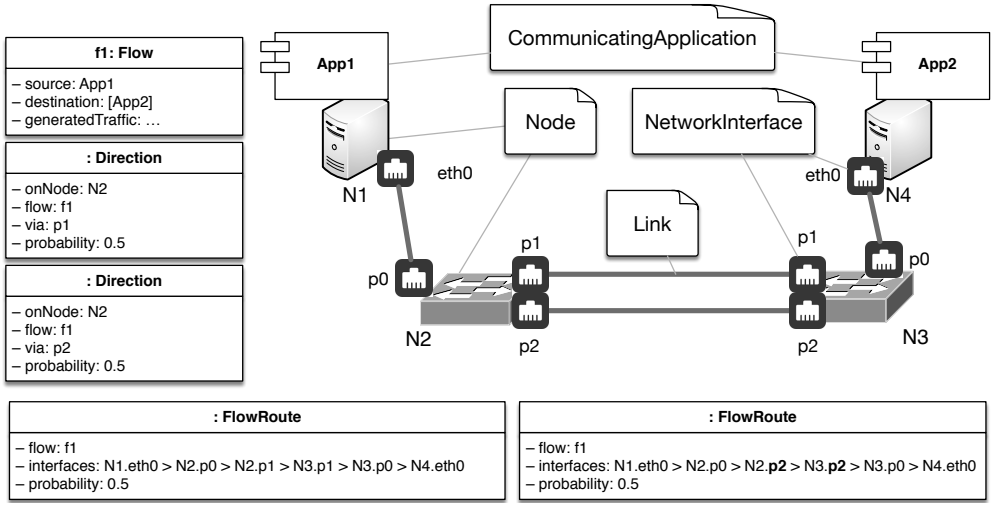


Figure 5.3: In-place transformation building `FlowRoutes` based on `Directions`. Demonstrated using example presented formerly in Fig. 4.10.

in-place DNI transformation removes branches, so that every DNI `TrafficSource` produces `Workload` containing only `Transmit`, `Wait`, and `Loop` actions.

Figure 5.4 depicts an example representing a `Workload` before and after applying the in-place branch removal transformation. The in-place transformation searches

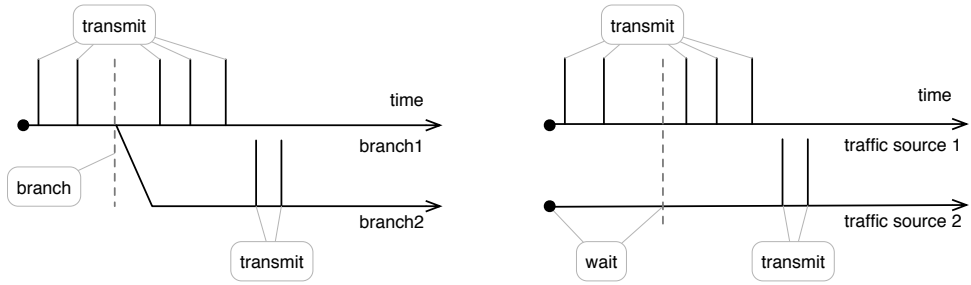


Figure 5.4: Example demonstrating in-place DNI transformation removing `Branches` from `Workloads`. Left: workload with a branch, right: two workloads without branches.

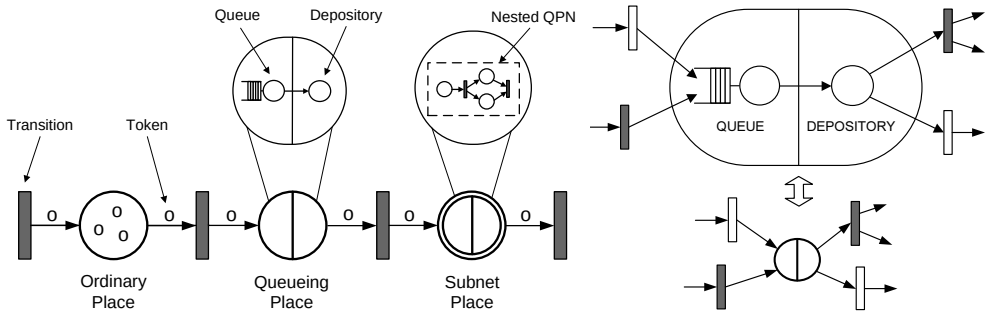
the workload and removes each `Branch` action. For each `Branch` removal, new `Workloads` are created—one for each branch of the `Branch` action. The first branch is left in the original workload. The search continues for all workloads, including the newly separated ones. The transformation adds a `Wait` action to each newly separated workloads to represent the time between the `Start` and the `Branch` actions in the original `Workload`.

5.2 Steady-State Performance Analysis with Queueing Petri Nets

In this section, I describe the transformation that transforms an instance of the DNI meta model to a QPN model. QPNs [Bau93b] are a combination of classic Queueing Networks (QNs) [BGdMT06] and Colored Generalized Stochastic PNs (CGSPNs) [CDFH93]. While CGSPNs are a powerful formalism to describe the synchronization and timing behavior of software programs, they lack the expressiveness to easily describe the scheduling of jobs at hardware resources. In addition to ordinary places and transitions known in CGSPNs, QPNs therefore introduce queueing places consisting of a queue and a depository. The queues correspond to those in a traditional QN, including a scheduling strategy and a service time distribution. Incoming tokens are first served in the queue and then put into the depository where they become available to outgoing transitions. Using QPNs, it is possible to model both software and hardware contention of software systems in a single model [KB03]. For solving QPNs, I use *SimQPN* simulator [KB06], which is part of Queueing Petri Net Modeling Environment (QPME) [SKM12b].

5.2.1 QPN Notation

The QPN formalism was introduced by Bause in [Bau93a]. The graphical notation used in this section is summarized in Figures 5.5 and 5.6.



(a) QPN notation elements. (b) Queueing Place.

Figure 5.5: Notation used in QPN diagrams. Excerpted from [KBB+11].

A QPN consists of set of places and set of transitions. Tokens can be grouped in to classes that are distinguishable by colors. In a transition, the incidence function defines the number of tokens required in each preceding place to be ready to fire. When a transition fires, it consumes the tokens from the preceding places and deposits defined number tokens in the succeeding places. A transition can fire in many ways—the different firing possibilities are referred to as modes. The graphical notation of a transition incidence function is presented in Figure 5.6.

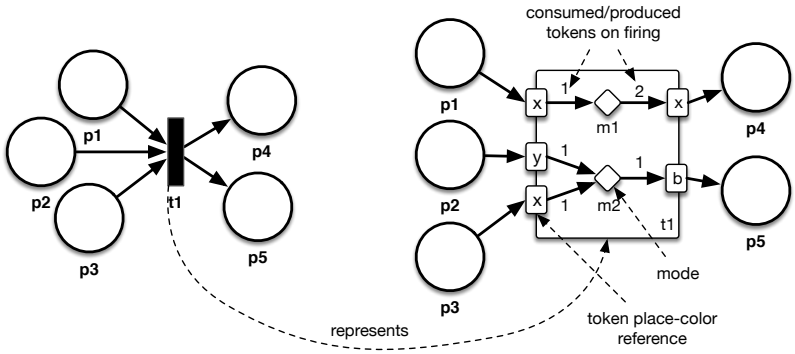


Figure 5.6: Notation used in QPN transitions.

Comparing to classical QNs, QPNs introduce a new type of place: the queueing place. A queueing place consists of a queue, and a depository. Tokens are placed inside the queue according to a certain scheduling strategy. The service time is defined through a parameter. Once a token has completed its service, it is put into the depository, which then behaves like a regular (ordinary Petri Net (PN)) place. Only tokens in the depository are considered available for the succeeding incidence functions. Furthermore, QPNs can be nested using subnet places that contain an arbitrary subnet with an obligatory single *input* and *output* place.

5.2.2 Network Topology

The transformation begins with translation of all network nodes into subnet places. For each *Node*, a subnet is created. To reflect the topology, the subnets are connected with links that are represented by two transitions—each for one of both directions. The incidence functions for links are defined as non-blocking, that is, they fire immediately for a single token of any color and deposit the token in the succeeding place in the graph. As an example, a QPN representing three end nodes connected with a single intermediate node is depicted in Figure 5.7.

5.2.3 Node

The internal structure of the subnet representing a `Node` is depicted in Figure 5.8. All incoming tokens are placed in the *input-place* first. Next, they are forwarded to the queueing places that represent receive queues of the network interfaces (*port-#-rx*). The *traversing-transition* has two tasks: (a) deleting the tokens that have destination in this node, (b) passing the traversing tokens to the *traversingTraffic* place. No tokens are passed to the traffic sources as the DNI meta-model and the transformation support only open workloads (exception: SDN controller can accept incoming packet-in messages—will be discussed later). Next, the *sdn-transition* gathers the traversing tokens and the tokens generated in the traffic sources and

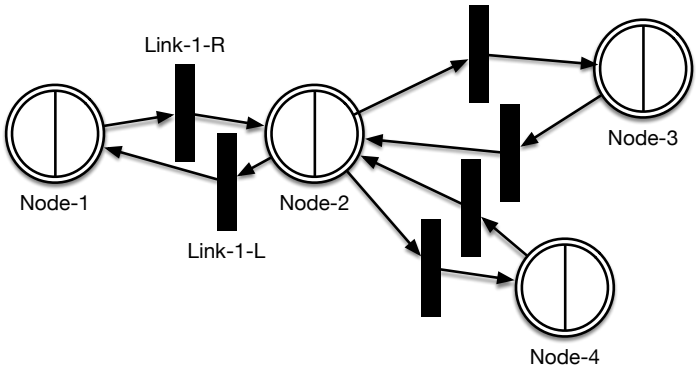


Figure 5.7: QPN representation of network nodes and links. The internal structure of `Node` subnets is presented in Fig. 5.8.

forwards them to the *switching* place or the *sdnSwitching* subnet based on the `Node` configuration. For a `Common Node`, all tokens are routed to the respective *switching* place. For `SDN Node` the tokens are routed via *sdnSwitching* if a matching `SdnFlowRule` exists. Once the switching delay is applied in the *switching* or the *sdnSwitching* place, the tokens are directed to the proper *port-#-rx* and leave the `Node`. Graphically, I depict the internal structure of the *sdn-transition* in Figure 5.9a.

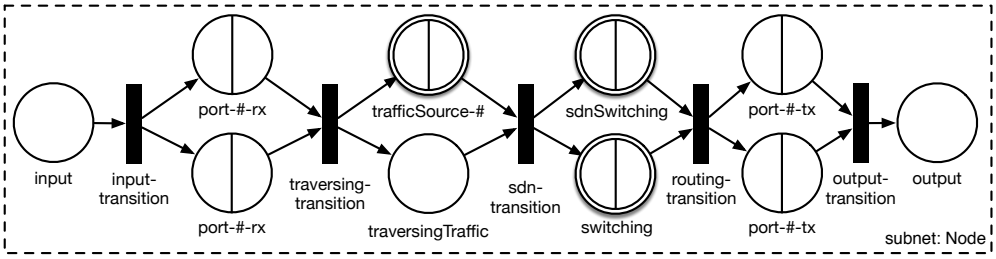
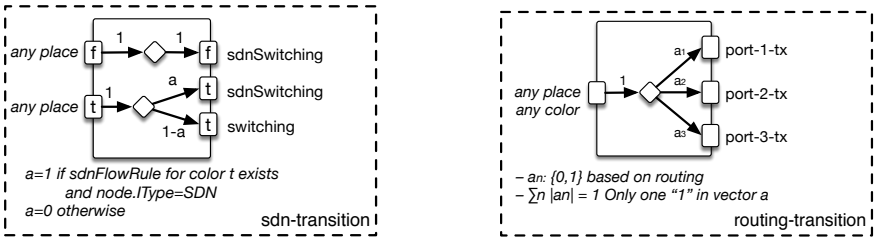


Figure 5.8: QPN representation of DNI's `Node`. Figure depicts the internal structure of a `Node` subnet place.

End and Intermediate Node

Each `End Node` represents a machine that can host `Communicating Applications` stack that may contain multiple `Traffic Sources`. An `End Node` without traffic sources specified in the input DNI model may be removed from the QPN model or may be transformed without building the respective *trafficSource-#* subnets. Similarly, each `Intermediate Node` is transformed into QPN subnet presented in Figure 5.8 with an empty set of *trafficSource-#* subnets.



(a) *sdn-transition*. (b) *routing-transition*.

Figure 5.9: Internal structure of transitions for non-load-balanced scenario. Token colors: *t* traffic, *f* flow-mod.

SDN and Common Node

The processing of the traffic in and `SDN Node` is conduced in the *sdnSwitching* subnet place. For a `Common Node` (or if a given flow is not forwarded in the SDN mode), the processing is done in the *switching* subnet place.

The *sdnSwitching* subnet groups entities that are responsible for forwarding in `SDN Nodes`. I depict its internal structure in Figure 5.10. Two separate subnet places handle the traffic switched in the *software-* and *hardware SDN switching mode*: *hw-* and *swSdnSwitching* respectively. The third switching mode—via the controller—is handled with the *buffer* and *toController* ordinary places and *tr-FM* and *controller-tr* transitions.

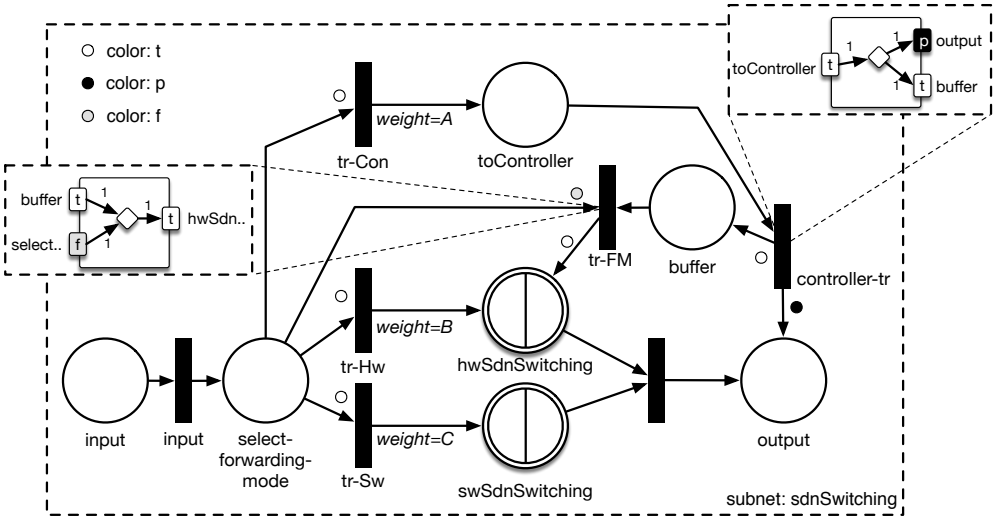


Figure 5.10: Internal structure of the *SdnSwitching* subnet place. Token colors: *t* traffic, *f* flow-mod, *p* packet-in.

The decision about the processing path in the SDN forwarding mode is executed in three transitions: *tr-Con*, *tr-Hw*, and *tr-Sw*. The arriving traffic tokens are deposited in *select-forwarding-mode* ordinary place. Next, the succeeding transitions fire according to their weights. The weights of the transitions are modeled using three parameters A, B, and C. Each of the three transitions will deposit one traffic token in one of the succeeding places.

The values of weights A, B, and C are related to the probabilities defined in the respective DNI's `SdnFlowRule` as follows. First, A, B, C are real numbers between zero and one: $A, B, C \in \mathbb{R}$ and $A, B, C \in [0, 1]$; Next, $A =$ *probabilityController*, $B =$*probabilityHardware*, and $C =$*probabilitySoftware*. For multiple colors representing traffic (*t* in Fig. 5.10), transitions *tr-Con*, *tr-Hw*, *tr-Sw* must contain multiple modes (or be represented as multiple single-modal transitions) for each traffic color because the weights are defined for each flow separately.

The traffic tokens directed to the SDN controller are forwarded to the *toController* place. Next, the *controller-tr* transition issues a new token with color *p* (*packet-in*) and forwards it to the *output* place and further to the node where the controller is deployed. At the same time, the traffic token *t* is deposited in the *buffer* where it waits for the controller's response. When controller replies (using *flow-mod* token *f*), the traffic token is released form the buffer and forwarded via the *hwSdnSwitching* place.

Forwarding Performance

The forwarding performance of an `Intermediate Node` is modeled using three subnet places: *switching*, *hwSdnSwitching*, and *swSdnSwitching*. Each place is responsible for representing the performance in native, SDN hardware, and SDN software switching mode respectively. The internal structures of the *switching*, *hwSdnSwitching*, and *swSdnSwitching* subnets are identical. THeir structure is presented in Figure 5.11. The forwarding performance is modeled using three queueing

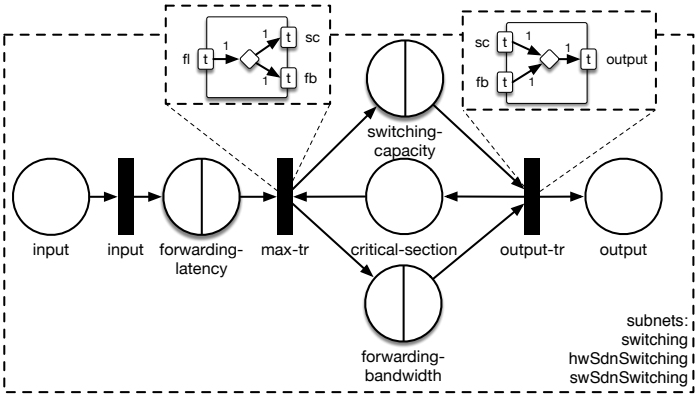


Figure 5.11: Internal structure of the *switching*, *hwSdnSwitching*, and *swSdnSwitching* subnet places.

places, each corresponding to a parameter from the `IntermediatePerformance` DNI entity. The *forwarding-latency* place models the *forwardingLatency* parameter of the `IntermediatePerformance` and so on. The transitions *max-tr* and *output-tr* represent *max* function that applies the larger delay to the traffic. The *critical-section* asserts that the maximum function operate on maximally two tokens at the same time.

SDN Controller

The *SdnController* subnet—depicted in Figure 5.12—is responsible for receiving the *packet-in* tokens (color p) and replying with the *flow-mod* tokens (color f). The *SdnController* subnet is located in the `Node` subnet in the same column as the *trafficSource-#* subnets. The tokes arriving to the controller are delayed twice. First, the delay of the controller is applied (controllers are complex software) and then the delay of the respective `SdnControllerApp` is added. The representation presented here can be enhanced using more detailed software modeling provided by Descartes Modeling Language (DML) [KHBZ16].

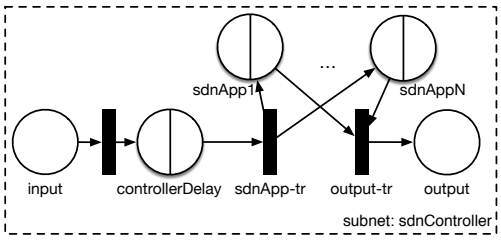


Figure 5.12: QPN representation of DNI's `SdnController`.

The *packet-in* and *flow-mod* tokens are routed using DNI `Directions` that bind the *openFlowEndPoint* of an `IntermediateNode` with the `Node` that hosts the `SdnController` and its applications. For each traffic color a separate set of *packet-in* and *flow-mod* colors is generated.

Load Balancing

Tokens that represent the load-balanced flows are processed normally with exception of the *routing-transition*. The *routing-transition* (as depicted in Fig. 5.8) is replaced by multiple transitions, each with a single mode but different firing weight. The firing weight represents relative firing frequency of the transition, so that it can properly represent load-balancing ratios. An example of SDN switching and "$60/40$" load-balancing is depicted in Figure 5.13. In this example, the tokens consumed from the *sdnSwitching* place are interchangeably deposited to port *port-1-tx* and *port-2-tx* with probabilities 0.6 and 0.4 respectively. For multiple traffic colors, the load balancing transitions *routing-transition-#* must contain multiple modes (or be represented as multiple single-modal transitions) for each traffic color because the weights are defined for each flow separately.

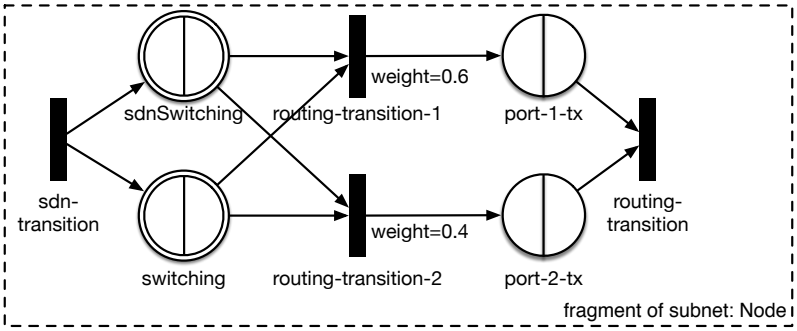


Figure 5.13: QPN representation of a `Node` with "60/40" load balancing for a single flow.

5.2.4 Virtual Nodes

DNI's `Nodes` can host other nodes to represent server virtualization scenarios. A `Node` that is hosted on another `Node` has identical internal subnet place structure to a regular `Node` (presented in Fig. 5.8). The hosting and hosted `Nodes` are connected using a queueing place named VMM (named after the virtual machine monitor). The VMM represents the overheads caused by the virtualization. In networking scenarios, the virtualization overhead is caused mainly by a virtual switch connecting the physical and virtual part of the environment. The structure of a QPN subnet representing a `Node` that host other `Nodes` is presented in Figure 5.14.

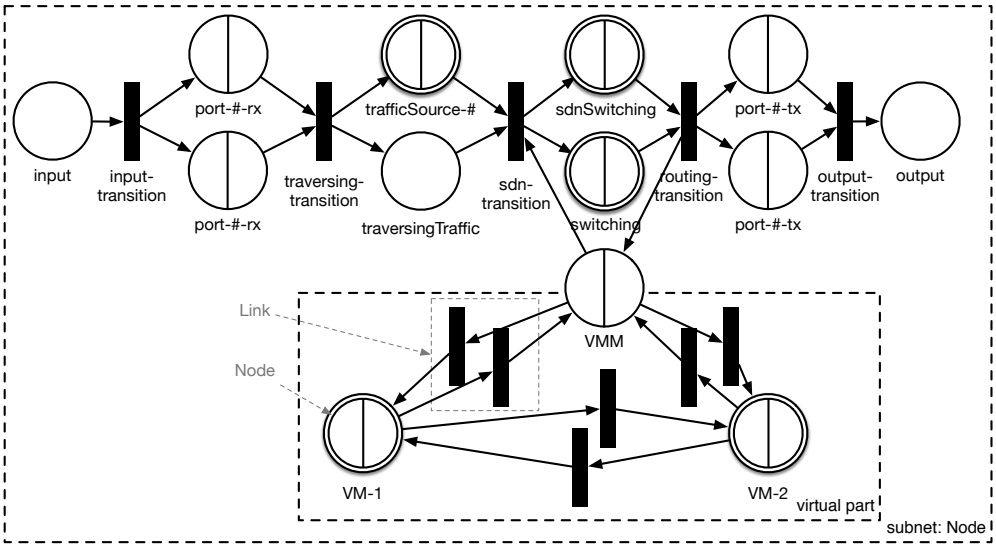


Figure 5.14: QPN representation of a `Node` hosting virtual nodes.

The upper part of Figure 5.14 represents a typical structure of a `Node` subnet. In the bottom, a virtual part is distinguished. The virtual part is connected to the physical part using the VMM queueing place. Note, that traffic originating and terminating in the virtual part needs to be forwarded via the physical part. This leads to *double forwarding*, which is an expected behavior, if a flow traverses the virtual part of a `Node`. First, the traffic is directed to the *sdn-transition*, *switching* or *sdnSwitching* places, and then it is directed to the virtual part in the *routing-transition*. Once the processing in the virtual part is finished, the traffic is directed to the *sdn-transition* in the physical part, so the forwarding in the *switching* or *sdnSwitching* place is conducted again.

The virtual part presented in Figure 5.14 can be freely defined and may represent an entire network (e.g., as presented in Fig. 5.7) including complex topologies. The network modeled in virtual part can be hierarchical and can contain other "virtual" networks.

Although *virtual-virtual* networks are supported in both DNI and QPN, the modeling requires special caution as the meta-model constraints do not report erroneous connections between virtual network levels. For example, the default DNI editor forbids modeling a direct connection between a physical and virtual `Node` (this is possible only via VMM) but no error is shown if a *virtual-virtual* network hosted on *node 1* is connected with a `Link` to a *virtual* network hosted on node *node 2*.

5.2.5 Traffic Source

A `TrafficSource` is represented as a subnet place in the QPN model. The main responsibility of the traffic source subnet is the generation of tokens according to the workload defined in the DNI model. Figure 5.15 depicts a QPN model of an exemplary traffic source. All unnamed transitions are generated as mandatory connections between two consecutive places; these transitions are passing all token colors in a non-blocking fashion.

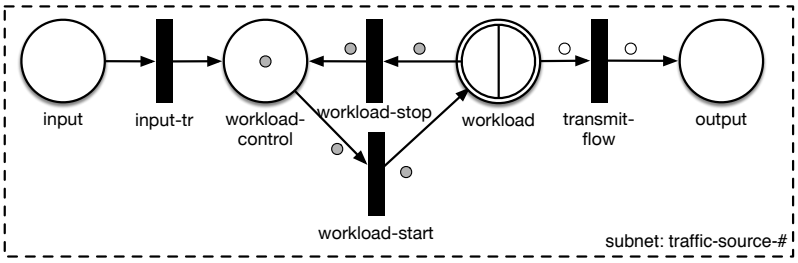


Figure 5.15: QPN representation of the TrafficSource. Token colors: gray workload execution token (WE) (workload-execution), white traffic.

To control the order of actions executed in the workload, the *workload-execution* color (WE for short) is defined. The *workload-control* place contains initial marking

(tokens at the start of the simulation) of a single WE token. Later, the WE token traverses the places and transitions according to the order defined in the DNI traffic workload model.

The traffic generation procedure runs as follows. No tokes arrive to the input place of the *traffic-source-#* as all are ignored and destroyed in the preceding transition (due to the assumed open workload model). The WE token (initial marking) is passed to the *workload-start* transition that stems from the *StartAction* defined in the DNI workload model. Next, the WE token is passed to the subnet representing the workload of the modeled `TrafficSource`. Once the workload execution is ready, the WE token is passed back to the *workload-control* place via the *workload-stop* transition. Then, the entire process is repeated. The example of a traffic workload modeled with a *workload* subnet is presented in Figure 5.16.

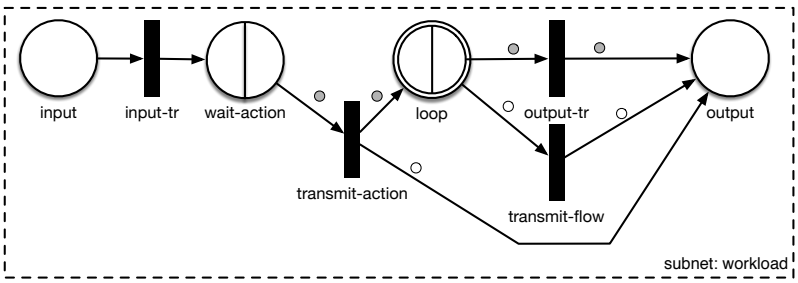


Figure 5.16: QPN representation of the a traffic workload example containing actions: wait, transmit, loop. Token colors: gray WE, white traffic.

Each `WaitAction` is represented using a queueing place. A `TransmitAction` is transformed to *transmit-action* transition. A *transmit-action* transition passes WE token and produces a token representing a flow (white in Fig. 5.16). The flow token is immediately deposited in the output place and is ready to be routed and transmitted by the end node. The WE token is passed further to the next action in the workload (loop in this example) until it reaches its end in the *output-tr* transition. Then, the WE token is passed to the higher subnet in the hierarchy as the workload generation process is hierarchical and recursive. The traffic tokens generated in the *loop* subnet are passed to the *output* using a separate transition *transmit-flow*.

The *LoopAction* is represented as a subnet and its internal structure is depicted in Figure 5.17. A single WE token arriving to the input is transformed into multiple tokens in the *1-to-num-loop-iter* transition (in the example presented in Fig. 5.17, the loop iterates three times). The transition produces exactly the amount of tokens that corresponds to the number of loop iterations defined in the DNI model. The next transition—*loop-start*—is a synchronization point; it requires one token from the *loop-iter-left* and another one from the *loop-control* place. The latter has a single WE token set as initial marking, so that the *subworkload* can start as soon as the loop subnet receives a WE token from outside.

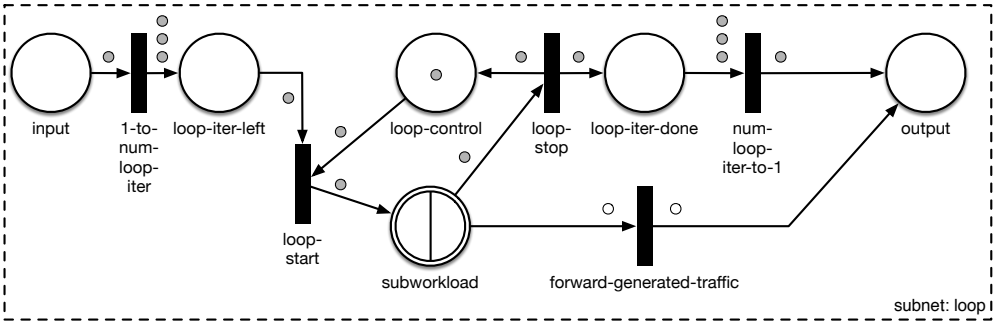


Figure 5.17: QPN representation of the workload loop action with three iterations. Token colors: gray *WE*, white traffic.

When all actions of the *subworkload* are finished, the *loop-stop* transition duplicates the WE token and passes one to the *loop-iter-done* and the second one to the *loop-control* place. Now, the next loop iteration can begin, as long as there are WE tokens in the *loop-iter-left* place left. When the *loop-iter-done* place contains the amount of tokens equal to the number of iterations (three in the example presented in Fig. 5.17), the *num-loop-iter-to-1* transition fires, consumes all input tokens and deposits a single WE token int the output place. The execution of all loop iterations is finished.

The actions that are looped are represented as a subnet for brevity and modularity. The internal structure of the *subworkload* subnets corresponds to the traffic source subnet presented in Figure 5.16. This allow to specify the workloads hierarchically and improves the maintainability of the transformation. The tokens that represent traffic flows are directly sent to the output place to be immediately passed to the *sdn-transition* in the respective parent node subnet.

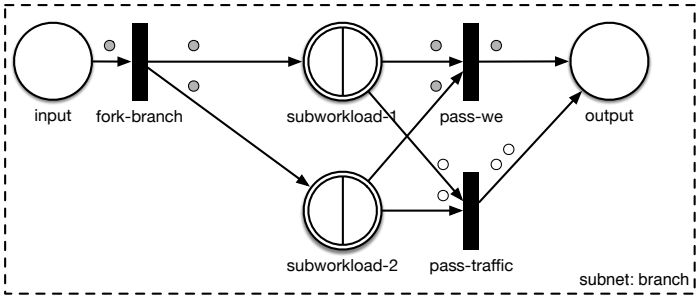


Figure 5.18: QPN representation of the workload fork and branch actions. Token colors: gray *WE*, white traffic.

The structure representing a `BranchAction` is depicted in Figure 5.18. The incoming WE tokens are multiplied in the *fork-branch* transition and passed to the *subworkload* subnets. Once the processing in the *subworkload* subnets is finished,

the WE tokens are transformed back to a single on in the *pass-we* transition. The *pass-traffic* transition passes all traffic tokens generated in the *subworkload* subnets without reduction of its quantity. In the practice, the branches are usually removed from the model by the flattening described in Section 5.1.3.

5.2.6 Routing Information

The information included in DNI's `NetworkConfiguration` is used, among others, for proper wiring of a resulting QPN. The DNI `Directions` and `Flows` define possible paths of communication in a network. A `Flow` identifies the communicating parties, whereas `Directions` define which paths are used for the communication.

For each DNI `Flow`, at least one set of `Directions` must exist, so that the flow's source and destinations can be reached in the graph defined by `Nodes` and `Links`. Providing incomplete routing information (e.g., missing `Direction` in DNI) cause the resulting QPN to be not live.

DNI `Directions` are used in the *routing-transition*. The parameter *via* defines which *port-#-tx* queueing place handles the tokens representing a given flow. The wiring is transformed by defining a transition mode that connects the preceding place (*sdnSwitching* or *switching*) with the succeeding queueing place.

Information modeled as `SdnFlowRule` defines the modes of *sdn-transition*s. Tokes representing a `Flow` are directed to the *sdnSwitching* subnet if a `SdnFlowRule` exist for a given `Flow` and `Node`. Otherwise, the tokens are processed by the *switching* subnet. The mapping of flows to QPN token colors is discussed in Section 5.2.7.

5.2.7 QPN Colors and Traffic Clustering

The QPME editor and the *SimQPN* simulator divide tokens into classes called colors. The tokens having the same color are indistinguishable. Thus, all important components of the simulated traffic (e.g., flows) must be modeled as separate colors. Except of the WE tokens described in Section 5.2.5, the transformation generates multiple token colors related to the modeled traffic.

The traffic-token colors are related to the traffic modeled in DNI. The colors are automatically generated based on the traffic model. The transformation generates a separate color for each flow in the model. Additionally, if a `Flow` has multiple `FlowTraffic` descriptions, each `FlowTraffic` is represented with a color. Multiple `FlowTraffics` can be *clustered* as well to reduce the number of colors and thus make the modeling granularity coarser.

The traffic clustering helps to reduce the number of colors and thus shorten the *SimQPN*'s solving times. I assume, that the message size defined in a `FlowTraffic` entity can be clustered. For example, for a *dataSize* modeled as normally-distributed $N(\mu, \sigma^2)$, the transformation may generate a single color for the flow with $dataSize = \mu$, or three with the following data sizes: $\mu - \sigma$, μ, and $\mu + \sigma$. The modeler decides how many clusters shall be used for modeling traffic as this affects the prediction accuracy and the solving time. By decreasing the number

of clusters, I accept the loss of prediction accuracy but expect shorter simulation times.

Additionally, each traffic-token color has additional pair of colors for representing *packet-in* and *flow-mod* tokens in SDN scenarios. In case of multiple SDN switches and controllers, each traffic color has multiple pairs of *packet-in* and *flow-mod* colors—each color pair for one switch-controller pair. This may lead to the explosion of colors and should be carefully modeled because the *SimQPN* solver can handle maximally $256^3 = 16\ 777\ 216$ colors.

5.3 Abstracting DNI with miniDNI and Solving with QPN

Sometimes, a quickly conducted but less accurate performance prediction is more valuable than a long-running but more accurate simulation. To the examples that leverage quick performance prediction I include: (1) run-time reconfiguration of resources to handle a workload spike, (2) trigger-based scaling of the infrastructure, (3) rapid self-reconfiguration due to a failure. For such cases, a coarse model-based prediction is of higher value than an expert's educated guess. To other examples where coarser modeling can be applied, I account all scenarios with insufficient data to feed and calibrate the complete model. In both cases, the modeler may decide to use a coarser model instead of DNI.

In this section, I present a transformation that converts a DNI model into a miniDNI model. The transformation transforms DNI models into respective miniDNI models. However, miniDNI models can be also built or extracted directly from a running system without needing to build a DNI model. The available extraction approaches are discussed in Sections 6.1 and 6.2.

5.3.1 Transformation of DNI to miniDNI

In the *DNI-to-miniDNI* transformation selected information is abstracted because the *miniDNI* model contains less detail than the respective DNI model. I provide an overview of transformation rules in Table 5.2.

As first, the transformation processes all DNI `Nodes`. A DNI `Node` is transformed to miniDNI `Node`, disregarding of its `IPosition` (`End` or `Intermediate`) and `IType` (`SDN` or `Common`). The DNI performance descriptions related to `Node` (`EndPerformance`, `IntermediatePerformance`, `SdnNodePerformance`) are simplified and aggregated in the miniDNI `NodePerformance`. The `IntermediatePerformance` is reflected in the *forwardingThroughput* parameter; the `EndPerformance` is transformed to the *softwareLayersDelay* parameter; and the `SdnNodePerformance` transforms into the following three parameters: *forwardingThroughput-sdn-hw*, *forwardingThroughput-sdn-sw*, and *forwardingThroughput-sdn-controller*.

The information about forwarding latency and switching capacity is abstracted from the miniDNI model. The transformation recalculates the value of the forwarding performance, so that the values stored in miniDNI are the minimal

Table 5.2: *DNI-to-miniDNI* transformation rules.

DNI	miniDNI	Comments
`Node`	`Node`	All miniDNI `Nodes` share the features of DNI's `End`, `Intermediate`, and `SDN Nodes`.
`Link`, `NetworkInterface`	`Link`	A DNI `Link` connecting two `NetworkInterfaces` is transformed into a miniDNI `Link` connecting two miniDNI `Nodes`.
`Communicating-Application`, `TrafficSource`, `Flow`, `Workload`, `Workload- Actions`	`TrafficSource`	All DNI information about software is abstracted and presented as a miniDNI `TrafficSource`. Information about traffic generated by a traffic source is aggregated into a single `TrafficSource` entity with parameters: *messageSize* and *numberOfMessagesPerSecond*.
`Network-Configuration`	`Route`, `FlowTable`	The information about DNI `NetworkProtocol` is abstracted from miniDNI, whereas DNI `Directions` and `SdnFlowRules` are simplified.

expected throughputs (cause by either limited switch capacity, forwarding latency, or forwarding bandwidth).

The DNI `NetworkInterfaces` and `Links` are transformed into miniDNI `Links`, so that a triple *(network interface, link, network interface)* is represented with the miniDNI `Link`. The performance descriptions of the `NetworkInterfaces` and the `Link` are aggregated to represent the slowest of the three DNI equivalents.

The DNI `NetworkTraffic` sub-meta-model is transformed into a single miniDNI `TrafficSource`. A `TrafficSource` corresponds to DNI `TrafficSource`, however the structure in the miniDNI model is flat—no information about the `CommunicatingApplications` is preserved. The DNI `Workload` and its actions is also flattened and represented as messages that have size (parameter *bytesPerMessage*) and are generated with a given frequency (parameter *messagesPerSecond*).

Transformation of the DNI `Workload` into the miniDNI `TrafficSource` abstracts the most of the data. The traffic is presented as a flat stream of messages generated in constant intervals, so that all traffic patterns are neglected. The transformation *unfolds* and *flattens* (removes branches and loops) the workload graph of DNI until all `Branch` and `Loop` entities are removed. Next, the duration of the workload is calculated by summing the duration of all `Wait` actions. Similarly, the total volume of data is aggregated by summing the size of flows referenced by the respective `Transmit` actions. Finally, I assume that a single message per second is generated and set the parameter *messagesPerSecond* to 1 and calculate the size of a message by dividing the total traffic volume by the workload duration in seconds.

The values of both `TrafficSource` parameters may be freely defined by the modeler when a miniDNI model is built manually. In the transformation however, the approximation needs to be used as the values are calculated automatically. The miniDNI `TrafficSource` parameters may be also indirectly extracted using the approach presented in Section 6.2.

The miniDNI meta-model represents routing information using `Route` entities that reference `Nodes` in a given order additionally distinguishing the *start* and the

end of a route. Traffic generated by each miniDNI `TrafficSource` needs to follow a single route. Load balancing is not supported in miniDNI.

Information about SDN is represented using `FlowTables`. A `FlowTable` defines the forwarding mode in which a `Node` processes the traffic generated in `TrafficSources`. The forwarding mode can be selected from *traffic-native*, *traffic-sdn-hw*, *traffic-sdn-sw*, *traffic-sdn-controller*. A `FlowTable` references `Nodes` where the given traffic should be processed according to the given forwarding mode. For example, *flowTable1.traffic-native=traffic-source-1* and *flowTable1.onNodes=(node1, node2)* means that the traffic generated by *traffic-source-1* will be forwarded using native (non-SDN) mode on nodes *node1* and *node2*.

The obtained *miniDNI* model is a descriptive model and needs to be further transformed in order to deliver performance predictions. In Section 5.3.2, I present the *mDNI-to-QPN* transformation that transforms a *miniDNI* model into a QPN model.

5.3.2 Transformation of miniDNI to QPN

The QPN models resulting from the *mDNI-to-QPN* transformation differ from the models obtained in the *DNI-to-QPN* transformation (presented in Section 5.2). Both resulting QPN models represent the same network, but they differ in the amount of detail being modeled. In the following, I present the *mDNI-to-QPN* transformation and describe the abstractions it introduces.

The *miniDNI* meta-model describes the structure of a network using `Nodes` and `Links`. These two entities are mainly used to generate the structure of the respective QPN. Every `Node` is represented as a subnet. Connections between subnets are obtained by transforming `Links` into pairs of queueing places connected to Subnets using immediate transitions.

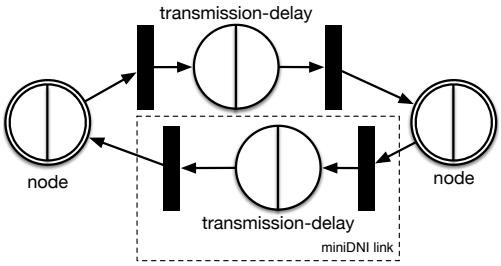


Figure 5.19: QPN representation of miniDNI `Network` including `Links` and `Nodes` in *mDNI-to-QPN* transformation.

The QPN representation of a `Link`, presented in Figure 5.19, consists two queueing places where contention effects from the network interfaces happen. The delays in *transmission-delay* places are calculated using information included in the `LinkPerformance` entities. Two pairs of immediate transitions are required by the QPN formalism to connect two consecutive places. The transitions contain

modes—one for each token color traversing the link. The colored tokens represent traffic in the QPN. There is exactly one color for each `TrafficSource`. Colors are assigned to places and transitions based on the information contained in the `Route` entities. The transformation reads the routing information and assigns a color to the place or transition if the respective link or node carries the traffic of the given flow.

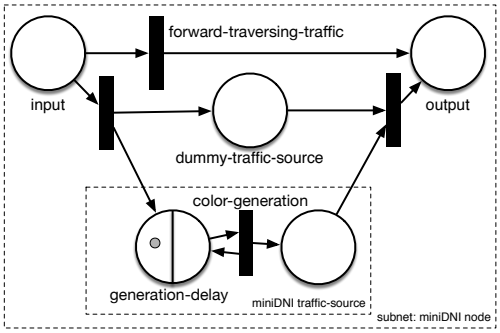


Figure 5.20: Internal structure of miniDNI `Node` subnet place including `TrafficSource` in *mDNI-to-QPN* transformation.

The colored tokens representing network traffic are generated in the `Nodes`. The structure of the QPN representing a `Node` is presented in Figure 5.20 and consist of three parts. The first part is the *forward-traversing-traffic* transition. It is responsible for processing the tokens from the input to the output place if the *Node* is neither the source nor the destination of the traffic flow represented by the token color. The transition is removed from the QPN model if a given node is not traversal for any color. The information about traversal nodes is derived from the `Route` entities. The second part consist of the ordinary place called *dummy-traffic-source*. This place is necessary to keep the QPN graph connected in case the respective node does not act as a traffic generator, nor a destination or a traversal node.

The third part of the subnet contains the set of traffic sources responsible for generating tokens representing traffic. Each traffic source (see dashed frame in Fig. 5.20) generates tokens of one color. A single token represents a single message of the given size. The intergeneration time—derived from *messagesPerSecond* parameter in the `TrafficSource` entity—is modeled as a parameter of the *generation-delay* places.

The transition between the *input* place and the traffic source is responsible for removing incoming tokens—it contains modes that remove every token that arrives to it. By such representation, I model the traffic as an open workload, exactly as specified in the original DNI model. Only tokens that have their destination in the given node are removed; other (traversing) tokens are passed to the *output* place using the *forward-traversing-traffic* transition.

5.4 Solving DNI with Discrete Time Simulation

DNI models may be transformed and solved with discrete time simulation. As an example of a discrete simulation framework, I present two solvers based on *OMNeT++* simulation framework [Var01].

First solver—named *OMNeT++INET*—and its model transformation was originally presented in [RZK13, RKZ13]. The solver is based on INET [Omn16] library that provides network models and algorithms present in the TCP/IP-based networks. The *OMNeT++INET* simulation does not cover SDN networks and its support is limited to protocols included in the INET library. The main goal of developing this solver was to reuse as many INET objects as possible and minimize the customizations in the simulation framework. The solver and the respective DNI transformation is presented in Section 5.4.1.

The second solver—named *OMNeT++generic*—and its model transformation is presented in Section 5.4.2. The second solver is based on *generic OMNeT++* discrete-time simulation. This means, that no external libraries are used in its development and all components are programed using the "vanilla" *OMNeT++* installation. This approach to simulation development simplifies the dependencies management and increases the solver life-time as the interfaces to external libraries do not need to be maintained anymore.

5.4.1 Classical Network: *OMNeT++INET*

INET [Omn16] library offers ready-to-use components for *OMNeT++* to simulate, among others, TCP/IP-based networks. The DNI meta-model covers wider scope than the modules provided by INET, so the *DNI-to*-OMNeT++INET transformation abstracts selected information included in DNI and narrows the scope to the elements available in INET. On the other side, the behavior of Transmission Control Protocol (TCP) is modeled in INET with much more detail compared to the description included in DNI. This leads to accepting default parameters offered by INET if an equivalent information does not exist in DNI. For example, DNI does not define any parameters of TCP whereas *OMNeT++INET* allows to define, for example, version of TCP (Tahoe, Reno) or initial window size. These factors need to be taken into account when using the *DNI-to*-OMNeT++INET transformation. The default parameters can be tuned manually in the simulation framework, however this requires that the user is knowledgeable with *OMNeT++*.

The transformation builds the *ned* and *ini* files used by *OMNeT++*. *OMNeT++* uses *ned* files for defining the structure of the network (topology, connections between modules, available parameters), whereas *ini* files are used to define values of the parameters defined in the *ned* files. The transformation does not generate any code that defines the behavior of the *OMNeT++* modules—it is assumed that the required code is delivered with the solver.

Table 5.3: *DNI-to-Omnet-INET* transformation rules.

DNI Entity	*OMNeT++* Entity	Comment
`Network Infrastructure`	`network`	A `network` in *OMNeT++* is a top-level entity and represents the simulated environment.
`End Node` *hosts=null*	`StandardHost`	
`End Node` *hosts<>null*	`VMM`	`VMM` is a custom module that contains a `Router` and `StandardHosts`.
`Intermediate Node`	e.g., `Switch` or `Router`	Depends on the value of `IntermadiateNode.type` enumeration.
`Link`	`Datarate-Channel`	Applies to both `PhysicalLink` and `VirtualLink`.
`Network-Interface`	`inout` gate of e.g., a `Router`.	Generated automatically for `StandardHosts` and `VMMs` based on the *connections* section of the *OMNeT++ ned* file.
`Network-Protocols`	represented directly	*OMNeT++* and INET simulate network protocols directly as long as DNI uses TCP, UDP, IP.
`Traffic-Source`	`TcpApp` or `UdpApp`	Choice of application type depends on the `NetworkProtocol` in the L4 (e.g., for TCP we use `TcpApp`).
`Workload`	configuration of e.g., `TcpApp`	The choice of concrete `Workload` depends on traffic generation implementation and configuration of `TrafficSources`.

Transformation Rules

The *DNI-Omnet-INET* transformation transform models by executing rules presented in Table 5.3. First, the main *ned* file describing a network is build. The top-level network represents the topology of the network and includes equivalents of the DNI `Nodes` and *Link*s. An example of *OMNeT++*'s top-level network is presented in Figure 5.21. The `End Nodes` are modeled using `StandardHost` module provided by INET, whereas `Intermediate Nodes` are represented as a `Switch` or `Router`. The transformation selects between a `Switch` or `Router` based on the `Protocols` in the `ProtocolStack` attached to the DNI `NetworkInterfaces` used by a given DNI `Node`. As the transformation supports only TCP, UDP, and IP protocols, the matching depends on the name of the protocol.

An `End Node` that hosts other `Nodes` is transformed into a *OMNeT++*'s VMM. The VMM does not exist in the original INET library, so I provide its implementation as an *OMNeT++* module. Its internal structure is presented in Figure 5.22 and contains a `Router` and an array of `StandardHosts`, which both are standard INET modules.

`NetworkInterfaces` are generated in *OMNeT++* automatically when two of DNI's `Node` equivalents (e.g., `StandardHost` and a `Router`) are connected. An example is presented in Listing 5.1—each `Node` (e.g., relate1, relatesw2) gets a *gate* named *ethg*.

The performance of a DNI `Link` and `NetworkInterface` is calculated in the transformation and transformed into a `ThruputMeteringChannel`. The `ThruputMeter-`

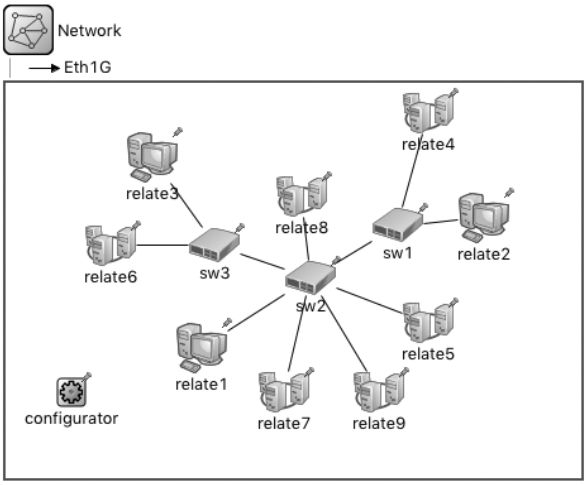


Figure 5.21: Example of a top-level *OMNeT++* network including modules: `StandardHost` (e.g., *relate3*), `Switch` (e.g., *sw1*), `VMM` (e.g., *relate4*).

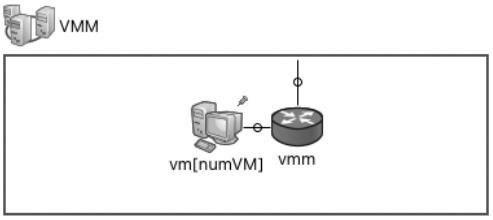


Figure 5.22: Internal structure of the custom `VMM` *OMNeT++* module.

`ingChannel` wraps the `DatarateChannel` and provides traffic statistics. Exemplary channel definition in a *ned* file is presented in Listing 5.2.

A DNI `TrafficSource` is transformed into *OMNeT++*'s `Tcp`- or `UdpApp` depending on the transport protocol used by the `Node` that hosts the `TrafficSource`. `Tcp` and `UdpApp` are standard INET modules with simplified specification of traffic generation behavior. As the default form of defining traffic is not satisfactory, I contributed modified versions of `Tcp` and `UdpApp`: `DNITcp` and `DNIUdpApp` respectively.

`DNI***Apps` allow to specify traffic generation behavior by defining time series of events in textual form. For example, a fragment: "send 800000 every 0.0010 for 999 times; send 1000 every 0 for 1 times;" means that 800 000 bytes will be generated 999 times in a loop with inter-generation time of 10ms. Then, a single 1000 bytes long message will be sent immediately. DNI `TrafficSources` with `Workloads` containing branches are transformed into multiple `DNI***Apps`—one

Listing 5.1: A definition of topology in *OMNeT++*. Nodes are connected using *channels* representing DNI `Links` and *gates* (ethg) representing DNI `NetworkInterfaces`.

```
connections allowunconnected:
    relatesw1.ethg++ <--> CHANNEL_098461 <--> relatesw2.ethg++;
    relatesw2.ethg++ <--> CHANNEL_064472 <--> relatesw3.ethg++;
    relate1.ethg++   <--> CHANNEL_035945 <--> relatesw1.ethg++;
    relate2.ethg++   <--> CHANNEL_021247 <--> relatesw1.ethg++;
    relate2.ethg++   <--> CHANNEL_079851 <--> relatesw2.ethg++;
```

Listing 5.2: A definition of a channel in *OMNeT++*.

```
channel CHANNEL_098461 extends ThruputMeteringChannel {
    datarate = 1.0Gbps;
    delay = 9.999999974752427E-7s;
}
```

`DNI***App` for each branch. The traffic specification format was later improved for the *OMNeT++generic* solver that is presented in Section 5.4.2.

Limitations

Simulation of a TCP/IP-based network in *OMNeT++* INET requires several further simplifications in order to comply with DNI. First, DNI defines explicitly which `Nodes` may communicate using a given path. In *OMNeT++*, communicating pairs and routing information need to be encoded in the form of static routing with predefined routes. Moreover, *OMNeT++* requires setting valid Internet Protocol (IP) addresses, whereas DNI identifies entries based on IDs. This may lead to a broadcast storm if the addressing information in DNI is missing or do not correspond to valid IP addressing schemes.

Similar problem appears if the modeled switches use Virtual Local Area Networks (VLANs) to separate logical networks (e.g., as defined in is IEEE 802.1Q). VLANs are modeled in DNI by explicitly defining possible communication paths for `Node` pairs. *OMNeT++INET* does not provide a configurable module to represent VLANs, so a `Router` must be used. Despite the limitations, the *OMNeT++INET* solver provides valid performance predictions for TCP/IP-based scenarios. Their evaluation is presented in Chapter 7.

5.4.2 SDN-based Network: *OMNeT++generic* Simulation

The second *OMNeT++*-based solver—named *OMNeT++generic*—is an independent *OMNeT++* simulation and does not require any additional *OMNeT++* packages or libraries. The main aim of this solver is to overcome the limitations of the *OMNeT++INET* simulation and support generic network protocols by abstracting

their behavior. The generic *OMNeT++* solver should support SDN and load balancing scenarios thanks to its generic character. I characterize the solver and its transformation in the following sections.

Nodes and Links

Similarly as in DNI, *OMNeT++generic* represents a network using nodes and links connected with network interfaces. An exemplary network is presented in Figure 5.23.

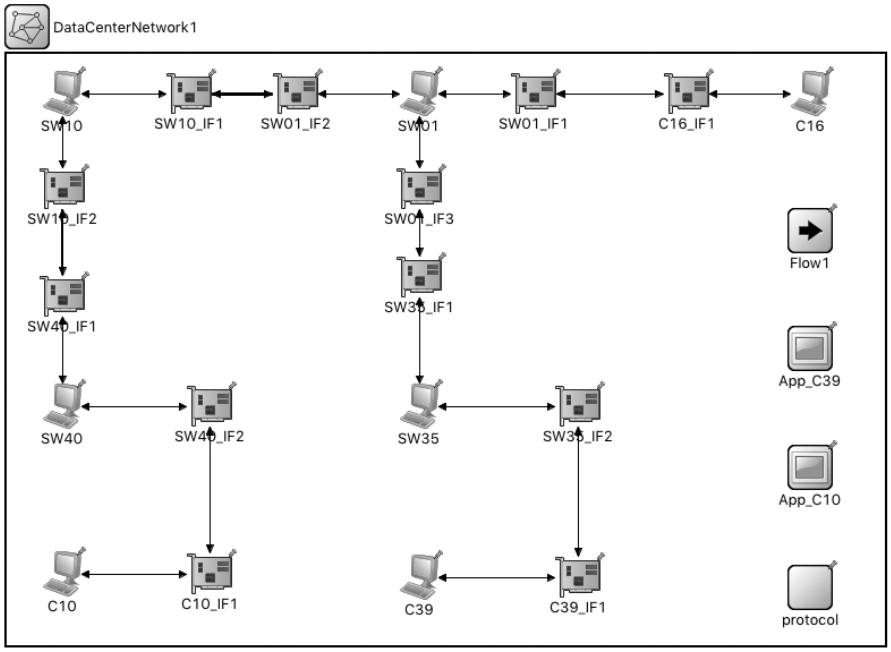


Figure 5.23: Example of a top-level *OMNeT++generic* network.

The example in Figure 5.23 includes the following *OMNeT++* modules: *node* (depicted as a desktop computer), *network interface* (depicted as a network card), *link* (depicted as black vectors connecting components), *flow* (depicted as box with arrow), *application* (depicted as a box with screen), and *protocol* (depicted as a gray box).

DNI `Links` are represented as *OMNeT++*`Channels`, which are generic *OMNeT++* entities. `Links` are grouped by their type (based on their maximal offered bandwidth) and represented in *OMNeT++* as channel types. Example is presented in Listing 5.3 where three types of channels are defined: 1Gbps Cat6, 10Gbps SFP+, and 40Gbps QSPF+. The channel types correspond to the cable types used in the real testbed. Any virtual `Links` (i.e., `Links` connecting VMs or a hypervisor)

Listing 5.3: Definition of channels in generic *OMNeT++*.

```
channel Cat6 extends ThruputMeteringChannel{
    datarate = 1Gbps;
    @display("ls=black,1");
}
channel SFP extends ThruputMeteringChannel{
    datarate = 10Gbps;
    @display("ls=,2");
}
channel QSFP extends ThruputMeteringChannel{
    datarate = 40Gbps;
    @display("ls=,3");
}
```

are defined as custom channel class, similarly as for *OMNeT++INET* presented in Section 5.4.1.

Channels connect network interfaces. A network interface in *OMNeT++generic* is represented with a pair of queues—one for transmitting, second for receiving data (see Fig. 5.24). Both queues have identical performance specification that stems from DNI's `NetworkInterface` and `NetworkInterfacePerformance` entities. Additionally, queues gather traffic statistics that are later reported as the results of the prediction.

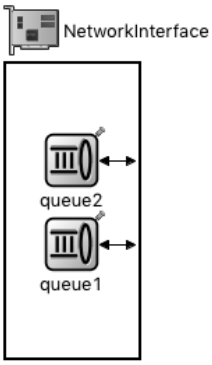


Figure 5.24: Internal structure of network interface in *OMNeT++generic*.

DNI `Nodes` are represented as *OMNeT++* custom modules called *node*. *OMNeT++*'s node represent any DNI `Node`, disregarding its `IType` (`SDN` or `Common`) and `IPosition` (`End` or `Intermediate`). An *OMNeT++* node is assumed to combine a `SDN`, `End`, and `Intermediate Node` in a single entity—the performance specification and hosted applications can successfully represent the relevant parameters included in DNI. In contrast to *OMNeT++INET*, *OMNeT++generic* does not represent the

Listing 5.4: A fragment of application, flow, and deployment configuration in generic *OMNeT++* solver.

```
**.App_C39.deployedOnNode = "C39"
**.App_C39.trafficXML = xmldoc("Flow1_Traffic.xml")

**.Flow1.uid ="Flow1"
**.Flow1.SourceSoftware = "App_C39"
**.Flow1.DestinationSoftware = "App_C10"
**.Flow1.DataSize = 1000.0MiB
```

Listing 5.5: Definition traffic source behavior in generic *OMNeT++*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<workload>
  <actions id="0">
    <action id="0" transmit="Flow1" waitTime="10.0"/>
    <action id="1" transmit="Flow1" waitTime="10.0"/>
    ...
    <action id="98" transmit="Flow1" waitTime="10.0"/>
    <action id="99" transmit="Flow1" waitTime="10.0"/>
  </actions>
</workload>
```

internal structure of a node. Instead, the behavior is defined in the code of the *node* module.

It is assumed, that any *OMNeT++ node* can act as a traffic forwarder. Its forwarding performance is defined by the respective DNI parameters (`IntermediatePerformance` for `Common Node` and `PerformanceSdnNode` for `SDN Node`) or set as unlimited in case of lacking the respective performance descriptions. The *OMNeT++* equivalent of an `SDN Node` includes descriptions for *software* and *hardwareSwitchingPerformance*, so that the proper behavior can be applied based on the `SdnFlowRule` configuration.

Applications and Traffic Sources

DNI `CommunicatingApplications` are represented in *OMNeT++* under the same name (see, for example, *App_C39* in Fig. 5.23). The deployment of applications and the destinations for the traffic they generate is defined in the configuration file *omnetpp.ini*. A fragment of this file is presented in Listing 5.4.

In the example presented in Listing 5.4, application *App_C39* is deployed on node *C39* and communicates with application *App_C10*. The *App_C39* sends a single type of message that is defined by *Flow1* and has size of 1000MB. The traffic pattern is defined in an xml file *Flow1_Traffic.xml*. An example of traffic generation specification is presented in Listing 5.5.

In the example from Listing 5.5, *App_C39* sends periodically a message defined by *Flow1* and waits 10 seconds before sending the next message. The action is repeated 100 times. The `Workload` descriptions included in DNI are unfolded to

Listing 5.6: A fragment of routing and load balancing configuration in generic *OMNeT++* solver.

```
**.SW35.directions = "Flow2␣SW35_IF1␣1.0"
**.SW40.directions = "Flow1␣SW40_IF2␣0.3␣Flow1␣SW40_IF3␣0.7"
```

represent a single time series for a traffic source. `Workloads` containing `Branch` actions are duplicated and handled separately, so that each branch is represented as a separate *OMNeT++* traffic generator. This allows to keep the *OMNeT++* generator descriptions in the form of a time series containing only transmit and wait events.

Network Configuration

The information about network configuration is used twofold. First, the `NetworkProtocols` and `ProtocolStacks` are used to calculate transmission overheads—the overhead may be calculated, given data header size (e.g., header length of the IP) and an average data unit length (e.g., payload size for Ethernet). The data is represented in DNI in the *dataPayload* and *packetOverhead* parameters. Moreover, the routing information is transformed into the configuration of *OMNeT++*'s node. An example is presented in Listing 5.6.

First line presented in Listing 5.6 defines that all *Flow2* messages on *SW35* node should be forwarded via interface *SW35_IF1*. The second line, on the other hand, defines a load balancing behavior where 30% of *Flow1* messages traversing via node *SW40* are directed via interface *SW40_IF2* and 70% via *SW40_IF3*.

5.4.3 Limitations of *OMNeT++*-based Solvers and their Transformations

In the preceding sections, I presented two *OMNeT++*-based solvers with different characteristics and capabilities. First—*OMNeT++INET*—offers fine-detailed support for such common network protocols as TCP, UDP, and IP. Unfortunately, as INET library does not include any SDN models (at the time of development), the solver cannot support SDN-based scenarios. Load-balancing scenarios are not supported as well. Moreover, the *OMNeT++INET* solver depends on the INET library, so its support for scenarios may change as the library evolves. Additionally, the simulation times may be long as the INET library introduces additional overhead.

The second solver—called *OMNeT++generic*—addresses the weaknesses of the previous one. It supports all network protocols, however at the abstracted, coarser level than in the *OMNeT++INET*. Moreover, it supports SDN-based scenarios as the node model was developed from scratch. Additionally, it supports load balancing scenarios where traffic is split into multiple network paths or directed to multiple receiver nodes. Unfortunately, the behavior of the *OMNeT++generic* solver was custom programmed. This implies that the future version of the

OMNeT++ simulation framework may not support the programed concepts any more and the risk of bugs in the implementation is higher when compared against *OMNeT++INET* that leverages well established simulation library.

The *OMNeT++* solvers, as the heaviest from the simulation frameworks used in this thesis, are expected to deliver long solving times. Their solving performance depends mainly on the number of messages (and events) created in the simulation engine. I compare the features of the solvers and their transformations in Section 5.6.1.

5.5 Layered Queueing Networks: Transformation and Solvers

In this section, I present model transformation to the LQN representation. The LQNs offer analytical solvers to speed-up the solving process. The formalism is briefly introduces in Section 2.3.2. I propose to transform QPN models representing DNI into LQNs as LQNs can be solved using efficient and well established solvers—including two analytical ones. The LQN solvers I consider are: LINE [PC13], LQNS [FMW+09], and LQSIM [FMW+09]. In the following, I describe my contributions to building the *QPN-to-LQN* model transformation.

The validation of the *QPN-to-LQN* transformation was not conducted for QPN models obtained using *DNI-to-QPN* transformation due to the limitations of the transformation and the LQN solvers. The QPN models obtained in the *DNI-to-QPN* transformations are big and have complex structure. Instead, I validate the *QPN-to-LQN* transformation using simpler QPN models. The validation is presented in Section 7.6. The contribution presented in this section was published in [MRSK16].

5.5.1 QPN and LQN Solvers and their Limitations

In this section, I discuss the following four solvers: *SimQPN* [KB06] for QPNs, LINE [PC13], LQNS [FMW+09], and LQSIM for LQNs. I briefly characterize the main known limitations of the solvers in their current versions.

SimQPN [KB06] is a tool for steady-state analysis of QPNs. It is based on discrete-event simulation of a QPN and can yield throughput, utilization and response time statistics as a result (including confidence interval and histograms). Its capabilities are limited by the amount of free memory to a simulation of few millions ($\times 10^6$) of tokens (tokes can be created and destroyed during the analysis) on a commodity hardware.

LINE solver [PC13] leverages the benefits of fluid analysis techniques for solving the LQNs. Currently, its coverage of LQNs is still limited, for example, it does not support the <*and*> node in the activity graphs what limits the set of models that can be solved efficiently. While support for this functionality is planned, no concrete release date is available yet. According to the developers of LINE, the <*or*> node is supported.

LQNS (analytical) and LQSIM (simulation) [FMW+09] are two state-of-the-art solvers for LQNs. The LQNS solver implements an analytical solving technique—mean-value analysis (MVA)—and combines the advantages of other existing solvers, namely SRVN [WNPM95] and the Method of Layers (MOL). According to [FMW+09], LQNS and LQSIM do not support recursive calls (a task calling its own entries) and provide only limited support for replication on subsystems. LQNS cannot handle activity sequences which fork is located in one task and join in another. Moreover, LQNS and has troubles solving models having exclusively external arrival flows.

The analysis of PCM models using QPNs and LQNs has been evaluated by Brosig et al. in [BMB+15]. Compared to LQNS, *SimQPN* was evaluated to provide full support of response time distributions, flexible parameter characterizations, and blocking behavior. On the other hand, the analyzed LQN models were more compact and the solving using LQNS was faster than the solving in *SimQPN* of the respective QPN models.

5.5.2 QPN-to-LQN Transformation

In this section, I present the QPN-to-LQN transformation that enables the fluid analysis for QPN models. The transformation consists of rules that are executed for each matching element of the source model (QPN) and that produces respective elements in the destination model (LQN).

In the simplest case, transformation rules are context-free, injective functions mapping the elements of a single QPN type to the equivalent LQN elements. However, when comparing the two formalisms, one can quickly note that this is not the case for the *QPN-to-LQN* transformation—certain behaviors (e.g., loops, forks, etc.) are explicit model elements in LQNs, while the same behavior is modeled in QPNs using a combination of places and transitions. In order to identify such combinations of places and transitions (in the following, I call this a *pattern*), the transformation rules need to consider the context in which the QPN model elements are used (i.e., identify neighbor elements and understand their roles). An example of such context information may be the neighboring places and transitions or a topology of the QPN. As a result, there may be several, context-sensitive transformation rules that apply to the same model element in a QPN.

To determine which context-sensitive transformation rules shall be used for a certain model element, the structure of the QPN needs to be analyzed first. In the analysis, the transformation searches for known QPN patterns (e.g., loops, forks, joins). In general, graph pattern matching is an NP-complete problem [GJ79], but many efficient pattern matching algorithms exist (e.g., [FWW13]). For discovering patterns, I leverage the fact that any colored Petri net can be unfolded into a single-colored one [LHY12].

The LQN formalism requires to explicitly model the starting point of the calls as top layers. In QPNs, one needs to determine these starting points first, as a net is represented as an arbitrary graph. In order to determine the starting places, the

reachability of places withing the QPN needs to be calculated and open or closed workload places must be identified (e.g., using the approach described in [WSK15]). In case of a closed workload, the cycle around the complete net is temporarily removed from the QPN model and transformed into the LQN as a special top layer that is annotated with a user population. The starting places define the starting point for the search of other patterns. Table 5.4 gives an overview of our transformation rules. The rules are described in detail and accompanied with examples in the following sections.

Table 5.4: Key rules used in the QPN-to-LQN transformation.

QPN element/pattern	LQN representation
Queues	Processors
Queueing places	Task with entry and assigned processor
Ordinary places	Depends on context. See Section 5.5.2.
Token colors	Individual entries for each color in the respective task
Modes of transitions	Activity Graphs for every input color that resemble the mode wiring (see Fig. 5.28)
Fork and join pattern	Fork and join nodes in activity graphs
Loop pattern	Loop notation (see Fig. 5.33)
Critical sections	Critical sections are created by a layer that marks the entrance to the section, has limited resources and uses a processor with a FCFS scheduling strategy.

Queues and Queuing Places

In QPN, I distinguish queueing places and queues. A queueing place consists of a queue and a depository. A queue may be shared between different queueing places. Queues are used to describe scheduling behavior in QPNs (e.g., at hardware resources). In LQNs, the same scheduling behavior can described using processors. The transformation directly maps queues to processors. The associated queueing places are mapped to tasks in the LQN that use the corresponding processor. In case of shared queues (i.e., multiple queueing places referencing a single queue), each queuing place is mapped to separate tasks using the same underlying processor. Figure 5.25 illustrates the mapping for the different cases.

Colors in Places

Tokens in QPN may represent a single request, a resource (e.g., database connection in the pool), or a user. Each token has an associated color. Colors are usually used to model the routing of requests (different colors are traversing different path) or to represent various classes of requests (e.g., separate colors for read and write requests). While colors help to reduce modeling efforts, they do not increase the modeling power of QPNs. Using replication of parts of the net, every colored Petri net can be transformed into non-colored one without loss of information [LHY12, JK09].

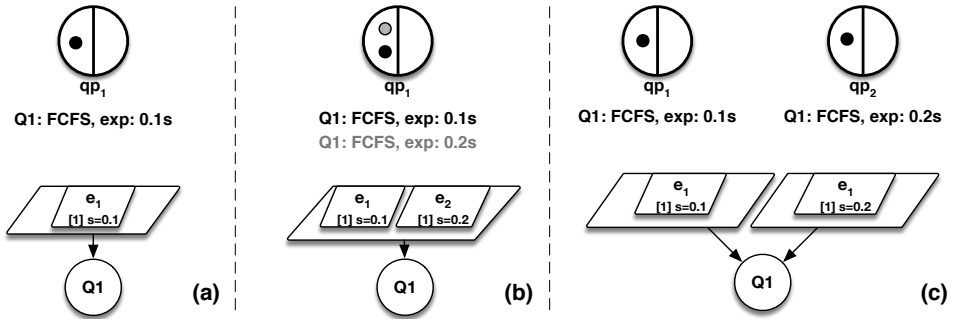


Figure 5.25: Transformation of queueing places with: (a) single place, queue, and color; (b) single place, queue, and two colors; (c) two places, single queue and single color. Processing times are modeled with the exponential distribution with a mean value defined in seconds.

The calls in a LQN are identical and cannot be distinguished by different types (or colors). In order to distinguish different types of calls to a task in LQN, I map each color to a separate entry. An example is presented in Figure 5.25b. In case of queueing places, the entries are parameterized with the service time specified in the QPN. In case of ordinary places, the service time is set to zero.

Ordinary Places

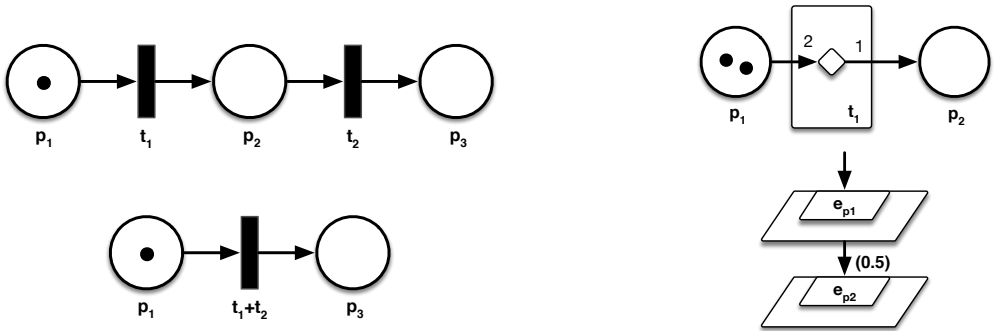
Ordinary places play a specific role in QPNs. They accumulate tokens but have limited influence on the time aspect of the QPN. I transform ordinary places based on the context in which they appear. The following cases are distinguished.

First, an ordinary place is a part of a pattern, for example, a critical section and represents the limited resources (see *pool* place in Fig. 5.34 on page 124). This case is covered by the critical section pattern described in Section 5.5.2.

Second, an ordinary place can be reduced if it does not influence the execution (e.g., it was used only for the convenience of the modeler). It can be reduced—i.e., the neighboring transitions can be merged—only if the place is the only successor of the preceding transition and the only predecessor of the succeeding transition. An example is depicted in Figure 5.26a.

Third, an ordinary place can be used also as a synchronization point. This happens when a succeeding transition consumes multiple tokens and the tokens are held in the ordinary place until the required amount is deposited. According to LQNS documentation [FMW+09], LQN supports this case using the *calls-mean* parameter that can be specified as a real variable. An ordinary place followed by a transition that consumes n and produces m tokens will result in *calls-mean*$= \frac{m}{n}$ in LQN. An example is depicted in Figure 5.26b.

Finally, an ordinary place (the same applies to a queueing place) can precede a branch where the deposited token is consumed by one of the succeeding transitions. I depict it in Figure 5.27, where the token in place p_1 has equal probability $= 0.5$



(a) Redundant place reduction.

(b) Reduction of tokens.

Figure 5.26: Transformation of QPN ordinary places depending on context.

to be consumed by transition t_1 or t_2. The probabilities can be calculated based on the firing priorities of the transitions—by default all transitions have equal priority.

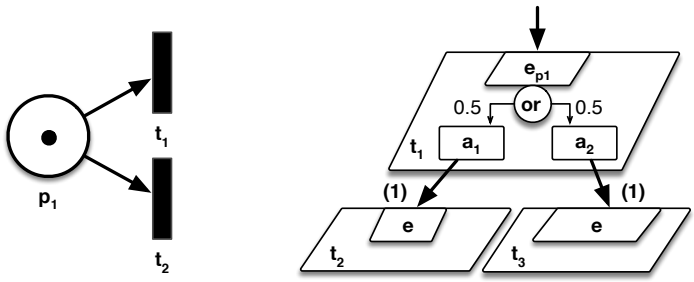


Figure 5.27: Transformation of QPN ordinary places depending on context: branch in workload.

Transitions and Modes

In QPNs, transitions consume tokens from incoming places and produces new tokens in outgoing places. Transitions can fire in different modes to model various dynamic behaviors. The incidence function defines the number and color of tokens consumed and produced by a firing mode. Figure 5.28 depicts the possible transition configurations and their LQN equivalents respectively.

Multiple incoming places connected to the same mode represent a synchronization point or a "join" (Fig. 5.28a). Transitions containing multiple modes can be decomposed into multiple transitions, each containing a single mode. As a result, they can be treated as independent calls to the same entry of a task (Fig. 5.28b). Multiple outgoing places from the same mode represent a fork (Fig. 5.28c). The

transformation maps the transitions to LQN activity graphs, where Fork and joins are represented by <*and*> nodes (depicted as &) (Fig. 5.28a and 5.28d).

Fork and Join Pattern

The fork and join pattern in QPNs is built by defining a mode in a transition that consumes a token and forwards the token to multiple succeeding places. I present a simple fork-join pattern in QPN in Figure 5.29. In LQNs, forks are modeled with activity graphs. The <*and*> nodes are used to execute calls in parallel and to join (synchronize) them after they are finished. In Figure 5.30, I present the transformed LQN equivalent model of the QPN presented in Figure 5.29.

Finding the start and end of forking process is challenging. While the start (the fork) is marked by a mode consuming a token and depositing multiple tokens, each in possibly separate place, the matching end (the join) must be found using a graph searching methods. Since colors can change on the way through the graph it is non-trivial to match a fork with the respective join.

To address this problem, I envision the following possibilities. First, one can try to fit the fork-join pattern in a single LQN task, so that more solvers can be used to solve such model (see solvers limitations in Section 5.5.1). The analysis of non-trivial fork-join patterns in QPN (e.g., with colors changing between fork and join) is conducted using algorithms for graph analysis (e.g., [RS10]).

Second, one may omit the search for matching forks and joins and proceed to the further transformation rules. In this way, the fork and join pair may be separated and placed on different tasks. Although this limits the compatible set of solvers (e.g., LQNS does not support separated fork-join), the model will be transformed correctly.

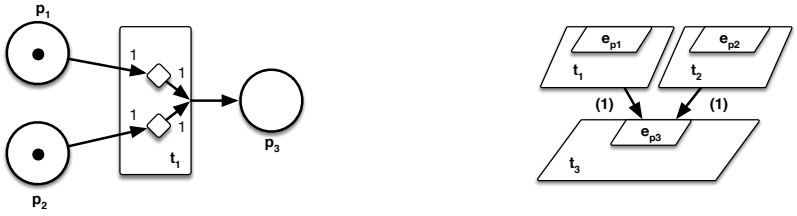
Loop Pattern

The QPN formalism does not support modeling of loops directly (there exists no QPN loop element) but a loop can be modeled indirectly using multiple simpler QPN elements. Examples of QPN loops modeled indirectly are presented in Figures 5.31 and 5.32.

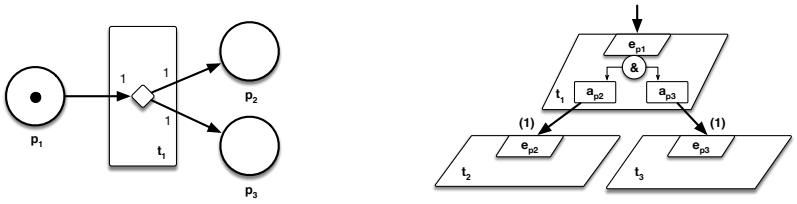
The loop presented in Figure 5.31 iterates based on the probability defined in the incidence function of the *Loop-Exit* transition. The expected number of iterations needs to be calculated in the transformation, as LQN requires exact number of iterations to be specified. In Figure 5.32, the number of iterations is defined deterministically by the number of tokens produced by the *1-to-num-loop-iter* transition. LQN supports loops directly, so once the loop pattern is recognized correctly and the number of iterations is calculated, the transformation rule is trivial. Graphical representation of an equivalent LQN loop is shown in Figure 5.33.

The QPN representations of loops are treated as patterns that need to be discovered by the transformation (or a separate QPN analysis library) in order to be transformed. In case of an unsupported loop pattern (there may exist other patterns than the two presented in Fig. 5.31 and 5.32), the transformation of a

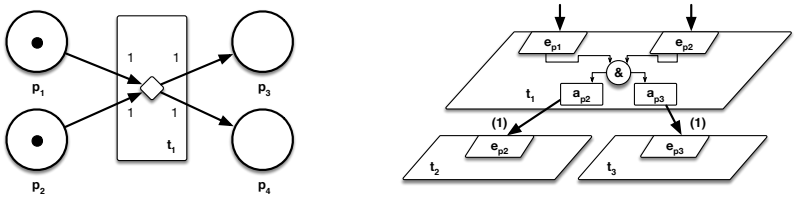
(a) Join-and.



(b) Join-or.



(c) Fork.



(d) Join-fork.

Figure 5.28: Transformation of QPN transitions. QPN transition t_1 contain modes that consume and produce tokens on fire. LQNs representation is simplified (no processors) for brevity.

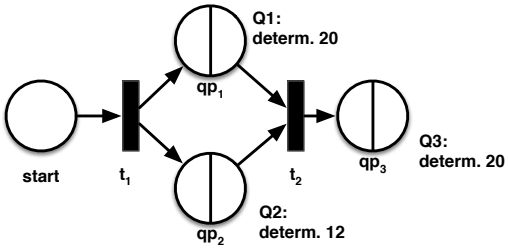


Figure 5.29: Exemplary QPN containing the fork and join pattern.

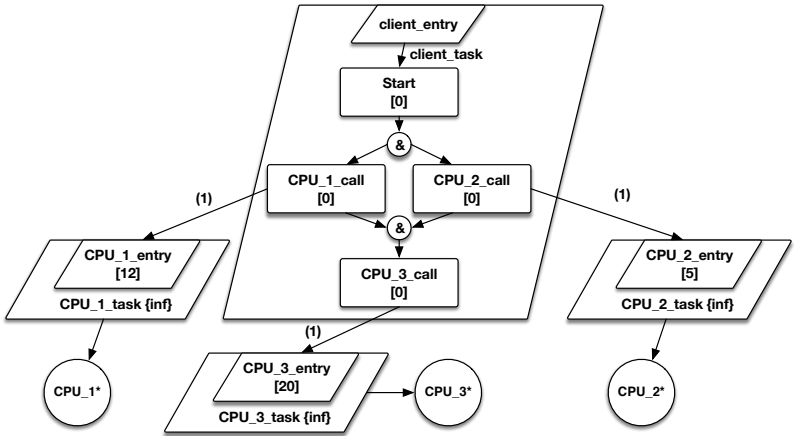


Figure 5.30: Exemplary LQN containing the fork and join representation of the QPN shown in Figure 5.29.

loop may be covered by the remaining transformation rules (depending on how the loop was modeled in QPN), however, the compact notation of LQN loop (as in Fig. 5.33) will not be used.

Critical Section Pattern

A critical section is a region which can simultaneously handle only limited number of objects. Both LQN and QPN can model critical sections. Figure 5.34 shows a critical section in QPN. It is modeled with the *enter section* transition that consumes a token from the *start* and second from the *pool* place. The amount of initial tokens in the pool defines the number of tokens that enter the section at the same time. At the end, the *leave section* transition passes the token further to the *end* place and at the same time deposits another token back into the pool, so that the next token from *start* can enter the section.

LQNs represents a critical section with a layer that contains a defined number of first come first serve (FCFS) queues. The number of FCFS queues in LQN

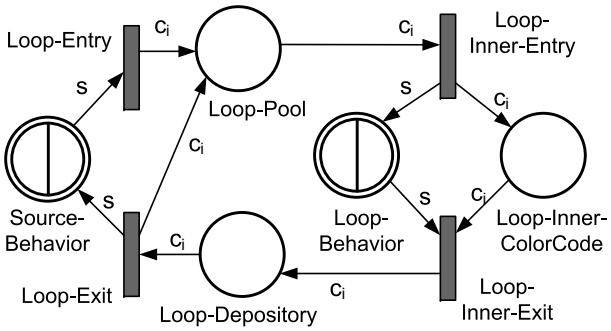


Figure 5.31: Example of a QPN loop representation with probabilistically modeled number of iterations. Excerpted from [BMB+15].

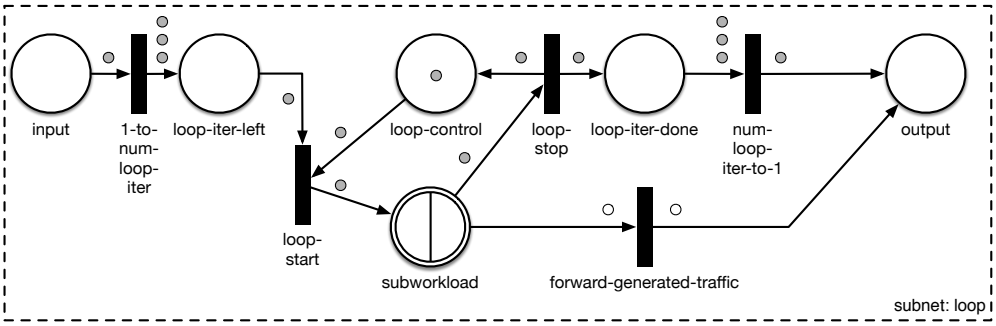


Figure 5.32: Example of a QPN loop representation with deterministically modeled number of iterations (repetition of Fig. 5.17).

corresponds to the QPN's pool tokens that limit the maximum number of tokens in the critical section. Every task in every queue will execute a synchronous call to perform the work in the critical section. Only when this call finishes, the next element will be dequeued and processed. Graphically, I depict LQN critical section in Figure 5.35. The size of the pool is denoted with the quantity of the task *critical_section[3]*.

5.5.3 Transformation Limitations

In contrast to previously presented model transformations, the *QPN-to-LQN* transformation consumes QPN models at the input. Unfortunately, the *QPN-to-LQN* transformation does not fully support all QPN models due to its limitations. The DNI models may be solved using LQN only if the DNI-to-QPN or miniDNI-to-QPN transformations produce a QPN model that is compatible with the *QPN-to-LQN* transformation. If the *DNI-to-QPN* transformation results in an incompatible QPN model, the *QPN-to-LQN* transformation will fail and the LQN solvers cannot be

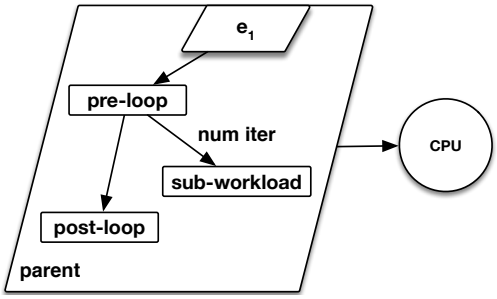


Figure 5.33: LQN loop representation with deterministically modeled number of iterations.

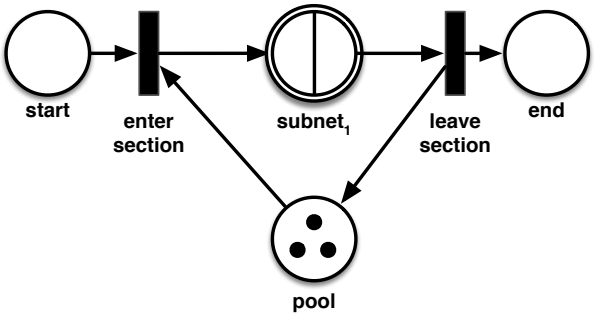


Figure 5.34: Example QPN containing a critical section. The pool contains, so maximally three tokens can enter the subnet.

used. Unfortunately, there is no guarantee that the QPN representation of a DNI model will be compatible with LQN. The partial support of QPN models in the *QPN-to-LQN* transformation is caused by the nature of both meta-models. QPN formalism is more general than LQN so QPN may represent more scenarios than the LQN. Due to that, some QPN scenarios cannot be represented by the LQN formalism what makes the transformation of many cases impossible. The *QPN-to-LQN* transformation has been validated in Section 7.6 using compatible QPN models. The resulting LQNs were solved using three LQN solvers: LQSIM [FMW+09], LQNS [FMW+09], and LINE [PC13].

In the remainder of this section, I describe the limitations of the *QPN-to-LQN* transformation. This section covers general limitations of the LQN formalism and do not focus on solver-specific limitations (the limitations of LQN and QPN solvers are presented in Section 5.5.1). The most challenging parts of the transformation are: loops where a higher layer in LQN needs to be called, and the problem of finding the top layers (also called reference layers). I discuss both problems in the following.

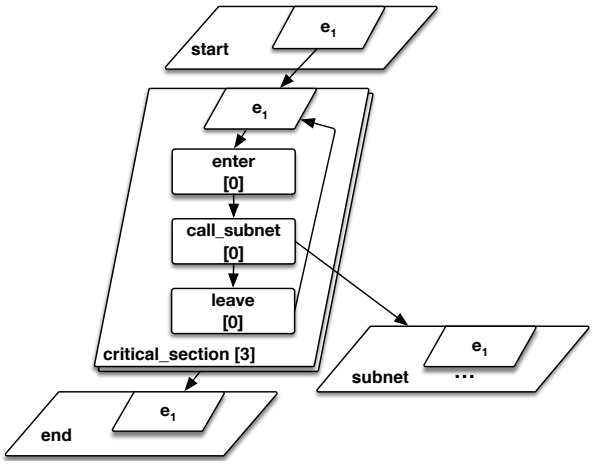


Figure 5.35: LQN representation of the critical section corresponding to the QPN in Figure 5.34.

Support for Specific Loops

Currently LQN supports a *loop* node, which executes a defined number of loop iterations assuming that the number of loop iterations is known beforehand. Unfortunately, it is impossible to build a LQN model of a QPN loop with unknown number of iterations, for example, as seen in Figure 5.36. This limitation stems form the lack of support of LQN to call layers that lie higher in the hierarchy. Solvers like LINE will run into recursion problems (exceeding the maximum depth). The general loop that models the closed workload of a complete LQN model is a special case and LQN supports it even if the number of iterations is infinite or unknown.

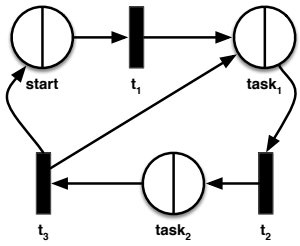


Figure 5.36: Example of QPN containing a second internal loop.

Finding the Top Layer

LQN use a special layer (called reference or top layer) to start the workload cycle. The task in the top layer will be executed periodically according to the think time parameter. QPN does not necessarily have obvious starting places. Finding the transitions that fire first or model the think time of the closed workload is challenging. In order to address this limitation, I propose to analyze the input QPN and estimate which transitions will fire as first. I aim to find the transition that models the beginning of a closed workload. Unfortunately, QPN allows to represent a system in many ways and it is not guaranteed that the transitions found are responsible for representing the think time of a closed workload loop. An approach to this problem was tackled by Walter et al. in [WSK15].

In the transformation, for each transition found a top layer is built. The top layer is treated specially in LQN as it is a starting point. Assuming that a top layer is found, I construct LQN tasks that succeed the top layers by traversing the subsequent QPN elements starting with the places that are successors of the first-firing transition.

5.6 Selection of Optimal Solver

Model transformations presented in this chapter generate various predictive models. The transformations may support only selected DNI models, based on the features represented in the DNI model (e.g., SDN-specific entities). In Section 5.6.1, I analyze the semantic gaps between the solvers, transformations that generate the predictive models, and formalisms that are used for a predictive model. The analysis serves as a basis to recommending a feasible solver for the user, based on hers requirements. Next, in Section 5.6.2, I propose a procedure for recommending the most feasible transformations and solvers for a given prediction scenario.

5.6.1 Differences between Predictive Models and Solvers

In the presented approach, DNI serves as the information source to automatically generate multiple predictive models out of a single descriptive model. In this section, I compare the predictive models by describing their performance-relevant network capabilities and limitations. A selection of features and metrics of the predictive models is presented in Table 5.5.

I characterize selected features that are supported by the transformation and the solver itself. Some features are supported partially (symbol $\oplus$) when the solver coarsely approximates the real behavior. On the other hand, if a real feature can be supported via extension of the transformation or the solver (e.g., for *OMNeT++* solvers), the feature is marked as $\odot$. The mark $\odot$ implies that the current version of the solver or transformation do not support the feature or metric. The minus symbol denotes that the feature is not supported and the support cannot be added without reprogramming of the solver. The question mark denotes that a feature is not supported and some data is missing to judge if the feature can be supported.

Table 5.5: Selected features of the predictive models resulting from model transformations.

Feature	*OMNeT++* INET	QPN (DNI)	QPN (mDNI)	*OMNeT++* generic	LQN (QPN mDNI)	LQN (QPN DNI)
Intermediate and end nodes	+	+	−	+	−	?
Node virtualization	+	+	−	+	−	?
SDN support	×	+	⊙	+	?	?
Traffic patterns	+	+	×	⊕	×	×
Packet-level traffic generation	+	×	×	+	×	×
TCP/IP	+	×	×	×	×	×
UDP/IP	+	+	+	+	−	−
Generic protocol	−	+	+	+	+	+
Load balancing	×	+	×	+	⊙	⊙
Performance metrics						
Throughput time series	+	×	×	+	×	×
Throughput distribution	+	⊕	⊕	+	⊕	⊕
End-to-End delay time series	⊙	×	×	+	×	×
End-to-End delay distribution	⊙	⊙	⊙	⊙	⊙	⊙

+ support, ⊕ partial support, ⊙ extension possible, − support in an abstracted form, × no support, ? no data (assume: no support).

The `IPosition` of the DNI's `Node` allows the modeler to distinguish , among others, network switches from the application servers. The `IPosition` of a `Node` is not abstracted in both transformations using *OMNeT++* and in *DNI-to-QPN*. The transformation *DNI-to-miniDNI* abstracts a `Node` with a generic entity, so the solver produced by *mDNI-to-QPN* transformation does not recognize whether a `Node` is `End` or `Intermediate`. Similar holds for LQN models, whereas the LQN model obtained from QPN and DNI is not fully supported by the *QPN-to-LQN* transformation, so no statement can be made.

Analogously, the modeler cannot distinguish a VM from a server (except of its name) when looking at the QPN model obtained from miniDNI and the respective LQN model. Only the *DNI→ QPN→ LQN* model lacks the information about the support of node virtualization due to the limitations of the transformation.

SDN-based scenarios are fully supported by the *DNI→ QPN* and the *OMNeT++generic* models. The *OMNeT++INET* solver cannot simulate scenarios with SDN elements. The model *mDNI→QPN* can represent SDN scenarios in an abstracted manner, yet the current form of the *mDNI→QPN* transformation cannot. There is insufficient data available regarding the LQN solvers, because the *QPN→LQN* transformation cannot cope with the QPN models obtained in the *DNI→QPN* and *mDNI→QPN* transformations.

Distinguishing traffic patterns (e.g., sudden load spikes, seasonal patterns—as specified in [vKHK14]) is supported by the *OMNeT++INET* and the QPN model obtained in *DNI→ QPN* transformation. The *OMNeT++generic* transformation simplifies the traffic profile so only coarse traffic patterns are supported (e.g., load spikes lasting several minutes). Rest of the models represent the traffic more coarse and flatten the traffic characteristic.

Only *OMNeT++* models are able to represent packet-level behavior of the generators. *OMNeT++INET* model describes traffic generator at the messages level (e.g., transmit a 5MB picture from A to B), but the simulation reproduces the packetization as in the real network. The *OMNeT++generic* model can be configured to represent traffic at the packet level, however by default, the model represents the traffic with messages. The other predictive models support the granularity of a message and the packet-level generation is unavailable.

The behavior of a TCP-based network differs from other protocols. TCP can modulate the network performance by instructing a traffic source to send messages slower if there is a congestion in the network. Moreover, TCP guarantees message delivery by retransmitting lost messages or their fragments. To model this behavior, few TCP algorithms need to be present in the predictive model (e.g., slow start, TCP Vegas). Only *OMNeT++INET* supports such behavior. The other predictive models cannot reproduce such behavior and offer UDP-similar transmission.

The UDP transmission is allows packets to be dropped if a network is congested. This is directly modeled in *OMNeT++INET*, whereas the *OMNeT++generic* model and both QPN models drop packets only if a queue is overloaded and the solving may fail (e.g., due to solver's high memory consumption). LQN models do not drop "calls" (the unit of flow in the model), so their behavior abstracts the real UDP behavior (UDP may transfer all packets without drops as well).

Finally, all models except of *OMNeT++INET* support a generic protocol—an abstraction that allows to coarsely model any network protocol. The *OMNeT++INET* requires a protocol to be selected from a predefined set (TPC+IP or UDP+IP) to run a simulation.

Load balancing scenarios can be solved with QPN solvers and with the *OMNeT++generic* solver. Despite the support of load balancing mechanism in LQN formalism, the *QPN-to-LQN* transformations in its current version is unable to transform such scenarios.

All performance models analyze throughput as the main performance metric, however, only *OMNeT++*-based solvers provide detailed throughput values for each moment of the simulation; the other models provide aggregated statistics. Despite the different granularity of traffic modeling (packet-level in *OMNeT++* versus message-level in QPNs), analysis of the end-to-end transmission delay is possible in both formalisms. The necessary extensions are available, however, the calibration and evaluation of the delay metrics is considered as a future work.

5.6.2 Optimal Solvers for Performance Prediction

The selection of an optimal solver for a given prediction scenario depends on the following user-given constraints:

1. time frame in which the performance prediction results need to be delivered,
2. accuracy of the performance prediction,
3. required performance metric and resolution of the result,
4. model-related factors—e.g., model size, support of features (e.g.: SDN, custom protocol, load balancing).

In the following sections, I discuss the constraints and propose an approach to selecting the optimal solver for performance prediction. The detailed evaluation of the solving time is conducted in Section 7.4.

Solving Time

The users usually require to obtain a result of performance analysis in a given time frame. This factor plays important role especially for run-time performance prediction scenarios where the performance predictions are used to optimize system's configuration in reaction to an event. The performance of the performance prediction process is challenging to predict and depends on multiple factors, for example:

1. size of the input model,
2. scalability of the solver,
3. performance of the transformation program,
4. required accuracy or size of the confidence interval for a given metric.

The user may however, estimate the solving time given the information about the transformation architecture and the overall performance of an solver. For example, the solvers that simulate the network traffic on a packet level require to process more events than the solvers that simulate messages as the units of flow.

A source model that includes less information, in the most cases, will usually result in a less complex predictive model. For example, the solvers transformed from a *miniDNI* model can be solved faster than the solvers transformed directly from the *DNI* model, as the *miniDNI* meta-model abstracts information from a *DNI* model.

Next, the architecture of a solver itself influences the solving time. Multi-threaded simulation solvers perform in general no worse than single threaded. Some solvers handle larger model better as they approximate the models with analytic formulas and the approximation is better for larger systems or more intensive workloads (e.g., LINE [PC13]).

Before a solver can solve a predictive model, a transformation needs to build the predictive model. Model transformations are applications which run times depends on the size and the complexity of the input model. Moreover, to conduct a sensitivity analysis (e.g., how performance of the network changes for various values of a parameter) multiple predictive models need to be generated thus, a transformation needs to be executed multiple times.

In the context of this thesis, *by no means a solver can guarantee a constant maximal solving time*. The detailed analysis of the solving time is conducted in Section 7.4. There, I show the usual solving times for typical models. Additionally, for selected solvers, I conduct solving time scalability analysis, that is, how the solving time changes depending on the input model size.

Prediction Accuracy

Each solver handles the user-required accuracy according its own internal mechanisms. For example, *SimQPN* repeats the solving until the steady-state is reached and the user-given stopping criterion is satisfied. On the other hand, the *OMNeT++*-based simulators do not provide means to increase prediction accuracy without changing the model or increasing the number of simulation repetitions. Neither they guarantee to reach steady-state if a simulated scenario is too short. Moreover, the user is required to conduct multiple simulation runs to meet the requirements regarding confidence intervals of the result.

The solving time is usually connected with prediction accuracy, so solvers providing more accurate performance predictions run longer than other less accurate solvers. The dependency between solving time and the delivered accuracy cannot be described with a defined dependency (e.g., linear or exponential) as it depends on the complexity of the input model and the solver itself.

Performance Metric

By selecting an optimal solver for a given scenario, the user needs to consider the performance metrics that a solver delivers. In this thesis, I focus mainly on the network capacity and network interface throughput, however many solvers measure other metrics as specified in Table 5.5.

Model-Related Factors

To conduct performance analysis, a DNI model needs to be transformed to a predictive model and then solved. This processing chain causes that selected models are not supported due to the limitations of the transformations and solvers. For each model—based on the scenario that it models—selected model transformations and solvers may return an error as the features included in the model may not supported. Based on Table 5.5, I propose a procedure for selecting the feasible solvers for a DNI model and scenario.

First, I define a set of solvers for which a model can be build using the currently existing set of transformations. Let S denote a vector containing real values defined as:

$$S = [s_1, s_2, \cdots, s_6]^T \tag{5.1}$$

where, s_n denotes the feasibility of an n-th solver for the given scenario. The solvers are enumerated as follows:

s_1: *OMNeT++INET*,

s_2: QPN DNI,
s_3: QPN minDNI,
s_4: *OMNeT++generic*,
s_5: LQN QPN DNI,
s_6: LQN QPN miniDNI.

I treat the three LQN solvers (LQNS, LQSIM, LINE) as one and assume that the *QPN-to-LQN* transformation supports any QPN model as an input. The limitations of this transformation (without this assumption) are described in Section 5.5.3.

To evaluate the solvers I assume, that each solver s_n is evaluated with a positive real number $s_n = [0, \infty)$, where 0 denotes the best fit and higher number less fit for the scenario. The infeasibility of a solver for a given scenario is modeled with infinity $s_n = \infty$. The user shall prefer solvers with lower score. Solvers with non-zero score solve the model as well but they may treat the user input model in and abstract way and omit or simplify some information. The solvers with infinity score will return error during the transformation phase.

The procedure for joint evaluation of the solvers and transformations is presented in Algorithm 2. The scoring used in the algorithm is based on Table 5.5. Whenever the table defines no support for a feature (symbol $\times$), the score is set to infinity. Abstracted support for a feature (symbol $-$) adds 100 to the evaluation score, partial support (symbol $\oplus$) adds 50. Full support (symbol $+$) adds nothing to the score. The features that are possible with an extension (symbol $\odot$) are assumed to exist.

Algorithm 2 A function evaluating feasibility of solvers and model transformations based on features included in the input DNI model.

function EvaluateSolversFeasibility
  $S = [0, 0, 0, 0, 0, 0]^T$
  if required load balancing **then**
    $s_1, s_3 = \infty$; $s_2, s_4, s_5, s_6 = 0$; ▷ s_5, s_6 Implementation required
  if required protocols other than 'TCP', 'UDP', 'IP' **then**
    $s_1 = \infty$
  if required protocol 'UDP' **then**
    $s_5, s_6 += 100$;
  if required strict behavior of protocol 'TCP' **then**
    $s_2, s_3, s_4, s_5, s_6 += \infty$;
  if required packet-level granularity **then**
    $s_2, s_3, s_5, s_6 += \infty$;
  if required traffic patterns **then**
    $s_4 += 50$; $s_3, s_5, s_6 += \infty$;
  if required SDN support **then**
    $s_1 += \infty$; $s_3, s_5, s_6 += 100$; ▷ s_5, s_6 more data required
  if required strict distinction of virtual nodes **then**
    $s_3, s_5 += 100$; $s_6 += 100$; ▷ s_5, s_6 more data required
  if required strict distinction of end and intermediate nodes **then**
    $s_3, s_5 += 100$; $s_6 += 100$; ▷ s_5, s_6 more data required
  return S

Algorithm 2 calculates the rating of six solvers based on nine questions. Each question should be answered as *yes* or *no* by the user to calculate the complete rating. The final score is calculated as a numeric value. The score 0 means the best fit of a solver to the model, score ∞ means no support.

Three questions concern "strict" requirements: TCP behavior, distinction of physical and virtual nodes, and distinction of end and intermediate nodes. The strictness is defined to stress the requirements of the user, for example, the user wishes no TCP behavior approximation using another protocol. Strict distinction of virtual and physical nodes means that the virtual and physical `Nodes` are modeled using different entities in the predictive model. The similar hold for `End` and `Intermediate Nodes`.

LQN Solvers (s_5 and s_6) support only selected input models due to incomplete transformation implementation. Due to that, the complete feasibility analysis cannot be conducted. Four questions in Algorithm 2 are answered assuming the required support is available in an abstracted form. The four cases are annotated with a comment that more data is required to conduct full analysis.

In Section 7.4, I conduct extended feasibility analysis that includes the solving times and resource consumption of the solvers.

5.7 Summary

In this chapter, I presented the approach to solving DNI models based on model transformations. I described six model-to-model transformations and presented ten possible ways of solving a DNI model.

In Section 5.1, I presented the pre-transformation steps including model validations, transformation parametrization, and *in-place DNI-to-DNI* transformations. The model validation rules guarantee that the user-build DNI model is valid and can be transformed by selected transformations (depending on the support for the network features modeled in DNI). Ambiguous model settings are concertized using transformation parametrization, so that the user-built DNI model is transformed according to the user's requirements. Next, the input DNI model is transformed in-place to translate the user-friendly DNI format into a DNI form that is easier to transform by the transformation developers. Afterwards, a valid DNI model can be transformed into predictive models using model transformations or their combinations.

The contributed set of model transformation enables a DNI model to be transformed into six predictive models (DNI-QPN, mDNI-QPN, DNI-QPN-LQN, mDNI-QPN-LQN, DNI-*OMNeT++INET*, and DNI-*OMNeT++generic*) that can be currently solved using the six available solvers: one for QPN, *OMNeT++INET*, *OMNeT++generic*, and three for LQN. These contributions allow to automatically generate up to ten various methods for performance prediction for a single DNI model (depending on the features included in the DNI model).

Finally, in Section 5.6, I characterized the differences and semantic gaps between the predictive models built using the transformations. I specified the limitations

of the transformations and solvers taking into consideration the network features modeled in an input DNI model. Additionally, I proposed an algorithm for evaluating the feasibility of a solver depending on the network features modeled in DNI.

In Chapter 7, I validate the approach to performance prediction using the predictive models generated using the model transformations described in this chapter.

Chapter 6

Extraction and Calibration of the DNI Models

Manual building of Descartes Network Infrastructure (DNI) models may be cumbersome if the modeled system (including the workloads) is complex. Fortunately, the major part of a DNI model may be extracted automatically or semi-automatically based on the data gathered from the running system. Such automatically prefilled models may be later manually fine-tuned by the human modeler.

In this chapter, I discuss possible methods for DNI model extraction as a secondary research contribution of this thesis. In Section 6.1, I present conceptions for extraction of network structure, selected hardware parameters, and network configuration. In Section 6.2, I provide an approach to semi-automated extraction of traffic models based on the traffic traces captured from the running system. Finally, in Section 6.3, I discuss the approaches to model calibration, which tune the model to represent the modeled system more reliably, so that the performance accuracy may be improved.

6.1 DNI Model Extraction

DNI models can be built manually using standard tooling, for example, Eclipse Eclipse Modeling Framework (EMF), or an automatically generated text editor (using e.g., Xtext, or Human Usable Textual Notation (HUTN)). To build a DNI model manually, a modeler needs to include information about the network: topology, configuration and traffic. Spinner [SWK16] states that "performance models can provide many benefits, their manual creation and maintenance is time-consuming and expensive, severely limiting their usage in real-world systems". Additionally, the incentive for model extraction is even stronger for highly dynamic systems because they may reconfigure them selves autonomously during their operation and thus manual extraction may be infeasible. Fortunately, major part of the information required by DNI can be extracted automatically allowing to generate an initial model populated with the relevant data. I propose the following detailed objectives for DNI model extraction:

(a) automatically extract information about the network topology, addresses, and interface names based on data from network configuration files;

(b) automatically extract information about the deployment of software on network nodes based on data provided by server operating systems;
(c) automatically extract information about the configuration of switches and routers based on data obtained through the OpenFlow protocol (for Software-Defined Networking (SDN) devices) or Simple Network Monitoring Protocol (SNMP) for non-SDN devices;
(d) semi-automatic extraction of workload profiles based on techniques for capturing and analyzing traffic traces;
(e) semi-automatic extraction of selected performance characteristics based on performance monitoring data (e.g., the statistics collected by OpenFlow or SNMP).

The accuracy and the amount of extracted information defined in the objectives a) to e) will vary depending on the logical access to the devices in the data center, their support for monitoring protocols, and the security policies used in the data center. Assuming that the data center is managed by a single operator with full access to the devices and monitoring data, I envision the extraction methods that are conceptually described in the following.

Topology can be extracted based on multiple complimentary sources of information. In SDN-based networks, the SDN controller can provide the information about physical topology including the basic data regarding network interfaces: the known nodes, and their addresses. For a non-SDN-based network, the following information sources may be used: traffic traces, SNMP data, routing/switching tables, and internal data from network node operating systems.

End-nodes (e.g., servers, virtual machines (VMs)) that do not produce traffic can be extracted using simple agents that would need to be installed on each node; the intermediate nodes (e.g., routers, switches) could be partially discovered based on host Address Resolution Protocol (ARP) tables or Link Layer Topology Discovery (LLTD). For networks based on non-IP and non-Ethernet protocols, only end-node agents and SDN controllers will supply the topology information. In the worst-case, a network administrator should provide the topology specification manually.

In this thesis, I assume, that the data center operator has full access to the infrastructure and information about physical topology is accessible. In some cases, however, the physical network topology may be invisible to the tools and protocols. This applies mainly to virtual networks (e.g., based on tunneling) that abstract the physical topology by provide a virtual one instead. Models extracted in such way are also valid for performance prediction purposes, however in such cases, the vendor-provided hardware performance specifications cannot be used directly to parametrize the model.

Software deployment can be extracted based on traffic traces and the mapping of network addresses (Media Access Control (MAC) or Internet Protocol (IP)) to the nodes. In case the topology cannot be extracted successfully (or the addresses

are unavailable), the manual specification of the software deployment is relatively simple: for each software application a single node needs to be assigned. In the worst-case, a system administrator must specify the mapping between nodes and software applications manually.

Automatic extraction of software deployment may face several challenges. Traffic sources extracted based on traffic traces may represent different software structure than in the reality. For example, an application composed of multiple modules may not be recognized as one but as multiple separate applications. Despite possible deviations, the extracted model will correctly represent that part of the system from the performance point of view and the performance analysis shall not be affected.

Network configuration is relatively simple to extract automatically as most of the data can be provided, for example, by switching/routing tables, SDN controllers, or SNMP monitoring tools. Additionally, many hardware vendors provide ready solutions for network monitoring and configuration so the relevant data may be read from a running system. Due to the medium-granular structure of DNI, the required network configuration data (i.e., routing, protocol overheads, and SDN flow rules) pose no major extraction challenges.

Performance descriptions capture information closely related to the performance of a given node, network interface, or a link. Some of that information may be found in a technical specification of a given device (e.g., switching delays of a switch, or the maximal throughput of a cable), whereas some of it may need to be measured experimentally. Assuming that no technical information is provided (or the vendor-provided data is inaccurate), the performance characteristics can be extracted from network monitoring data using resource-demand estimation techniques [SCBK15b].

Workload and traffic profiles. Modeling of the workloads in a system during the run-time is the most challenging part of extraction. The challenges are mainly caused by non-deterministic behavior of the users who use the modeled system. In practice, traffic traces need to be captured in order to reliably represent a particular network traffic workload. Unfortunately, the traffic traces may be to big to model directly as the traces need to be captured on all network interfaces of the nodes. To address these challenges, I propose a flexible approach to network traffic model extraction based on the traffic traces. The approach is described in Section 6.2.

6.2 Traffic Model Extraction

Traffic profiles are usually difficult to extract manually due to the amount of data transmitted over the network. Fortunately, traffic models can be extracted from captured traffic traces. The traces can be captured on the monitoring port of a

switch (a port that aggregates and mirrors the traffic from all ports) or at the network interfaces of the servers producing traffic.

In the proposed approach, the traffic profiles are represented as time-series that are build based on the captured traces. For each traffic source, the traffic profile captures only model-relevant data (simplified time series) without any payloads. The traffic profiles are represented the original time series in a simplified and compressed form. An optimization method is proposed to assure compactness and representativeness of the extracted traffic profiles. The method ensures balance between the size and the level of detail of the extracted profiles.

The approach to traffic model extraction was developed in cooperation with Viliam Šimko and published in [RSS+16].

6.2.1 Network Traffic Generator Model

In the proposed approach, the extraction of a network traffic model is divided into two coarse steps: (a) extraction and optimization of the traffic generators, and (b) transformation of the traffic generators into an instance of the DNI meta-model. The first step is tailored to simplify the second step and to provide the intermediate format for the traffic model: a generator model (described in Section 6.2.1). The output of the first extraction step is not bound to the DNI model and can be used for other models as well.

Before I explain the extraction pipeline and signal decomposition, it is necessary to summarize important assumptions about the network traffic. Typically, in a network traffic dump, one can identify multiple communicating parties based on IP addresses, ports or other parameters. For example, inside the communication from a single server I can identify two applications by the Transmission Control Protocol (TCP) port, albeit the IP address is always the same. Here, I assume that a pre-processing step takes place to isolate individual sessions that are of the interest of the modeler. I assume that the network traffic is represented as a univariate time-series with one-second time granularity and positive values (packets with negative size do not make sense).

In reality, I may observe regular and periodic data transfers but also irregular and highly asymmetric signals depending on the application. Therefore, the decomposition algorithm has to handle irregular signals well and to take advantage of regularities whenever possible.

The last assumption in the approach concerns the DNI model and the simulation models that can be automatically generated from a DNI model. Network traffic in simulators generated by DNI (e.g., *SimQPN* [SKM12a] or *OMNeT++*) has to be modeled as a collection of traffic generators emitting packets with a certain frequency. The information about packets emitted is encoded into DNI *flows* that are later transformed into entities used in the respective simulation (e.g., packets in *OMNeT++* or tokens in *SimQPN*). There is not much room for implementing arbitrary real-value functions, which also hinders the application of Fourier and Wavelet transforms as explained further in Section 6.2.3.

Simple Traffic Generator Model

The extracted network traffic is stored in an intermediate format that represents a collection of *simple traffic generators*. I assume the model of a simple traffic generator as depicted in Figure 6.1.

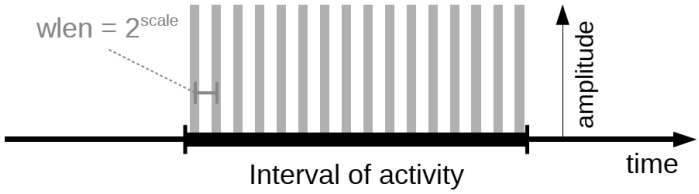


Figure 6.1: Model of a simple traffic generator.

A simple traffic generator is a tuple:

$$Generator = (scale, amplitude, begin, end)$$

where the parameters are defined as follows:

- **scale**: how frequently the generator emits packets; every $wlen = 2^{scale}$ seconds,
- **amplitude**: size of the packet to be emitted,
- **begin**: beginning of the interval when the generator is active,
- **end**: end of the activity interval.

The extracted set of generators is saved in a text format for further processing. I parse the generator descriptions and build a DNI model but the data can be also used for other purposes (e.g., protocol debugging). Figure 6.2 demonstrates the idea behind the decomposition into activity of simple traffic generators and the corresponding fragment of the DNI model.

Examples of Network Traffic

In Figure 6.3, I present four examples of a recorded network traffic. The first time-series represents data transfer during 60 minutes of video streaming with occasional caching. The second example represents an irregular data transfer with a heat-up phase containing several small data transfers, followed by a continuous transfer reaching the link capacity. The third example represents a signal with several outliers—an example of multiple applications transferring data in parallel. The fourth time-series represents a regular signal.

Clustering of the traffic

I observe that the traffic rates can be efficiently clustered because an application usually transfers data in packets of certain sizes. This can be seen in Figure 6.4. I plotted the kernel density plot for each of the exemplary traffic dumps showing

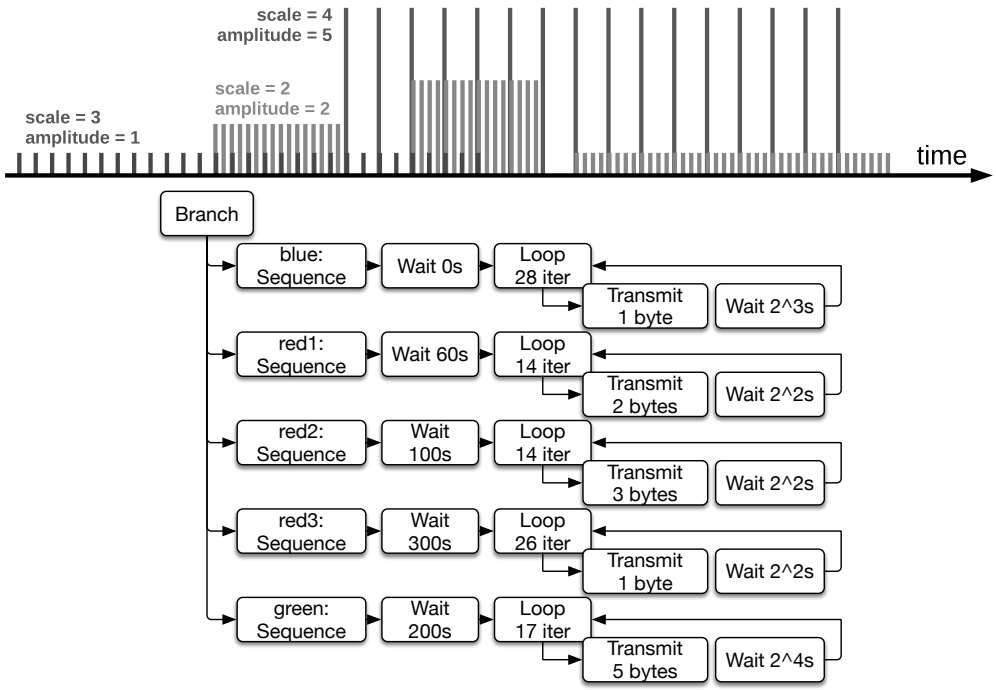


Figure 6.2: A toy example: decomposition into traffic generators at different scales and amplitudes and the corresponding instance of a DNI traffic model.

the amount of different transfer sizes. In order to compare all the densities, I applied normalization and plotted them together in a single diagram. Each local maximum in the diagram represents a cluster candidate. From Figure 6.4, I observe that a good estimate for the clustering is between 3 and 7 clusters. I also provide an mechanism for estimating the optimal number of clusters automatically, tailored for a given signal. Similar behavior can also be observed in other network traces, such as those made available publicly by the LBNL-ICSI (Lawrence Berkeley National Laboratory and International Computer Science Institute) Enterprise Tracing Project [PAB+05].

6.2.2 Traffic Model in DNI

The generators extracted from the traffic traces are meant to be represented in DNI model. The DNI meta-model and its part representing network traffic is presented in detail in Section 4.1.2. Here, I present it very briefly for the sake of self-containment of this chapter.

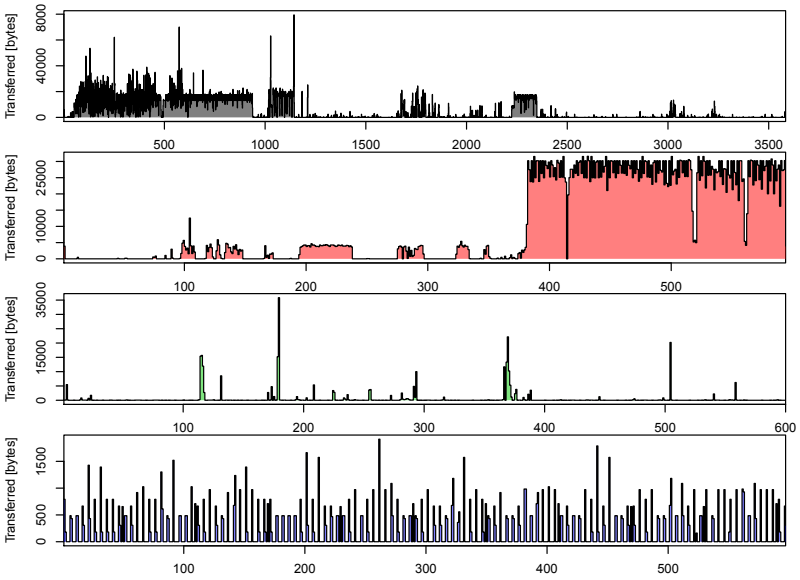


Figure 6.3: Four examples of typical traffic dumps (with duration of 60, 10, 10, 10 minutes respectively). X-axis represents time in seconds, y-axis represents bytes transferred per second.

In the DNI meta-model, network traffic is generated by traffic sources originating from communicating applications that are deployed on so-called end nodes (e.g., servers, VMs). Each traffic source generates traffic flows that have exactly one source and possibly multiple destinations. The flow destinations are software components that are located in nodes and can be uniquely identified by a set of protocol-level addresses. Flows can be composed in a workload model that defines how each flow is generated (e.g., including sequences, loops, or branches).

Besides the network traffic information, the DNI model represents also the network topology and its configuration. I say, that the extracted model is partial, because I focus on the traffic part. In fact, the tools derive simplified models of network topology and the deployment of communicating applications on nodes, but this information is incomplete as its extraction is a part of the future work. For simplicity of this extraction approach, I assume that all nodes are connected with a star topology (which is not always the case in general). The concepts of approach to extracting the other parts of DNI (i.e., topology, software deployment, and network configuration) were discussed in Section 6.1.

6.2.3 Approach based on Multi-Scale Decomposition

In Figure 6.5, I present the high-level overview of the approach, that is, the extraction pipeline. The process starts by recording real traffic on multiple interfaces

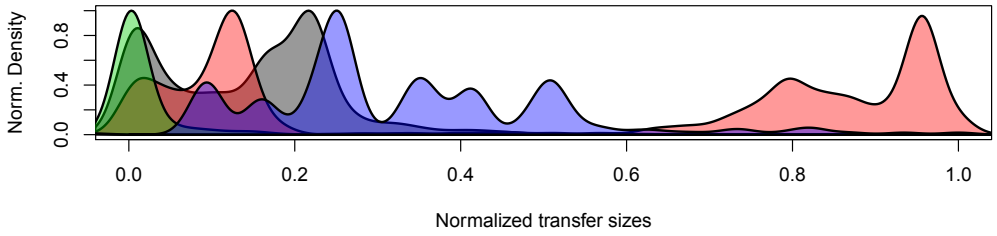


Figure 6.4: Normalized kernel density plots of transferred bytes from the example traffic dumps.

within the network (e.g., using *tcpdump*). A single traffic dump (here depicted as *tcpdump.log*) is treated as a univariate irregular time-series that is further passed into the *Multi-scale decomposition* step (explained in detail later), which in turn produces the *Decomposition matrix*. The matrix is later transformed into text format in step *Create network traffic generators* (general output) and converted to the DNI model in step *Transform generators to DNI* (DNI-specific output).

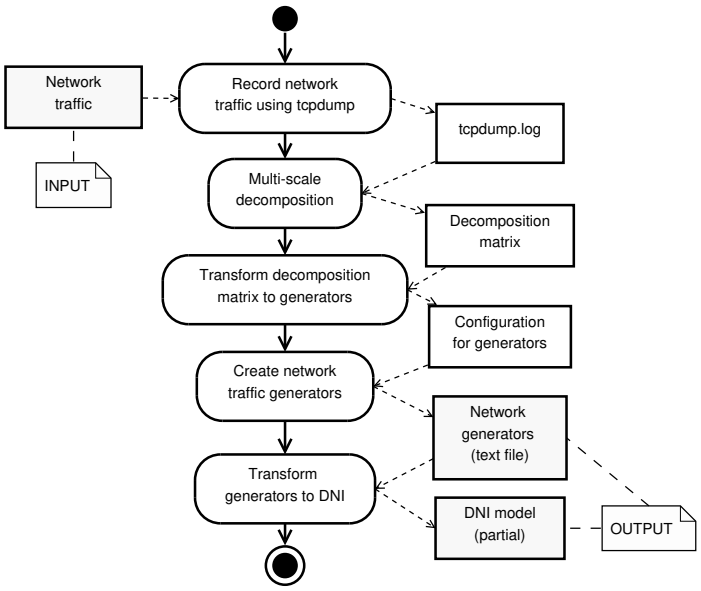


Figure 6.5: Overview of the extraction pipeline (rectangles represent data, ovals represent actions).

Each row of the matrix is a separate regularized time-series. First column represent the original signal (for debugging purposes), while the other columns represent packets emitted by simple traffic generators operating at different fre-

quencies and amplitudes. The decomposition matrix is then further transformed into the *Configuration for generators*. This is a sequence of tuples in the form of $(scale, amplitude, begin, end)$ as already explained in Section 6.2.1. Finally, a DNI model is produced which reflects the decomposition in the desired format.

Discrete Wavelet Transform for time-series decomposition

The Multi-Scale Decomposition (MSD) decomposition is loosely inspired by Wavelets, in particular by Discrete Wavelet Transform (DWT) [Mal08]. In DWT, the signal is processed at multiple scales: $Scales = (1, \ldots, maxscale)$, where: $maxscale = log_2|signal|$.

The DWT software packages, such as `wavethresh` in R, expect the input signal length to be a power of two. For each scale $s \in Scales$, the signal has half the size of the signal at the previous scale $(s-1)$, i.e.: $|signal_s| = |signal_{s-1}|/2$. Scale 0 represents the original signal, that is, $signal = signal_0$. Roughly speaking, at each scale $s \in Scales$, DWT computes the interference between the $signal_s$ and a particular wave (stretched and shifted *mother wavelet*). This yields a vector of coefficients $Coef_s = (c^s_1, \ldots, c^s_{|signal_s|})$. All scales together form a pyramid which can be represented as a single linear vector:

$$Coef = \bigcup_{s \in Scales} Coef_s$$

The DWT coefficients represent the original signal in a new space of functions. It should be noted that the time complexity of DWT is $O(|signal|)$, the length of the original signal and the length of $Coef$ vector are the same:

$$|Coef| = |signal|$$

In a similar way, the original signal can be reconstructed by a process which uses DWT coefficients $Coef$ and stretched/shifted wavelets in an inverse manner.

The applications of DWT usually involve manipulation of DWT coefficients, such as removing all coefficients from a particular scale $Coef_s$ (e.g. for compression). However, there are other application areas of science and engineering, for example, noise reduction, edge detection, time/frequency analysis.

Nevertheless, given the purpose of traffic modeling with DNI, the DWT is not particularly useful. Although it enables frequency analysis at different bands, the DWT coefficients cannot be directly mapped to simple traffic generators. Remember that a DWT coefficient represents the *strength* of interference between the signal and a wavelet at a particular time (shift) and frequency band (scale). A single coefficient can well be a negative number. This makes sense for inverse DWT, when multiple functions are summed together forming the original signal, but has little use when modeling a simple traffic generator that can only produce *positive* traffic. Due to that limitation, I *propose a custom DWT-inspired transformation* instead—the aforementioned Multi-Scale Decomposition (MSD).

Multi-scale time-series decomposition

Similarly to DWT, the notion of scales is preserved in MSD, however, the scales are processed in reversed order: $maxscale, \ldots, 1$. Instead of a pyramid of DWT coefficients, MSD transforms the signal into a matrix of *Emits*. Each scale s corresponds to a vector $Emits_s$ of equal length:

$$|Emits_1| = \ldots = |Emits_{maxscale}| = |signal|.$$

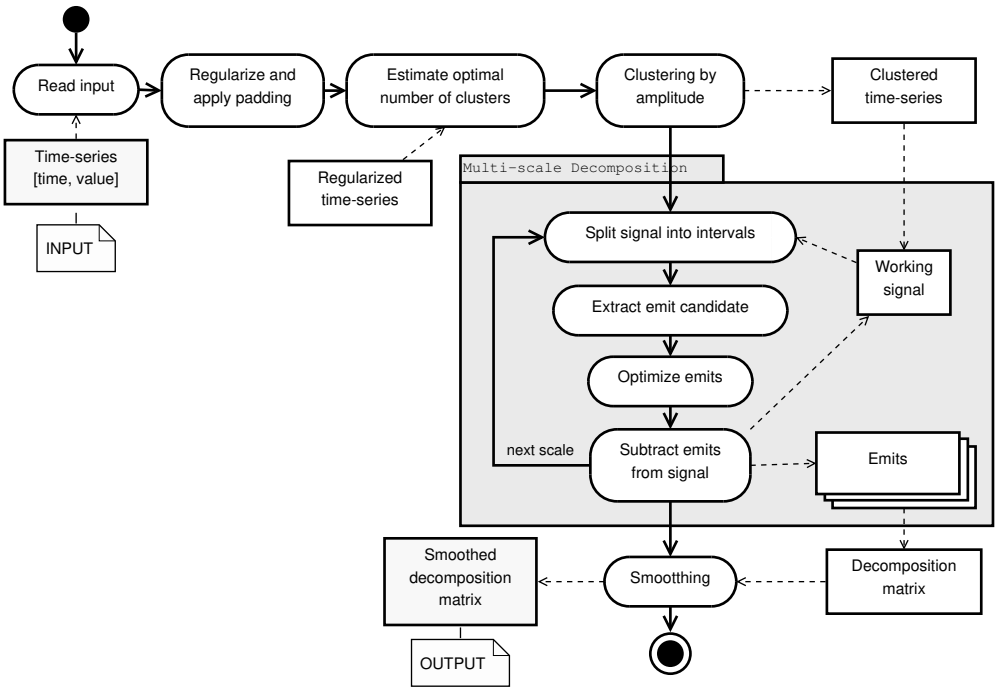


Figure 6.6: Multi-scale decomposition algorithm. Rectangles represent data, ovals represent actions.

Before the actual MSD loop, the input time-series needs to be preprocessed in three steps including: regularization, padding, and clustering.

Regularization. Every second should be represented by a single value. Gaps in the signal are replaced by zeros, while multiple values are aggregated using the *sum* function. (see Fig. 6.6: *Regularize and apply padding*).

Padding. The signal length is padded with zeros to the nearest power-of-two (e.g. signal with the size of 4000 samples is padded to 4096 samples).

Clustering. Values in the signal are clustered. As already explained, I can efficiently apply clustering of values using k-means. I can either manually set the number of clusters (e.g., six clusters) or I can use an automated estimation approach (Fig. 6.6: *Estimate optimal number of clusters*). I compute total within-cluster sum of squares for up to 25 clusters. Then, I pick the lowest number such that a higher number would only differ by less than a preselected cutoff value (10% by default). After *k-means* clustering of values, a new signal is generated where original values are replaced by the cluster centers.

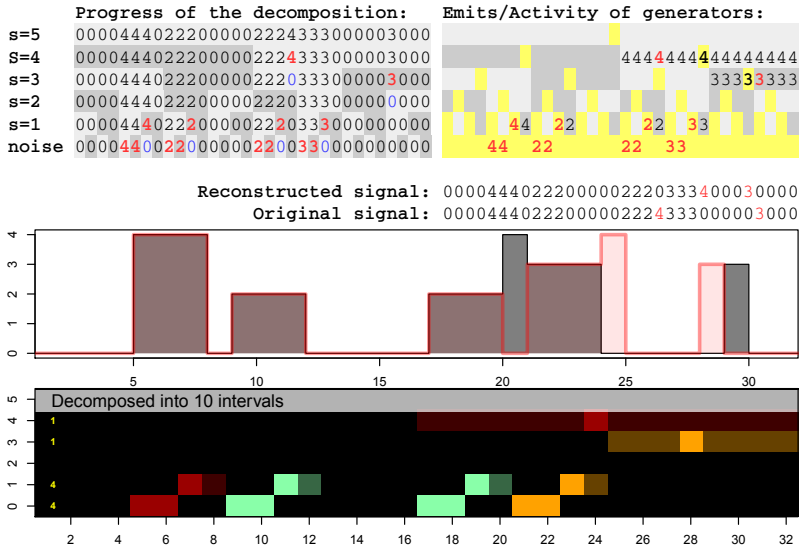


Figure 6.7: Example decomposition explained step-by-step.

Such a preprocessed signal then goes into the MSD loop, where it is passed through a sieve which subtracts certain sub-signal in each iteration until only "noise" remains. The subtracted sub-signal (a vector of *emits*) is such a traffic that can be generated by a simple traffic generator with a frequency corresponding to the particular scale.

An example decomposition of a short signal is depicted in Figure 6.7 (without any clustering nor optimization). Signal in this example consists of 32 samples which is then processed at five scales. Deriving the *emits* at each scale require: 1) split the remaining signal into equal intervals (Fig. 6.6: *Split signal into intervals*), 2) extract a single emit candidate from each interval (Fig. 6.6: *Extract emit candidate*), 3) optimize the vector of emit candidates (Fig. 6.6: *Optimize emits*).

Split the signal into equal intervals. At a given scale s, the working signal is split into intervals $(I_1, \ldots, I_{wtimes})$ of equal length $wlen$.

$$\begin{aligned} wlen &= 2^{scale} \\ wtimes &= |signal|/wlen. \end{aligned}$$

Extract a single emit candidate from each interval. For each interval I_i, a single emit candidate is selected that has the highest value such that it appears only once inside the interval.

$$Emits = \bigcup_{i \in 1}^{wtimes} select(I_i)$$

If no such a candidate exists, the $select$ function returns 0 as a candidate.

Optimize the vector of emit candidates. The $Emits$ vector is then optimized as follows:

1. Small sequences of equivalent consecutive emits are removed. The reason for this is to prevent isolated peaks to become emits at higher scales. Example:

   $$001100201110 \rightarrow 000000001110$$

2. Small gaps are removed in order to achieve higher compression. Example:

   $$000111211100 \rightarrow 000111111100$$

Time complexity of MSD is $O(nlog(n))$, $n = |signal|$. After emits from all scales were collected, vector of emits at each scale can be further smoothed (an optional step, Fig. 6.6: *Smoothing*).

Decomposition of a real signal

I have demonstrated in Figure 6.7 the result of decomposition on a toy signal consisting of 32 samples. I now take a look at decomposition of a real network traffic (10 minutes sampled at 1 second, total 600 samples). This signal has been already presented in Figure 6.3 (second diagram from the top). Without any optimization nor clustering, MSD decomposes the signal into 253 intervals as depicted in Figure 6.8.

The top diagram shows the original and reconstructed signal combined. To provide more insight into the decomposition algorithm, I use three visual tools for comparing signals from the frequency-content point of view:

1. *Periodogram* [Blo76] compares spectra of the original signal(solid line) and reconstructed signal (dashed line) as given by the Fourier Transform. From left to right are shown densities of all frequencies (from lower to higher) in the signals' spectra.

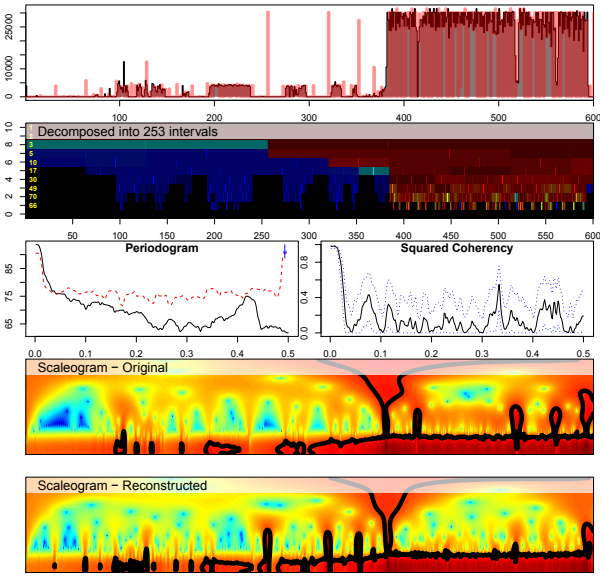


Figure 6.8: Decomposition example #1: 10 minutes of traffic without optimization.

2. *Squared Coherency* [Blo76] diagram estimates the percentage of variance in the original signal that is predictable from the reconstructed signal at the same frequency band. The higher the value, the more similar the reconstructed signal to the original is.

3. *Scaleograms* [TC98] represent signals in a time/frequency domain as given by the continuous wavelet transform using the *Paul* mother-wavelet.

When I reconstruct the signal, I can see that it matches its original with an exception of few isolated peaks and some additional high-frequency content. Frequency, however, is not the main criterion for evaluating the quality of the decomposition. More important is the overall shape of the signal that plays the major role later in the simulation phase.

With a moderate optimization, as depicted in Figure 6.9, I was able to reduce the number of intervals to 33 (i.e., 13%). At the same time, the reconstructed signal still preserves the shape of the original. The trade-off between the reconstruction accuracy and the model size can be fine-tuned using the following parameters: (1) number of clusters, (2) reduction of gaps and isolated peaks in emit vectors, and (3) final smoothing of emit vectors.

In Section 7.5, I fix the parameters of the flexible extraction process and evaluate the approach using different traffic traces from a real-world telemaintenance case study.

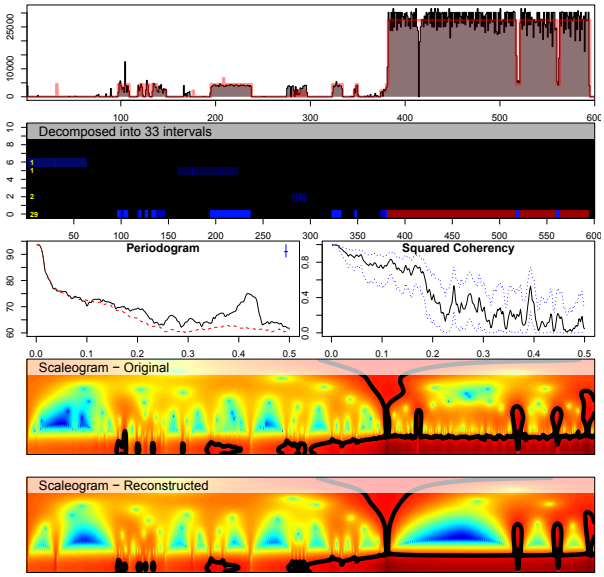


Figure 6.9: Decomposition example #2: 10 minutes of traffic with moderate optimization.

6.3 DNI Model Calibration

The main purpose of the model extraction is to support the modeler in the modeling process and provide a DNI model for optional fine tunings. The DNI model describes a network, however, to be useful for performance prediction, it first needs to be annotated with performance-relevant information. Many performance related factors included in DNI may be obtained from technical documentation or vendor data sheets. Unfortunately, not all required data is available in automated extraction process, whereas some may be unreliable or imprecise.

In this section, I briefly present the procedures for calibration of a DNI model. In the calibration process, the user manually provides missing performance annotations to fill the gaps remained after the extraction. Additionally, calibration is used for fine tuning of the performance descriptions of DNI models that deviates from the vendor-provided performance data. The following sections present the calibration of the respective parts of a DNI model.

Network Structure

Annotating the network structure—mainly the processing rates of the interfaces and devices—is done directly using the values provided in the technical specifications of the respective devices. In such way, I describe all network interfaces of switches and physical hosts, as the vendor provided data is accurate in majority of cases.

The links themselves are physical media and the only factor that influences their performance is the propagation delay, which is specified as a tabulated value for the underlying network. The maximal forwarding delays of network nodes can be found in the technical documentation. Unfortunately, as I showed in [RSKK16], some SDN switches devices do not follow the technical specification for certain conditions.

Performance measurements are required for situations in which the performance data provided by vendor is inaccurate. DNI models the performance of a network node using standard parameters used by hardware vendors: forwarding capacity (expressed in packets per second (Pps)), forwarding bandwidth (expressed in bytes per second (Bps)), and forwarding delay. Normally, the first two parameters describe the characteristics of the switching matrix of the node, whereas the forwarding delay encapsulates internal device operations, for example, flow table lookup. I approximate the node total forwarding delay as:

$$\begin{aligned} node_forwarding_delay = & forwarding_delay + \\ & + max\left(\frac{1}{capacity}, \frac{dataPayload + packetOverhead}{bandwidth}\right) + \\ & + \frac{2(dataPayload + packetOverhead)}{interface_bandwidth}. \end{aligned}$$

The *forwarding delay* describes the delay of the switching matrix, whereas the *dataPayload* and *packetOverhead* are taken as a sum from the respective protocols in the protocol stack. For example, for a typical TCP/IP/Ethernet protocol stack, an IP router processes the overheads of the IP and TCP, however it does not include the overheads for Ethernet protocol as the IP router does not forward Ethernet frames.

In the practice, the calibration of a network forwarder (e.g., switch, router, or a node based on Network Function Virtualization (NFV)) consist of estimation of the forwarding delay value (assuming that capacity and the bandwidths are known). The forwarding delay may be approximated with formula 6.1 using the data stored in the DNI model.

$$\begin{aligned} forwarding_delay = & \frac{MTU}{measured_throughput} - \\ & - max\left(\frac{1}{capacity}, \frac{MTU}{bandwidth \cdot capacity}\right) - \\ & - \frac{MTU}{bandwidth}. \end{aligned} \tag{6.1}$$

In equation 6.1, the maximum transfer unit (MTU) is a sum of data payload and packet overhead respective to the network layer in which the given device operates.

The measured throughput of the calibrated device is expressed in bytes per second, the capacity is obtained from the vendor data sheet of from the switch configuration (e.g., if capacity limiting is enabled), and the bandwidth denotes the

minimum from: the advertised bandwidth of the slowest network interface and the switching matrix (e.g., $1Gbps$).

In [RSK16], I applied the described calibration procedure for finding the forwarding delay of the *HP 3500yl* switch that worked in the SDN software mode (i.e., the active SDN flow rule was located in the software flow table). In the example from [RSK16], the capacity of the switch was limited to $10Kpps$, expected bandwidth $1Gbps$, MTU 1542 bytes (1500 bytes payload plus 42 bytes total overhead at the Ethernet protocol level), and measured throughput about $61Mbps$. Using the formula 6.1, I obtained the approximated forwarding delay of $75.6\mu s$. I conducted a batch study using a series of automatically generated QPN models varying the forwarding delay to find the best fit to the measured data. The study returned the best results for forwarding delay between 80 and $90\mu s$, whereas the same study with *OMNeT++generic* solver gave $82\mu s$.

SDN Nodes and SDN Controller

Parameterization of the SDN part of a DNI model consist of finding two groups of values: (1) three probabilities of forwarding using a given forwarding mode: SDN software flow table, SDN hardware flow table, and via the SDN controller (for each SDN device), and (2) the delays of the SDN controller applications. The later may be estimated analogically to the forwarding delay for a network node, whereas the former requires an analysis of the flows and flow tables in a controlled experiment.

The probabilities that define which forwarding mode is applied for a flow on a device may be estimated a priori as follows. For a given flow and an SDN device, one needs to count the flow rules in the flow tables that match the flow. If a rule is found only in the software flow table (with the timeout parameter set to zero, what means that the rule does not expire), then the $probablity_software = 1.0$ and the rest is 0. Similarly, one can examine the hardware flow table to set the value of $probablity_hardware$. If both tables contain a matching flow rule, then the forwarding mode of the device is decided based on the rule priorities. However, if no rules are found, or the rules have defined timeouts > 0, then the calibration requires to conduct an experiment.

In the experiment, one shall run a representative network workload while observing the flow table match counters on the SDN device. The ratio of the messages matched in a given flow table to the total number of messages in the flow will approximate the probability of forwarding in the respective forwarding mode. For example, if a DNI workload contains 100 messages (pictures, connections, etc.) and the counters report the following values: $table\ miss = 13$, $match\ table_sw = 50$, $match\ table_hw = 37$, then the probabilities can be modeled as follows:

$$\begin{aligned} probablity_software &= 0.5, \\ probablity_hardware &= 0.37, \\ probablity_controller &= 0.13. \end{aligned}$$

Protocol Overheads

Information about protocol overheads is extracted from the operating system (e.g., using the Linux command *ifconfig*). Parameters like the data payload and the protocol overhead play an important role for calculating protocol overheads in solvers that do not represent network traffic at the packet-level. To calculate an overhead, I add the value of header length multiplied by the estimated number of packets. The number of packets is calculated by dividing the message size by the value of data payload. For large messages, even small errors by specifying the data payload and the protocol overhead may add up and decrease the prediction accuracy, so caution and precision is advised.

Identification of improperly modeled protocol parameters may be conduced by comparing the total amount of data received by the receiver in the solver against the measurements in the real system at the lowest level of the network (usually in Ethernet layer). For non-standard network protocols where overheads are difficult to obtain, the values of data payload and protocol overhead may be estimated as follows: (1) pick data payload arbitrary (e.g., 2000 bytes), (2) assuming the known total volume of transmitted user data (Layer 7 or L7 data) and the volume of received data at the lowest measurable network layer (Layer 2 or L2 data, (3) calculate per packet overhead using the following approximation formula:

$$packetOverhead = dataPayload \cdot \frac{(L2_{received} - L7_{sent})}{L7_{sent}}.$$

This procedure applies to the lowest protocol in the protocol stack assuming that the measurements are conducted at the level of the lowest protocol. For example, specifying proper data payload and packet overhead (i.e., the values for MTU and headers size) of the Ethernet protocol plays important role for measurements conducted at the switch ports (a switch receives Ethernet frames). Using this calibration procedure, the overheads of the higher layer protocols are abstracted and presented as a cumulative overhead of the protocol stack.

Network Paths and SDN Flow Rules

In this thesis, I assume that the configuration of the network is known and can be acquired directly form the nodes. This included routes or paths in the network, as well as the currently installed SDN flow rules on each SDN-enabled device. Network links and ports disabled by Spanning Tree Protocol (STP) are represented in the structure part of DNI, but no routes are allowed to reference the disabled interfaces. No model calibration procedure is envisioned for the DNI part containing routing and SDN flow rules due to the nature of these data (either it is fully available or not available at all).

Load Balancing

DNI supports load balancing scenarios by allowing to specify a single source and multiple destinations of a traffic flow. The rates at which the load balancing

algorithm operates may be approximated by dividing the total data volume arrived to a load-balanced destination by the total data volume transmitted by the sender. This approach abstracts the behavior of many non-trivial load balancing algorithms, however a more detailed representation than the load balancing ratio is not supported by the DNI meta-model.

Traffic Patterns

Excluding the approach presented in Section 6.2, the traffic patterns can be modeled manually on the protocol level based on traffic monitoring tools or recorded traffic traces. Silence intervals between sending consecutive messages can be calculated manually and averaged over the duration of the entire experiment. In case of any parameter variability, a probability distribution can be constructed by observing the durations of the transmission and the silence in a network interface.

Assuming that a traffic source generates the traffic based on the ON-OFF traffic pattern (e.g., "send a 2MB picture every 10ms"), the calibration of the traffic patterns can be done manually based on the protocol level traces. However, manual calibration is error prone and the errors usually cumulate when the network is under high load, yet low accuracy calibration is possible.

The main challenge is the extraction of the duration of the silence period (OFF period). I depict schematically this phenomenon in Figure 6.10. Although the silence period at the software level may be known, it may differ at the protocol level. In many solvers (also in *OMNeT++*), the traffic generation process happens immediately, whereas in the real system, the transmitted data must be copied between the respective memory cells and mutexes need to be freed before a *sleep* instruction (that maps to the beginning of OFF period) can be executed. The same hold for the estimation of the duration and the start of ON periods. The precise modeling of the generation delay may be omitted by infrequent data generations, however, for short OFF periods, this parameter has strong influence on the accuracy of the prediction.

The generation delay shown in Figure 6.10 stems from the overheads caused by the computing nodes. This causes that the user requests submitted in a defined interval may stretch at the network level. Unfortunately, manual tuning of the length of the OFF interval is cumbersome and usually requires to either: (a) know the exact value of the network bandwidth between the sender and receiver, or (b) conduct manual analysis of the traffic traces. These challenges were the main incentives for proposing the approach to traffic extraction presented in Section 6.2.

Server Virtualization and Computing Overheads

Another challenging part is the extraction of the parameters that describe the virtual entities (e.g., nodes, network interfaces, links). In case of data transmission from one VM to another collocated VM, the data is in fact copied between the memory cells of the host machine. The exact path of the data and the overheads

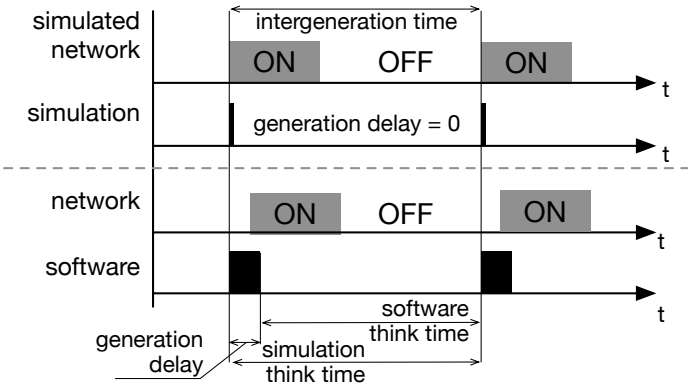


Figure 6.10: Differences between OFF period durations in the simulation (upper part) and in the real system (lower part).

depend on the host hardware, hypervisor type and the delays are difficult to estimate a priori.

Additionally, the performance of the network bridge (that is built-in in the hypervisor) depends on the overall system load, which is not reflected in DNI since all model input parameters (e.g., resource demands) are modeled as load-independent.

The values of the performance parameters describing virtual entities are finally estimated experimentally and modeled in a black-box manner. It is required to examine the performance of the hypervisor-emulated network by running stress tests using, for example, the *iperf* tool. The average bandwidth achieved in the tests is used directly to describe the speeds of the virtual network interfaces in VMs and in the hypervisor bridge. Unfortunately, obtaining repeatable calibration experiment results is difficult as the network performance of the hypervisor depends on the load and the configuration of the host system (this information is missing in DNI, however it may be available in Descartes Modeling Language (DML)). Increasing the prediction accuracy for scenarios with high VM-to-VM workloads requires to integrate DNI with a respective DML model (as described in Section 4.4. DML models the computing, virtualization, and software architecture and provides modeling entities that DNI does not support. In Section 7.3.5, I present an example of modeling a scenario including VM-to-VM workload.

6.4 Summary

This chapter describes the secondary contributions of this thesis. It includes the processes in which a DNI model can be extracted and calibrated afterwards. I presented an approach to automated extraction of traffic models as they are the most challenging to model manually in complex scenarios. For the rest

of the modeling, I proposed instructions how the automated extraction can be implemented as a part of future work.

Finally, I described how a model can be tuned in terms of performance descriptions once it is built. In this chapter, I leverage the fact that the modeled system exists, controlled experiments are allowed or trace data is available to calibrate the model. The approaches described in this chapter were applied in practice in the evaluation (if not stated differently). In Chapter 7, I validate the DNI approach and the secondary contributions of this thesis.

Chapter 7

Validation

This chapter presents the evaluation of the primary and secondary research contributions of this thesis. I validate the primary contributions that include data center network modeling abstractions presented in Chapter 4 and the transformations and automatically generated solvers presented in Chapter 5. Additionally, I validate the secondary contributions: traffic model extraction (presented in Section 6.2 and the QPN-to-LQN transformation (presented in Section 5.5).

In Section 7.1, I describe the evaluation goals. Thereafter, I present two case studies that I use to validate the contributed models and transformations with respect to the goals stated in Section 7.1. In Section 7.2, I present the *SBUS-PIRATES* case study and using it, I demonstrate the modeling capabilities of DNI and solving accuracy of the DNI solvers. This case study does not include SDN-based scenarios. In Section 7.3, I validate DNI and its solvers in the SDN context. As a case study, I use the *Cloud file backup* that is implemented using the *L7sdntest* software. Additionally, I validate the modeling flexibility aspect based on the *Cloud file backup* case study focusing on the performance prediction accuracy and the solving time of the generated SDN DNI models. In Section 7.4, I validate the modeling flexibility aspect based on the *SBUS-PIRATES* and *Cloud file backup* case studies. I focus on the model solving time and resource consumption during the solving process. In Section 7.5, I validate the approach to traffic model extraction introduced in Section 6.2. The validation is conducted using a separate telemaintenance case study. Section 7.6 validates the QPN-to-LQN transformation as a part of the secondary contributions of this thesis. I summarize the conducted experiments and conclude the validation in Section 7.7.

7.1 Evaluation Goals

By evaluating the proposed performance abstractions introduced in Chapter 4 and the model transformations presented in Chapter 5, it is not possible to *prove* that the proposed formalism and its transformations provide accurate performance predictions for any network. The model may not capture complex behaviors of the network infrastructures that are driven by sophisticated algorithms that are not represented in DNI. If the influence of the algorithms on the performance is high, the predictions may be inaccurate. Thus, the proposed meta-model and the model

transformations cannot guarantee a constant prediction accuracy. The accuracy may vary depending on scenario details.

Moreover, it is also nearly impossible to *prove* that a transformation is valid, as the size of a model instantiated from the DNI meta-model has no upper bound. However, I will show that the transformations are *complete*. Completeness of a transformations should be understood as *the ability to map any performance-relevant change in the DNI model to a respective change in the predictive model that is generated by the transformation.* In this chapter, I demonstrate that the typical data center scenarios and common *what-if* analysis questions are well supported by the DNI approach to fit into real situations. The proposed set of model transformations is representative to demonstrate the flexibility of the prediction process, however, it is by no means complete and further transformations can be implemented to extend the variety and flexibility of the approach.

The goal of the evaluation presented in this chapter is to demonstrate that:

1. The proposed DNI meta-model provides good modeling abstractions to represent the typical *what-if* performance analysis scenarios in data center network context including SDN-based networking setups.
2. The proposed set of model transformations provides the flexibility in terms of generating multiple predictive performance models that differ in provided prediction accuracy and solving times, so that the user can select the most feasible solving method based on her requirements.
3. The proposed methods for traffic extraction support the flexibility of the approach by allowing to select a trade-off between detailed traffic modeling and compactness of the model. This will influence the details included in the generated predictive models and thus contribute towards the flexibility of the approach.

As a part of the validation, I evaluate also the QPN-to-LQN transformation as a secondary contribution to demonstrate it support for performance prediction for selected Queueing Petri Net (QPN) models.

In the following, I break down the evaluation goals into three orthogonal aspects. First is the support of DNI to model data center network infrastructures. I evaluate the approach using classical network scenarios (Section 7.2) and later evaluate it for SDN scenarios (Section 7.3). Moreover, in selected scenarios, I demonstrate modeling of selected Network Function Virtualization (NFV) concepts. This evaluation aspect is discussed in Section 7.1.1.

The second aspect of the evaluation is the ability to provide performance prediction capturing the data center-specific performance characteristics. This part evaluates the model transformations that generate multiple predictive models from a single DNI model. I evaluate how well a change in the DNI model (after being transformed into the predictive models) can represent a change in a real environment. This evaluation aspect is discussed in Section 7.1.2.

Finally, I evaluate the nonfunctional features of the generated predictive models in terms of resource consumption (number of CPU cores and memory) and the solving time. Providing variety of solving times combined with different

prediction accuracies contribute to the modeling flexibility of the DNI approach. This evaluation aspect is discussed in Section 7.1.3.

7.1.1 Modeling Capabilities

By evaluating the modeling capabilities of the DNI approach, I take into account the ability of the model to represent typical data center networks scenarios in terms of: topology including the virtual topologies, hardware-related performance-influencing factors, configuration of the network including the protocols, and the network traffic. The questions that should be answered by the validation may be the following. Does the meta-model include all constructs to represent the factors and parameters that influence performance the most? Does the meta-model abstracts the aspects that have minor influence on the performance? Is it possible to represent a system that uses virtualization on the server side as well as on the network side? Which parts of the system cannot be modeled and what are the consequences of these limitations? Is it possible to represent the most performance influencing factors of SDN-based networks in DNI? Which NFV-based scenarios are supported?

The evaluation based on these questions is addressed using two case studies. First case study, called *SBUS-PIRATES*, represents a realistic road traffic monitoring scenario where distributed cameras photograph the cars in a city and transmit the pictures to the data center for processing. With this case study, I analyze the modeling capabilities of DNI in non-SDN data center networks.

The second case study, *Cloud file backup*, represents a distributed data exchange scenario. The clients request backup file resources that are distributed in the network. The users request the downloading or uploading of the files in a batch, whereas the servers deliver the requested data. Using this case study, I analyze the modeling capabilities of DNI in SDN-based data center networks. Some scenarios represent NFV setups, however no direct evaluation of NFV is provided.

7.1.2 Prediction Capabilities

I evaluate the prediction capabilities of the generated predictive models by analyzing the prediction accuracy. The main goal is to minimize the prediction error considering a given level of performance abstractions. According to [MDA04], the commonly accepted throughput prediction inaccuracy should not exceed 5% for properly calibrated model, whereas 30% is acceptable for non calibrated models (i.e., with no access to run-time data that help to tune the model accordingly to the scenario). I stress, that minimization of the performance prediction accuracy is not an ultimate goal of this thesis. I value more the flexibility of the prediction process where the user can select a feasible trade-off between prediction accuracy and the solving time of the performance model.

I evaluate the prediction capabilities in Sections 7.2 (case study *SBUS-PIRATES*) and 7.3 (case study *Cloud file backup*). In the evaluation scenarios of the former

case study, I operate mainly with uncalibrated DNI models, whereas in the latter, I calibrate the models according to the calibration procedure presented in Section 6.3.

7.1.3 Flexibility of Performance Prediction

I evaluate the flexibility of the performance prediction approach by analyzing the nonfunctional characteristics of the generated predictive models and their solvers. I focus on the solving time and the consumption of resources during solving as main metrics of nonfunctional characteristics. The flexibility is analyzed by comparing the average prediction accuracy against the solving time and consumption of resources. It is expected, that the predictive models with higher level of detail provide predictions in a longer time but with higher accuracy. In evaluation of prediction flexibility, I aim to provide a wide variety of prediction accuracies and solving times, so that the user can select the most feasible approach for a given scenario.

There are three factors that impact the performance prediction flexibility: the set of available model transformations, the set of available solvers that solve the generated models, and the flexibility of the modeling itself (as presented in Section 4.3). Additionally to the flexibility of performance prediction, I demonstrate the flexibility of modeling. This means, that a given scenario can be modeled in DNI in multiple ways (e.g., using miniDNI, or coarser approximating the granularity of the traffic model), which also influences the solving time and prediction accuracy. The analysis of the flexibility of performance prediction is discussed in Section 7.4.

7.2 Performance Prediction of Classical Networks

In this section, I evaluate the prediction accuracy of predictive models obtained in the flexible performance prediction approach. The approach leverages DNI models instantiated from the DNI meta-model (described in Chapter 4) and obtains predictive models using model transformations (presented in Chapter 5). For each DNI model, multiple descriptive models can be generated. In this section, I evaluate three predictive models: *OMNeT++INET*, DNI-QPN, and miniDNI-QPN that were generated using model transformations: DNI-to-*OMNeT++INET*, DNI-to-QPN, and DNI-to-miniDNI chained with miniDNI-to-QPN respectively.

As evaluation scenarios, I use various data center configurations that differ in terms of: network traffic workload, topology, deployment of applications, and network configuration. I use *SBUS-PIRATES* as a case study for generating network traffic. The case study is presented in Section 7.2.1.

The evaluation presented in this section is focused on prediction accuracy of the generated predictive models. I investigate non-SDN network setups. I calibrate DNI models manually if not stated otherwise. Note, that the evaluation quantitatively compares prediction accuracies of selected model solving techniques. The evaluation of solving performance of various solvers is presented in Section 7.4.

7.2.1 Case Study: Event-oriented Message Bus

The system under study is a traffic monitoring application based on results from the Transport Information Monitoring Environment (TIME) project [BBE+08] at the University of Cambridge. The system consists of multiple distributed components and is based on the *SBUS-PIRATES* (short *SBUS*) middleware [Ing09a].

SBUS Structure

The *SBUS-PIRATES* serves as communication middleware for a traffic monitoring application. It monitors the road traffic using cameras and license plate recognition techniques. The cameras photograph registration plates, which are later recognized using a recognition algorithm. Additionally, the application measures speed of cars and position of the public traffic vehicles (e.g., buses or trams). Based on this data, the application allows city traffic controllers to: (1) fine the speeding drivers; (2) collect toll based on local zone regulations—for example, when driving into the city center requires to pay a fee (e.g., in London, UK, or Göteborg, Sweden); (3) prioritize public transportation by controlling the traffic lights.

The original application scenario—as presented in [Ing09b]—includes the following application components, which communicate over the *SBUS* message bus:

- *Cameras* that take pictures of vehicles,
- *Speeding* component that calculates if a photographed car is speeding,
- *Toll* component that calculates if a fee for entering a paid zone should be applied,
- *LPR* (License Plate Recognition) that runs the recognition algorithm,
- *SCOOT* component that controls the scheduling of traffic lights,
- *Location* component that calculates the location of public transport vehicles,
- *ACIS* component that reports the current state of a bus ,
- *Bus Proximity* component that calculates the distance between a bus and traffic lights.

In the evaluation scenarios for this case study, I consider mainly two kinds of components: cameras and license plate recognitions (LPRs). The cameras are distributed in the city and take pictures of cars that are speeding or entering a paid zone. Each camera is connected to a local *SBUS-PIRATES* component that sends the pictures together with a time stamp to the LPR components. LPRs are deployed in a data center due to their high consumption of computing resources. They receive the pictures emitted by cameras and run a recognition algorithm to identify the license plate numbers of the vehicles.

In the conducted experiments, I disabled the recognition algorithm of the LPRs, so that no additional computational load is generated. Each LPR discards a message after receiving it. This allows to investigate the network performance without the influence of software bottlenecks.

SBUS middleware passes the messages between system components at the software level. Unfortunately, the original implementation of *SBUS* contains software bottlenecks (single-threaded implementation) and sporadic memory leaks

as presented in [RKZ13]. This affects the accuracy and stability of performance measurements in the hardware testbed for data-intensive scenarios. To cope with the limitations of *SBUS*, I use additionally the network performance benchmark names Unified performance tool for networking (Uperf) [upe12].

Uperf Benchmark

Due to technical limitations of the *SBUS* implementation, I used Uperf as a reference for experiments under high load to exclude the influence of software bottlenecks on the network performance. Uperf benchmark [upe12] is a network performance tool that supports modeling and replay of various network traffic patterns. Uperf was shown to emulate the network traffic without exhibiting any scalability or stability issues under high load [RKZ13].

Uperf allows the user to model the real world application using an Extensible Markup Language (XML)-based profile. It allows the user to use multiple protocols, varying message sizes for communication, a *one-to-many* communication model, support for collection of statistics.

The XML-based profiles in Uperf are similar to DNI traffic models. The profiles contain a mix of operations which can be looped and composed to represent any network workload. The set of available operations include:

- *Connect* that opens a connection to the listener,
- *Accept* that accepts a connection on the listener,
- *disconnect* that interrupts an opened connection,
- *read, recv* that represents receiving abstract data,
- *write, sendto* that represents sending abstract data,
- *sendfilev* that represents sending existing file as data,
- *NOP, think* that does nothing for a given time (waits).

Uperf supports *one-to-one* and *one-to-many* communication. For the purpose of my evaluation, I have wrapped the execution of Uperf in a software, so that a central experiment controller may start multiple Uperf instances at once and thereby support *many-to-many* communication.

7.2.2 Validation of DNI-to-*OMNeT++INET* Transformation

In this section, I apply the proposed approach to the *SBUS-PIRATES* case study and validate the DNI-to-*OMNeT++INET* transformation. I investigate the capacity of a data center network that handles the stream of pictures captured in the cameras. I assume, that the picture streams arrive to a data center via multiple external networks (e.g., metropolitan area network (MAN)) that are attached to the servers in the data center. The detailed modeling of the external connections is abstracted and the respective traffic sources are modeled as they were deployed locally. This setup presents the proper way of representing traffic from external networks in DNI and does not influence the realism of the case study.

The main goal of the experiments conducted in this part is to demonstrate that the proposed approach can support a network operator to estimate the

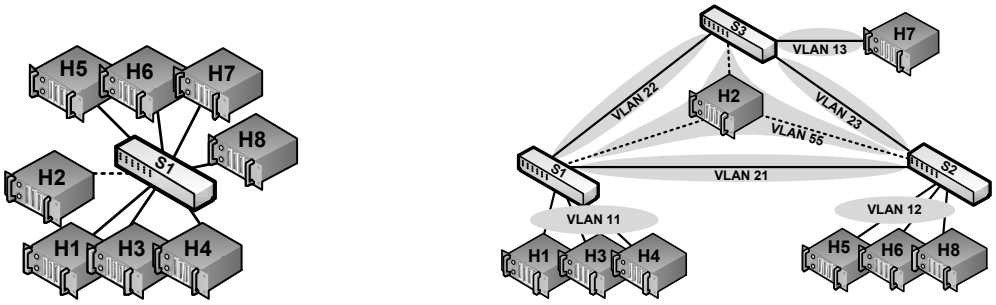
required network capacity for Transmission Control Protocol (TCP)-based networks using automatically generated simulation models that are solved later by the *OMNeT++INET* solver.

The experiments conducted in this part of the validation base on the DNI meta-model and transformation that do not support SDN-based networks. The validation and conducted measurements were published in [RKZ13].

Hardware Testbed and Experiment Setup

The system under study was deployed in a local data center in the environment consisting of eight servers and three switches. Each server is equipped with an eight-core processor with $3.3GHz$, $16GB$ of memory, and four $1Gbps$ Ethernet ports. The servers are running Ubuntu 12.04-Server operating system. I use HP ProCurve $3500yl$ switches, each with 24 $1Gbps$ Ethernet ports. I consider two physical topologies: star and tree.

The star topology of the network environment is depicted in Figure 7.1a, whereas the tree topology is depicted in Figure 7.1b. In both topologies, the host $H2$ is connected to the switches with an isolated connection to acquire the monitoring data from the switches using Simple Network Monitoring Protocol (SNMP). Host $H2$ does not take active part in the scenario to minimize the influence of the measurement traffic on the measured traffic.



(a) Star. (b) Tree.

Figure 7.1: Experimental environment and network topology for *SBUS-PIRATES* case study. Dashed links are used for monitoring and measurements, solid links for experiment data traffic.

In every scenario, I measure the amount of the traffic that traverses the network interfaces of all switches. I use the set of byte counters to measure the number of bytes transmitted through each interface of a switch. I read the values of the counters through SNMP every second and calculate the average throughput for that interval. The sizes of transmitted messages are constant in all experiments. The experimental network was isolated from other networks (e.g., Internet). During the measurements, all think times were modeled as exponentially distributed; confidence intervals are calculated for significance $\alpha = 0.05$.

In each experiment, I send predefined amount of pictures (10 000 for each camera if not stated differently) and execute the experiment 30 times for *SBUS* and 16 times for *OMNeT++*. I use 16 repetitions for *OMNeT++* due to long simulation time; to simulate one real second, the generated *OMNeT++* model needs about 100 seconds.

Modeling

For each scenario conducted in this part of validation, I prepare a DNI model and transform it using the DNI-to-*OMNeT++INET* transformation. I build each DNI model manually according to the following steps.

1. For each server Hx, create an instance of an `EndNode`.
2. For each switch Sx, create an instance of an `IntermediateNode`.
3. For each physical cable, create an instance of an `Link`.
4. For each `Node`, add the respective `NetworkInterfaces`. Annotate each `NetworkInterface` with performance description and assign bandwidth of $1Gbps$.
5. Annotate each `IntermediateNode` with performance data from the vendor data sheet: switching capacity $75.7Mpps$, switching bandwidth $101.8Gbps$, forwarding latency $< 3.4\mu s$.
6. Represent each camera and LPR module as `CommunicatingApplication` and deploy it to the respective node.
7. Assign a workload to each camera. The workload contains a loop with 10 000 iterations. It repeats two consecutive actions: transmit and wait.
8. Each transmit action refers to a `Flow` that defines source and destination of the data exchange. Cameras are sources, whereas LPRs are destinations. For each `Flow`, specify the transferred data volume according to the scenario setup. LPR components are modeled to generate no traffic.
9. Model network configuration to contain the TCP/IP/Ethernet `ProtocolStack`. Use standard values for data payload and protocol overheads, that is, payload 1500 bytes, overhead 42 bytes for an Ethernet frame (22 bytes overhead plus 8 bytes start-of-frame delimiter plus 12 bytes interframe gap). The overheads for the IP and TCP are both 20 bytes, whereas the payloads equal to 1480 bytes and 1460 bytes respectively.
10. Model the `Directions` according to the available communication paths in the network. The traffic can be directed to the neighbors of a node only if the routing table contains a correct entry. This allows to represent properly the Virtual Local Area Networks (VLANs) configuration, that is, no traffic is allowed to cross VLAN boundary if no entry in the routing table exists.

No calibration of the DNI model is conducted in this part of the validation. The consequences of this are explained in the discussion of the experiment results (Section 7.2.2). The `EndNodes` were intentionally modeled as ideal, that is, no performance bottlenecks are assumed in the operating system or middleware.

Results

In the experiments, I assume that there are N cameras connected with the data center using a dedicated network line. The network line is assumed to be a leased channel that is characterized with a maximum bandwidth $1Gbps$. The channel is only used by the cameras—there is no other traffic in the network. Every camera sends data (pictures, time stamp, etc.) of size L every p units of time. I model the intensity of the road traffic with λ photographed cars per second. I consider two scenarios. In scenario #1, I use the star topology, whereas in scenario #2, I use the tree topology. The scenarios and their parts are described as follows.

Scenario #1A: A single camera is connected with the data center using a dedicated leased line. The line has maximum bandwidth of $1Gbps$. The road traffic intensity is $\lambda = 2$ photographed cars per second. The scenario represents a real situation where the following question arises: How much bandwidth will be utilized by the system on the path between LPR and the switch if the traffic intensity increases?

Scenario #1B: I consider a situation when new cameras are added to the observed area in the city. The data center operator may consider the following question: How many cameras can be handled without network traffic congestion for a defined road traffic intensity? In this scenario, all cameras are deployed in a single VLAN with servers connected in a star topology (see Fig. 7.1a). The cameras transmit the data over a single shared network link to the LPR component.

In scenario #2, I consider the situation with multiple cameras and LPR components deployed on different servers due to predefined distribution of computational workload. This represents multiple areas in the city connected to the data center over separate network lines, each represented with nodes that are described as senders. The senders deployed in VLAN 11 transmit the data to the LPRs deployed in VLAN 12. Moreover, I model an additional source of external traffic by deploying additional camera component on host $H8$ and investigate the impact of connecting an additional part of the city to the data center.

Scenario #2 consists of four sub-scenarios. In scenario #2A, I examine the bottlenecks in VLAN 21 and in the switches $S1$ and $S2$. In the scenario #2B, I analyze the changes in the network performance if the connection between switch $S1$ and $S3$ fails. In Scenario #2C, I assume that a network operator reacted to the failure of the connection by disabling the connection to the fourth camera hosted on $H7$. Finally, in scenario #2D, I represent the situation where VLAN 22 is brought back to operation, however the camera component deployed on host $H7$ remains turned off.

Scenario #1A. In the first scenario, I set the message size to $L = 2000kB$ and vary the think time p for a single sender. In experiment A (see Fig. 7.2 left), I set the think time to $500ms$ and decrease it in steps of $50ms$. The values are arbitrarily selected to represent different traffic intensities. The measured throughput grow exponentially for lower think times. Note, that the throughput values for *SBUS* drop for the p value below $100ms$. This phenomenon is caused by scalability

problems of *SBUS* (single-threaded implementation of the *SBUS* wrapper). I comment on it more in the discussion.

Due to the technical limitations of the *SBUS* implementation, I used Uperf as a reference for experiments including high network loads. As shown in Figure 7.2, Uperf emulates the *SBUS* network traffic very closely for the feasible range of think times without exhibiting any scalability or stability issues under high load.

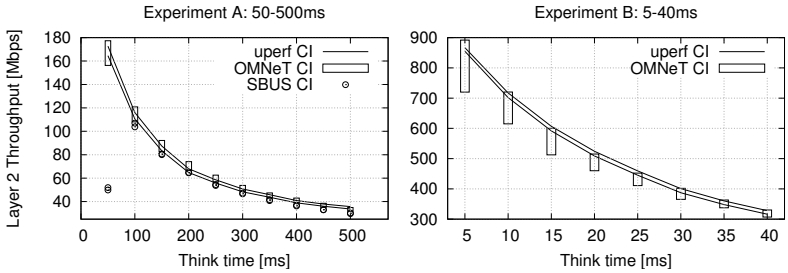


Figure 7.2: Scenario #1A: Confidence intervals for the mean throughput for think time 50–500ms (left, experiment A) and for 5–40ms (right, experiment B).

Both, *OMNeT++* and Uperf followed the trend of *SBUS*, slightly overestimating the actual throughput. Additionally, I investigated a range of smaller think times in experiment B (Fig. 7.2 right). In this experiment, the prediction errors were lower than 16%.

Based on the prediction results, the data center network has enough capacity to handle a single camera taking theoretically up to 200 pictures per second. The result of 200 is a theoretical upper bound under assumption, that transmission of the picture takes 0 time. In reality, a $2000kB$ picture is transferred for about $16ms$ over a $1Gbps$ link (excluding overheads), so the maximum intensity of the picture stream that can be handled is bound by $1000ms/(16 + 5ms) \approx 47$ pictures.

Scenario #1B. In the second scenario, I varied the think times and the number of cameras producing the traffic. I arbitrarily select the think times as 25, 50, 75, and 100ms to represent high traffic intensities. Due to the scalability issues of *SBUS*, the measurements were taken only for *OMNeT++* and Uperf, which served as a reference. The results are depicted in Figure 7.3; the relative errors are given in Table 7.1.

For think time $p = 75ms$ and $100ms$, I observe linear dependency between the number of cameras and the achieved throughput. For $p = 50ms$, the maximum capacity of the network was reached with five cameras. Similar situation was observed for $p = 25ms$. The network was saturated with the traffic from four cameras.

In the presented experiments, the relative prediction error was higher than in scenario #1; the extremes of confidence intervals of both models were a maximum 21% of the reference value apart. According to Uperf, the maximum

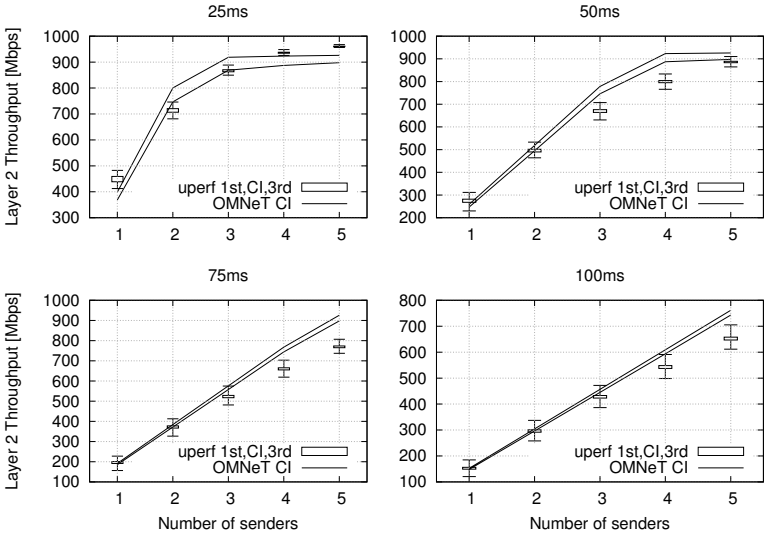


Figure 7.3: Scenario #1B: Confidence intervals for varying number of cameras and think time p. For Uperf, 1st and 3rd quartiles are shown additionally.

Table 7.1: Prediction errors and goodness of fit for scenarios #1A and #1B.

	OMNeT vs. SBUS	uperf vs. SBUS	OMNeT vs. uperf	OMNeT vs. uperf
Scenario #1A				
Think time:	50-$500ms$			5-$40ms$
Error (min–max)	$0-14.1\%$	$1-16.4\%$	$0-11.6\%$	$0-15.8\%$
Fit R^2	0.9877	0.9859	0.9885	0.99743
Scenario #1B				
Think time:	$25ms$	$50ms$	$75ms$	$100ms$
Error (min–max)	$0-20.2\%$	$0-17.1\%$	$0-20.9\%$	$0-17.5\%$
Fit R^2	0.9287	0.9793	0.9963	0.9980

achievable bandwidth is $963Mbps$; *OMNeT++* reports the maximum as $935Mbps$ (underestimating it by 3%).

Analyzing the prediction results, I observe, that the data center network can handle theoretically picture streams from: more than five cameras assuming 10 pictures per second ($100ms$ think time); five cameras assuming 13 pictures per second ($75ms$ think time); up to four cameras assuming 20 pictures per second ($50ms$ think time); and maximally two cameras assuming 40 pictures per second ($25ms$ think time).

The second scenario showed how *OMNeT++INET* predicts throughputs for high network loads. The analysis of prediction accuracy is presented in Table 7.1. In the scenarios #1A and #1B, the relative prediction error was higher than in scenario #1; the extremes of confidence intervals of both models were distanced by a maximum

20.9% of the reference value. Also the goodness of fit was usually lower than in the previous scenario.

Scenario #2. In this scenario, I represent a more complex set of communicating pairs. I deploy the following set of traffic flows (sources and destinations): $H1 \rightarrow H7$, $H3 \rightarrow H6$, $H4 \rightarrow H5$, $H5 \rightarrow H6$, and additionally in the scenarios #2A and #2B, $H8 \rightarrow H7$. The camera components are configured to spawn two threads, each sending a picture of size $L = 400kB$ every $1ms$. This represents a hypothetical situation of pairs of cameras that photograph cars from both directions (front and rear) with lower quality pictures. This may put network into stress due to high volume of smaller data portions.

In every experiment run, each sender host sent $10\ 000$ pictures. The measurements stop once all cameras report successful transmission of all pictures. The measured values of throughput are presented in Table 7.2 for scenarios #2A, #2B, and in Table 7.3 for #2C, and #2D respectively. Prediction errors were calculated as the difference between the mid points of confidence intervals.

Scenario #2A. The model kept the prediction error at the acceptable level not exceeding 25%. *OMNeT++INET* tend to overestimate the prediction. All existing bottlenecks were detected but there was also false positives: the bandwidth on paths $S3 \rightarrow H7$ and $S2 \rightarrow H6$ were overestimated by *OMNeT++INET* 24% and 8% respectively and bottlenecks were reported, although there were still free resources on that link in reality. I comment on the results in more detail in the following. The results are presented in Table 7.2, where *SBUS* is treated as the reference and *OMNeT++* as the prediction.

Table 7.2: Scenarios #2A and #2B: Measured and predicted bandwidth between network nodes.

Measured link	SBUS *Mbps*		OMNeT *Mbps*		Relative error %	SBUS *Mbps*		OMNeT *Mbps*		Relative error %
	lCI	uCI	lCI	uCI		lCI	uCI	lCI	uCI	
Scenario #2A: VLAN 22, $H8 \rightarrow H7$						#2B: no VLAN 22, $H8 \rightarrow H7$				
H1→S1	352	413	440	448	13.4%	210	246	298	314	31.9%
H3→S1	396	499	407	494	0.1%	277	302	303	323	7.7%
H4→S1	540	563	595	611	9.1%	402	440	355	367	−15.1%
H5→S2	533	545	667	684	25.4%	443	533	364	474	−13.1%
H8→S2	376	440	491	500	18.7%	470	530	630	647	25.8%
S1→S2	915	932	926	944	1.3%	897	946	937	950	1.7%
S1→S3	352	413	440	448	13.4%	—	—	—	—	—
S2→H5	616	638	667	684	7.6%	410	469	393	418	−8.8%
S2→H6	860	872	926	944	8.1%	732	851	591	680	−19.8%
S2→S3	376	441	491	500	18.5%	695	822	623	712	−12.5%
S3→H7	680	803	931	948	23.7%	696	822	623	712	−12.5%

In scenario #2A, I observe the following phenomena that affect the prediction accuracy. Firstly, the scenario simulates network traffic carried over TCP. TCP does

not allow to exceed the network capacity and cause packet drops in theory. In practice, the traffic can be dropped or delayed in a congestion situation, so that TCP can adapt the sending rate at the sender node to not exceed the capacity [Jac88]. This causes, that the flow throughput on an overloaded link propagates back to the sender, so that it transmits the traffic at a lower rate. Once the throughput on the bottleneck resource drops, the algorithms increase the sending rate. This results in the average throughput fluctuation and thus provide large confidence intervals. Note, that the DNI meta-model do not support modeling of the TCP parameters, so the behavior of the simulated protocol was defined by the default configuration of *OMNeT++INET*.

Secondly, *SBUS* confirms each successfully received message. Additionally, TCP uses retransmissions and confirmations to assure delivery guarantee. This was a direct cause of the high traffic observed on the S2-H5 link. The receiving rate of the H5 node were higher than the sending rate of the sender node H4. The difference between the throughputs is caused by the confirmations that arrived to the H5 node as a result of the H5-H6 communication.

Thirdly, the senders in this scenario transmitted data at a lower rate than modeled in DNI. The *SBUS* implementation was unable to guarantee the $1ms$ break between consecutive picture transmissions and thus transmitted the data slower than assumed. The modeled intergeneration break of $1ms$ were increased in reality by the *SBUS* software stack and the operating system overheads. Careful model calibration at the packet level (e.g., analysis of a *tcpdump*) would provide the exact length of the pauses between picture data transmissions.

Finally, several prediction errors may sum up and thus increase the throughput prediction error on an aggregate link. This was the case for the flow S2-H6 that was affected by two elementary flows that merged on the S2-H6 link, that is, H5-S2 and H3-S1. As both elementary flows were overestimated by *OMNeT++*, the throughput prediction on link S2-H6 suffered double prediction error. Similarly, the prediction accuracy of S3-H7 was affected by the elementary flows H1-S1 and H8-S2. These errors can be avoided in the future by calibrating the DNI model, not only for the senders, but for all network interfaces in the data center.

Scenario #2B. In scenario #2B, I investigate the failure of VLAN 22 (link S1-S3), that causes even more congestion on the link S1-S2. The congestion on S1-S2 was predicted accurately but it affected the flows sharing this link in the way described previously in scenario #2A. TCP plays an important role here—it defines the shares for a congested resource based on the control flow algorithms and the order in which the traffic arrives to the bottleneck. The prediction errors are at the similar level as in scenario #2A, however, several underestimations appeared due to the new traffic situation. I comment on the results in more detail in the following.

In this scenario, the S1-S2 link was shared among the flows H1-H7, H3-H6, and H4-H5. The summarized throughput rate varied between 889 and $988Mbps$ for reference measurement and 956 and $1004Mbps$ for simulated traffic. The prediction error for the summarized throughput is low, however the prediction errors for the

component flows vary between -15 and 32%. This phenomenon is mainly caused by the TCP configuration, which exact behavior is not modeled in DNI.

Similar situation is observed on the S2-S3 link that carry two component flows H1-H7 and H8-H7. Additionally, the prediction was affected by the confirmations sent for the reverse flow H7-H8, so the throughputs measured for S2-S3 are higher than the predictions.

Scenario #2C. In scenario #2C, I assume, that the network operator disables one camera (on node H8) to decrease the load in the However it did not affect the bottleneck between switches $S1$ and $S2$. The largest relative prediction error is observed on the not overloaded links (path H1-S1-S2-S3-H8) and amounts up to 26% higher predicted bandwidth than the *SBUS* reference. The most overloaded link S1-S2 was predicted accurately. The results are presented in Table 7.3, where *SBUS* is treated as reference and *OMNeT++* as prediction.

Table 7.3: Scenario #2C and #2D: Measured and predicted bandwidth between network nodes.

Measured link	SBUS *Mbps*		OMNeT *Mbps*		Relative error %	SBUS *Mbps*		OMNeT *Mbps*		Relative error %
	lCI	uCI	lCI	uCI		lCI	uCI	lCI	uCI	
Scenario #2C: no VLAN 22, no $H8 \rightarrow H7$						#2D: VLAN 22, no $H8 \rightarrow H7$				
H1→S1	244	291	311	323	15.9%	819	846	807	822	−2.4%
H3→S1	272	287	295	312	8.6%	354	365	410	495	29.2%
H4→S1	366	394	346	356	−8.3%	524	540	604	612	15.1%
H5→S2	443	521	347	457	−15.1%	527	548	677	685	26.1%
S1→S2	931	952	945	951	0.4%	884	902	938	947	5.4%
S1→S3	—	—	—	—	—	823	849	807	822	−2.8%
S2→H5	368	409	383	405	0.6%	546	571	677	685	21.3%
S2→H6	714	807	584	676	−16.8%	864	880	938	947	7.9%
S2→S3	247	302	349	363	26.3%	—	—	—	—	—
S3→H7	248	306	349	363	25%	824	849	904	919	8.7%

In this scenario, the link S2-S3 became less loaded as it now carries only the H1-H7 flow (the link S1-S3 is still disabled in this scenario). However, the H1-H7 flow still traverses the overloaded link S1-S2 and is thus affected by the TCP discrepancies between the *OMNeT++* and the reference measurement. The congestion on S1-S2 propagated back to the H1-S1 link and caused high prediction errors for H1-S1, S2-S3, and S3-H7, which carried the flow to node H7.

The influence of the TCP is also visible on the S2-H6 link. The throughput of this link was underestimated by *OMNeT++* by over $130Mbps$ what resulted in relative error of -17% (midpoint for *OMNeT++* $630Mbps$, whereas for the reference $760Mbps$). The node H6 received two component flows: from H3 and H5. Throughput for H5-S2 was underestimated, whereas the flow H3-S1-S2-H6 traversed the overloaded link S1-S2. That caused retransmissions as the queueing time on the switch S1 grew due to the congestion. Although the interface throughput on H3-S1 was overestimated, the competing flow H4-S1 achieved less bandwidth

from TCP and the errors canceled each other out. This should impact the S2-H6, however *OMNeT++* has simulated less retransmissions and thus predicted less load on the link S2-H6.

Results scenario #2D. Scenario #2D represents the situation where the VLAN 22 is repaired and link S1-S3 becomes available again, however the camera on the node $H8$ remains off. In this case, the load in the network was more balanced as five out of 9 links reached utilization of over 85%. The predictive model discovered all eight highly saturated links and reported all possible bottlenecks correctly.

OMNeT++ leveraged the freed capacity to a higher degree as it could generate more traffic in a time unit due to lack of software and operating system overheads. Higher prediction errors (up to 30%) are observed for not fully utilized links, that is, H3-S1 (absolute error $\approx 100Mbps$), H5-S2 (absolute error $\approx 150Mbps$), and S2-H5 (absolute error $\approx 110Mbps$). They were affected by the bottlenecks (and thus TCP-related discrepancies between *SBUS* and *OMNeT++*) on the links S1-S2 and S2-H6.

Discussion

In this section, I applied the DNI approach to the *SBUS-PIRATES* case study and validated the DNI-to-*OMNeT++INET* transformation. I investigated the capacity of a data center network that handles the stream of pictures captured in the cameras. The validation was conducted for manually built, uncalibrated DNI model. I used the model to investigate possible data center situations in two scenarios.

Despite the factors that decreased the prediction accuracy, the performance predictions are acceptable. Summarizing, the automatically generated simulation model solved by *OMNeT++INET* provided accurate predictions not exceeding the average prediction error of 32% in the worst case. The model correctly recognized all bottlenecks. Additionally, in case of high traffic load (presence of bottlenecks), the relative prediction errors were low and did not exceed 13%; in case of not fully saturated resources, the relative error was higher—up to 25%. I stress that the presented predictions were obtained by an automatically generated simulator; the simulation model was generated from a DNI model where most of low-level details were abstracted.

In scenario #1, I analyzed the capacity of the data center network organized in a star topology and processing the external stream of camera pictures arriving to the data center over a single dedicated link. I presented, how the network operator may analyze the capacity of the network in means of the maximal number of camera pictures processed in a second. In scenario #1A, I showed that the infrastructure can handle a stream of $2000kB$ pictures arriving to the data center at $5ms$ intergeneration intervals. Scenario #1B demonstrated, that the same infrastructure can handle multiple cameras transmitting the pictures at variable intergeneration intervals.

Scenario #2 demonstrated more complex network topology and more camera traffic flows present in the data center simultaneously. I showed, how the con-

figuration of the data center may be manipulated in the model without affecting the production environment. I demonstrated four sub-scenarios where selected links and services were enabled and disabled respectively. For this scenario, the manual analysis would be cumbersome as many flows need to share capacities of bottleneck resources, whereas DNI was able to represent the performance situation correctly providing up to 30% relative performance errors in the worst case. Note, that I assume the prediction errors up to 30% are commonly accepted values for uncalibrated models. Moreover, the way I modeled the system in this scenario resembles more from design-time than from the run-time performance modeling because the model was not calibrated using the run-time data.

There are at least three factors that caused the worst-case relative prediction errors to reach the border of 30%. In the following, I will discuss them in more details.

TCP Configuration. TCP congestion avoidance and flow control algorithms are responsible for tuning the sending rate of the sender based on the current network conditions. There are multiple flow control algorithms that observe multiple parameters to control the flow of packets [Jac88]. Scenarios in this case study run over TCP. The DNI meta-model includes only the most essential information about network protocol and does not provide support for modeling any internal TCP parameters. *OMNeT++INET* does provide simulation of the TCP internal behavior, yet the protocol is simulated with default configuration as no TCP parameters are provided by DNI. This leads to an information gap and the prediction accuracy depends on the *OMNeT++INET* default configuration.

In the scenarios, *OMNeT++INET* uses *TCP Reno* algorithm (default *OMNeT++* setting, that is not supported by the transformation), whereas modern Linux servers (starting from kernel 2.6.19) implement the CUBIC algorithm [HRX08]. Although the delivered performance at the medium-detailed level should be similar, the algorithms may prioritize flows competing for a bottleneck resource differently depending on multiple factors (e.g., moment of start of the connection, network path load level, number of competing flows). Under this assumption, the sum of throughputs of the competing flows should be predicted correctly, however the component flows may share the resource using different ratios—both phenomena were observed in the measurements and the predictions.

Calibration of the DNI model. The lack of calibration of DNI also model influenced the prediction error. This allowed to investigate the role which calibration plays in performance prediction. There are two main factors that are affected by the lack of calibration: software and operating system overheads, and the replies issued by LPR component after receiving a picture form camera.

Software overheads caused that the transmission of the pictures was additionally delayed by the software layers on the sending host. As a result, the pictures were not precisely sent in the intergeneration intervals specified in the model but suffered an additional delay. Although the software overheads are expected to be

low, they play important role if the generation of consecutive pictures is specified with intervals at the millisecond level. DNI can cope with such problems leveraging the *softwareLayerDelay* parameter in the `EndNode` entity. Finding a proper value of this parameter may be challenging however. This strengthens the incentives for providing the automated extraction method presented in Section 6.2.

Another factor that affected the accuracy of prediction was lack of calibration of the LPR reply size. High number of the small messages generated by every LPR component may incur higher throughputs caused by the protocol overheads (e.g., when the maximum size of data payload is not exceeded). Additionally, the TCP confirmations (ACK messages) for the picture streams cause additional traffic in the reverse directions. This may play important role (depending on the modeled software) for predicting performance of a bidirectional resource (e.g., link S2-H5 and H5-S2). Although the influence of the ignored stream was expected to be negligibly low, the TCP overheads may cause either (a) an underestimation of the reference throughput or, (b) observation of more traffic on the receiver side than generated by the sender (in some scenarios, for example, in scenario #2A).

Reference Measurement Method. The throughput measurement procedure may affect the accuracy of the measurements. I use the byte counters located on every switch to observe the number of bytes sent or received currently. The counters are queried every second, however the counters update-rate depends on the hardware and may differ for some devices. The measurements may be distorted for update rates that are larger or equal to the query rate. For example, the counter value may be read multiple times within the period between updates. Thus the calculated throughput for the current second may be 0, whereas it may be doubled in the next measurement (e.g., 300, 300, 0, 600, 300 ...). This does not affect the average throughput over the experiment duration, however, the confidence intervals may be larger.

Finally, long simulation times for *OMNeT++INET* heavily impacted the duration of the experiments. I decided to conduct less repetitions of *OMNeT++INET* simulations than for the reference measurements. This increased the confidence intervals for results reported by *OMNeT++INET*. Note, that *OMNeT++* was not affected by the issue with port counters update rate.

7.2.3 Validation of the DNI and miniDNI QPN Transformations

In this section, I validate the DNI approach using the *SBUS-PIRATES* case study with larger set of model transformations. The transformations include: DNI-to-*OMNeT++INET*, DNI-to-QPN, DNI-miniDNI-to-QPN. In this part of validation, I use two sources of picture streams that are directed to three LPR components deployed in virtual machines (configuration selected arbitrarily). Additionally, I manually calibrate the traffic part of the DNI model to decrease the prediction errors observed in scenarios #1 and #2 (presented in Section 7.2.2).

The main goal of the experiments conducted in this part is to demonstrate that the proposed approach can deliver multiple performance predictions with different

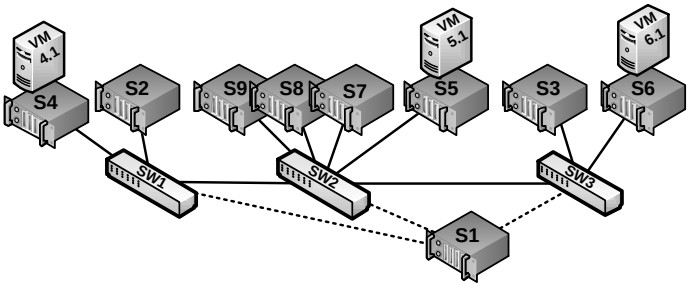


Figure 7.4: Network topology used in the experiment. Dashed links are used for monitoring, solid links for experiment data traffic. Server $S1$ is the experiment controller.

level of abstraction for a single DNI model. The generated performance models are solved using two solvers: *OMNeT++INET* and *SimQPN*, whereas *SimQPN* is used twice for models generated by the DNI-to-QPN and DNI-miniDNI-to-QPN transformations. The experiments conducted in this part of the validation base on the DNI models and its transformations that do not support SDN-based networks. The validation and conducted measurements were published in [RKTG15, RK14a].

Hardware Testbed and Experiment Setup

The system under study was deployed in a local data center consisting of nine servers and three switches. Each server is equipped with four $1Gbps$ Ethernet ports. The switches are HP ProCurve 3500yl. The physical topology and the configuration of the network environment is depicted in Figure 7.4. Server $S1$ is used to control the experiment and to acquire the monitoring data from switches using SNMP. Two servers, $S2$ and $S3$, are native (not virtualized) and serve as sources of the camera picture streams. The nodes $S4$–$S6$ are hosting virtual machines (VMs) on top of the VMware virtualization stack. Server $S8$ hosts VMware Control Center and $S9$ is a storage. Servers $S7$,$S8$, and $S9$ do not take active part in the experiment.

To obtain the baseline performance values, I measure the amount of the traffic flowing through the network interfaces of three switches. I use the counters located in the switches to measure the number of bytes transmitted through each interface. I read the values of the counters through SNMP every second and calculate the average throughput for that interval. Server $S1$ makes measurements using an isolated VLAN. The size of transmitted messages is constant in all experiments and equals 2.5 megabytes. The experimental network was isolated from other networks (e.g., the Internet). During the measurements, all intergeneration times were modeled as exponentially distributed; confidence intervals are calculated for a significance level of $\alpha = 0.05$. In every experiment, I send predefined amount of pictures ($5\ 000$ for each camera) and execute the experiment 30 times for *SBUS*,

uperf, *OMNeT++* and once for QPN (*SimQPN* cares internally about the required number of repetitions).

Modeling

For each scenario conducted in this part of validation, I prepared a DNI model and transformed it using four transformations that are subject of validation in this section. I built each DNI model manually, similarly to the procedure presented in Section 7.2.2. The DNI models contain several elements that were not covered in the previous validation scenarios. I model them as presented in the following.

Traffic Model Calibration. As observed in Section 7.2.2, proper calibration of traffic flows play important role by performance prediction. The main source of inaccuracy stems from the influence of the software and computing hardware on the intergeneration times (here called think times).

In this experiment, I minimize the influence of imprecise modeling of the picture intergeneration gaps in the camera components by manual analysis of the *tcpdump* traces. The problem can be explained as follows. The application is requested to generate and transfer a picture over the network. Assume, the picture is generated in moment $t_{app,0}$ and passed to the operating system for the transmission over the network. At the same moment, the think-time timer starts counting the time when a next picture should be generated, say $t_{app,1}$. Hence, the think time is defined as $think_time = t_{app,n} - t_{app,n-1}$. The network interface that transfers the picture data starts the transmission in moment $t_{net,0}$ and $t_{net,1}$ respectively. Ideally, I expect that $think_time = t_{app,n} - t_{app,n-1} = t_{net,n} - t_{net,n-1}$, however, the software stack and the operating system incur an additional overhead that cause $t_{app,n} - t_{app,n-1} < t_{net,n} - t_{net,n-1}$. Moreover, the pictures are not transmitted immediately, as the transmission itself takes non-zero time. The challenge is to find the difference between the think times at the network interface level and at the software application level.

Normally, the software and operating system overheads could be estimated by modeling the computing resources with Descartes Modeling Language (DML). For this validation however, the delay caused by the overheads needs to be estimated manually. The procedure for manual calibration of the traffic model was presented in Section 6.3.

To calibrate the DNI model for the following experiments, I analyzed an output of the *tcpdump* program and added a constant factor to every application-level think time of about $10ms$. The calibration was conducted for the scenario #3A with original think time of $100ms$. As shown later in the results, this allowed to decrease the prediction error for *OMNeT++INET* when compared against the results presented in Section 7.2.2. Note, that I use here a constant delay added to each think time, whereas in the reality, the value varies depending on the server load and the type of hosted applications (e.g., *SBUS* vs. Uperf), modeling of which is not supported by DNI currently.

Modeling of Virtual Machines. The scenarios assume that the LPR components are deployed onto three virtual machines. The main LPR functionality (i.e., image recognition) was disabled to minimize the CPU load on the receiver nodes. This allowed to minimize the influence of server virtualization on the transmission delays. Using Uperf as a second reference allowed to decrease the influence of the computing load even more, because the traffic generation and receiving procedures are simplified when compared against the *SBUS* implementation.

Lack of visible influence of the server virtualization architecture allowed to simplify the modeling of VMs. Each physical server hosting a VM was modeled as `EndNode` hosting a virtual switch (`IntermediateNode`) that connected the VMs to the physical interfaces. Measurements of the virtual switch performance revealed high throughput and very low latency of the virtual switching—at the level of few microseconds. This had no visible influence on the performance in this scenario, however, a highly loaded physical host (and thus the hypervisor) may impact the performance of virtual switching. I discuss more on this in Section 7.3.5, where the maximal capacity of VM-to-VM network connection is investigated.

Results

I evaluate the proposed approach in five scenarios (#3A–#3E). I deploy the camera components and the LPR components and configure the communication between the components according to the following plan: S2→VM4.1, S2→VM5.1, S3→VM6.1, and S3→VM5.1. I increase the amount of transmitted pictures per second by decreasing the think time between sending consecutive pictures to: 100, 50, 35, 20, and 10ms respectively.

I measure the throughput on the switch ports (for the reference models: *SBUS* and Uperf) and compare them against the values predicted by the generated simulation models. In this experiment, I measure throughputs on selected switch interfaces: S2→SW1, S3→SW3, and SW2→S5. I expect the monitored throughputs to be equal on each interface. The variations may happen when the network capacity is saturated, because the TCP protocol may divide the throughput unequally among the flows. The results are presented in Table 7.4, whereas the relative prediction errors in Figure 7.5.

All three predictive models estimated that the capacity of the network can accommodate a stream of $2.5MB$ pictures that arrive to the data center every $20ms$. Such stream would consume about 75% of available network capacity. Handling a stream with double frequency of interarrival times would exceed the capacity of the network.

Reference measurements. Based on the results, I observe the expected equalities in the measured throughputs for scenarios #3A–#3D. By unsaturated network, the two reference models performed similarly (max. 5.3% throughput variation relatively). In scenario #3E, the network capacity was exceeded and *SBUS* returned larger confidence intervals for the reference measurements. This phenomenon is caused by software performance bottlenecks in the original *SBUS* implementation

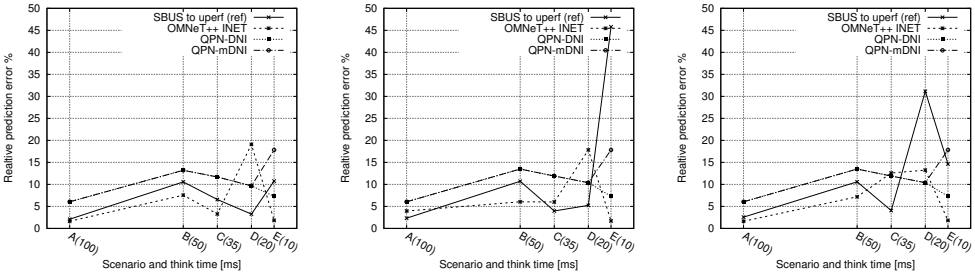
Table 7.4: Scenarios #3A–#3E: measured and predicted throughput. All values in mega-bits per second.

Link	SBUS reference1		uperf reference2		OMNeT DNI		QPN DNI	QPN mDNI
	lCI	uCI	lCI	uCI	lCI	uCI	avg.	avg.
Scenario #3A (think time $100ms$)								
S2→SW1	205	216	211	219	199	224	202	202
SW2→S5	205	215	211	219	199	214	202	202
S3→SW3	204	215	210	220	200	223	202	202
Scenario #3B (think time $50ms$)								
S2→SW1	430	449	385	410	407	448	450	450
SW2→S5	431	447	390	403	404	437	450	450
S3→SW3	430	447	383	410	408	442	450	450
Scenario #3C (think time $35ms$)								
S2→SW1	541	562	496	539	471	530	578	578
SW2→S5	526	548	505	528	454	517	578	578
S3→SW3	524	551	495	538	419	484	578	578
Scenario #3D (think time $20ms$)								
S2→SW1	631	640	579	652	702	764	675	675
SW2→S5	639	648	583	640	689	752	675	675
S3→SW3	416	426	575	648	657	728	675	675
Scenario #3E (think time $10ms$)								
S2→SW1	686	941	882	941	914	942	978	1074
SW2→S5	482	506	884	939	883	909	978	1074
S3→SW3	615	941	882	941	914	942	978	1074

as previously described in discussion in Section 7.2.2. Uperf provided more stable network load in scenario #3E, as it generates minimal additional CPU load and thus can saturate network better than *SBUS*. For that reason, I use Uperf as the reference model for prediction accuracy validation.

OMNeT++INET. The *OMNeT++INET* model predicted the throughput with the lowest average relative prediction error 7.4% (calculated as the relative difference between the mid points of confidence intervals). However, in scenario #3D, *OMNeT++* reported the highest inaccuracy of 19% mispredicting the throughput maximally by 117 Mbps for the link SW2→S5. In scenario #3E, *OMNeT++* accurately provided the maximum achievable capacity of the network and thus minimized the prediction error below 3%.

QPN models. The QPN models performed identically in low-load scenarios (#3A–#3D). The average prediction errors for the *DNI-QPN* model are usually below 10% (with maximal error of 13.5% in scenario #3B), whereas for *miniDNI-QPN* below 11.4% (maximum of 17.8% for scenario #3E). The differences between the two QPN models appear first when the network gets saturated. In scenario #3E, the model generated from the *miniDNI* overestimates the measured throughput by about $130Mbps$ reporting higher throughput than achievable in the practice.



(a) Link S2→SW1. (b) Link SW2→S5. (c) Link S3→SW3.

Figure 7.5: Scenarios #3A–#3E: relative prediction errors. Solid line compares two reference measurements using *SBUS* and Uperf.

Discussion

In this part of validation, I generated three predictive models for solving the DNI model. I analyzed the prediction accuracy of the generated models in five scenarios that represent different intensity of incoming camera pictures. I observe improvement in the stability and accuracy of predictions—compared against the results presented in Section 7.2.2—mainly thanks to manual calibration of the traffic model and lessons learned from the previous validations.

Despite the different model solving methods, the predictive models generated from the DNI model provided accurate performance predictions with prediction errors below 20%. The prediction errors depend on the applied predictive model and the scenario. *OMNeT++INET* provided the lowest average prediction error, however the predictions contained outliers (e.g., in scenario #3D). The DNI-QPN model (obtained from transforming the full DNI model) provided the most stable results prediction error below 10% (with an exception of maximal error of 13.5% in scenario #3B). The miniDNI-QPN model performed identically to the DNI-QPN model for unsaturated network. The prediction error rose in scenario #3E up to 19% overestimating the available capacity by about $130 Mbps$.

As discussed in the previous experiments, the calibration of the DNI models plays important role for delivering accurate predictions. Moreover, the semantic gaps between the predictive models need to be taken into consideration—only *OMNeT++INET* supports modeling of the TCP, whereas both QPN models simulate the traffic similar to the User Datagram Protocol (UDP). The prediction errors may be also affected by the reference measurement procedure, as well as by the differences in measuring performance by the solvers. I have investigated the inaccuracies of the performance predictions and formulate the following observations and challenges.

TCP Configuration. The first challenge is the TCP protocol configuration. Among the generated predictive models, only *OMNeT++INET* is able to mimic the behavior

of the TCP protocol. TCP relies on multiple fine-grained parameters that influence the performance, however, I do not model the values of these parameters in DNI. *OMNeT++INET* uses a larger set of parameters for the simulation than is supported in the transformation. This modeling gap was discussed in Section 7.2.2. To increase the prediction accuracy the missing simulation parameters should be provided manually.

DNI Calibration. The calibration of the traffic patterns is done manually based on the level of protocol traces. Manual calibration procedures are error prone and the errors usually cumulate when the network is under high load.

The duration of the generation delay depends on the respective software implementation and differs for *SBUS* and Uperf. In case of a single threaded implementation (e.g., in case of *SBUS*), the generation delay may take significant time. For example: for one megabyte message, the generation of a message takes about $7ms$ on average, whereas the transmission $8ms$ (for not loaded $1Gbps$ network interface). The generated predictive models represented the transmission delay properly, whereas the message generation delay was added manually to the traffic workload specification. This was the main incentive for introducing the *softwareLayersDelay* parameter to the `EndNode` entity in the next versions of the DNI meta-model (the parameter was not present in the model at the time of experiments). Thus, current implementation of the DNI meta-model provides more support for modeling software-related delays. The precise calculation of the generation delay requires the available throughput to be known beforehand. This causes the fine model calibration challenging. The issue will be fully covered after the integration of the DNI and DML solvers that is planned as a future work.

Reference Measurements. The measurement of reference values impacts the width of confidence intervals (as discussed in Section 7.2.2). The performance simulators, on the other hand, are less affected by the performance metric update frequency. *OMNeT++INET* provides the programmer full control over the statistics, so the issue of infrequent update of statistics affects the predictions minimally (only in scenarios with infrequent data transmissions). *SimQPN* run multiple simulations of the provided model and the statistics are provided by the simulation core. This minimizes the measurement errors.

Steady-state analysis of QPN. *SimQPN* conducts analysis in a steady state, which is reached after a warm-up period. This may extend the duration of the simulation as there is additional time required to reach the steady state. The steady-state analysis provides a cumulative statistics gathered during the complete experiment. This does not allow to analyze the temporal behavior of network in highly dynamic scenarios, unless the model is specially crafted to represent only an interesting fragment of a scenario.

QPN for miniDNI. The miniDNI meta-model abstracts numerous details. Despite the abstractions, proper traffic calibration plays an important role. Although the traffic patterns are not reflected in *miniDNI*, the *DNI-to-miniDNI* transformation uses the original DNI traffic profile to flatten the traffic and present it as average data volume transmitted over a given time period. Thus, errors caused by the imprecise DNI calibration may propagate to the QPN model over the miniDNI model.

7.3 Performance Prediction of SDN-based Networks

In this section, I evaluate the prediction accuracy of predictive models obtained in the flexible performance prediction approach for SDN-based data center networks. The approach leverages DNI models instantiated from the DNI meta-model (described in Chapter 4) and obtains predictive models using model transformations (presented in Chapter 5). For each DNI model, I generate two predictive models using the *DNI-to-*OMNeT++generic and *DNI-to-QPN* model transformations. The generated models are solved with *OMNeT++generic* and *SimQPN* solvers respectively.

As evaluation scenarios, I use various data center configurations that vary in terms of: network hardware, SDN configuration, SDN controller applications, load-balancing, deployment of VMs onto servers, traffic profiles, and deployment of applications. In three scenarios, I evaluate the prediction accuracy by comparing the predictions against reference measurements in a real testbed. Two additional scenarios demonstrate the capabilities of the transformations and solvers—that is, no reference measurements on the real hardware are provided. I use *Cloud file backup* as a case study for generating network traffic. The case study is presented in Section 7.3.1.

The evaluation presented in this section is focused on prediction accuracy provided by the generated predictive models. Building and calibration of the DNI models is discussed separately for each scenario. Note, that the evaluation quantitatively compares prediction accuracies of selected model solving techniques. The evaluation of solving performance of various solvers is presented in Section 7.4.

7.3.1 Case Study: Cloud Files Backup

For the evaluation scenarios in this section, I use the *Cloud file backup* case study that leverages the *L7sdntest* software [Sto16]. The *Cloud file backup* represents a scenario, in which clients request distributed file resources from servers.

There are two types of actors in this case study: servers and clients. The servers host various files in a distributed manner. A single file resource is identified by its ID. The data is interpreted as a backup snapshot but can be treated as any other identifiable regular file (e.g., a video or an archive). The file resources are redundantly located on the servers, so that a request of a client may be handled

by one or more servers and thus provide the resources from an optimal network location.

Typical scenario consists of a client issuing a request, in which it specifies the files to download. A single client request may contain a list of files as well. The server addressed by the client replies by transmitting the requested files in a batch. The client may also specify that the requested files should be delivered by transmitting each file one-by-one by specifying the duration of a pause between consecutive transmissions. The duration of the pause may be specified deterministically, as well as exponentially distributed. An example of a client request may look as follows: "request files ID: {1, 7, 18}, each every 5 seconds". *L7sdntest* uses TCP as transport protocol.

The file uploads can be examined as well by reversing the roles of client and server. In fact, the *L7sdntest* software implements each client and server in an unified way, so that each client has the functionalities of a server and vice versa. The naming of client and server is used for clarity based on the currently assigned role to the application component.

I use the *L7sdntest* software mainly for network benchmarking purposes. It has similar features to Uperf [upe12], but additionally supports centralized experiment controller, signaling, and SDN-based load balancing. The added value of *L7sdntest* (when compared against state-of-the-art solutions) consist of three SDN-specific features, that are currently under active development:

1. Each client may signal to the SDN controller informing it about the planned requests. The signaling may be transmitted over a separate network connection (out-band) or using the experimental network (in-band). The client signals to the controller, so that the controller can prepare the network to optimally handle the request, for example, reroute it to the least loaded server.
2. Each server may inform the SDN controller about its current working conditions, for example, disk or CPU load. This allows the SDN controller to optimize the routing of the network based on the current load of servers.
3. The SDN controller can manage the flow tables of the SDN switches. This allows to manipulate the contents of the SDN flow tables in a proactive fashion, that is, before (or shortly after) the traffic arrives to the switch. This functionality allows, among others, to load-balance the system without requiring a separate load-balancer server that would need to be queried for each request.

Furthermore, *L7sdntest* provides a central *experiment controller* (separate from the SDN controller) for controlling complex experiments. The experiment controller manages the experiments by executing the following tasks: transmit the experiment configuration file to each client and server; assign the roles (client or server) to each application component; start and stop the experiments; run batch experiments in which a single scenario is repeated multiple times; query all network devices for SNMP statistics regarding the traffic; and gather the measurements data for further processing. In contrast to this, Uperf works only on a defined pair of servers where each pair needs to be configured manually. The measurements gathered

by Uperf represent only the amount of data transmitted and received by the end points ignoring the state of the network devices between the end points.

Moreover, in selected scenarios, I use *Iperf* [ipe16] as an alternative tool for measuring the performance of the reference network. According to its authors, "*Iperf* is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, protocols, and buffers. For each measurement it reports, among others, the maximal throughput, packet loss, and other parameters depending on configuration." *Iperf* is used in scenarios, where *L7sdntest* cannot be applied, for example: if SNMP monitoring is unavailable (e.g., scenario #5B), or I validate network misconfiguration, so that the switches cannot be accessed from the experiment controller.

The *L7sdntest* tool allows to create realistic service provisioning scenarios and investigate the behavior of heterogeneous SDN switches under various load levels. The current support of SNMP and OpenFlow can enrich the experiments by providing monitoring the load of network interfaces, cumulative throughputs, and per-flow behavior control. The SDN-based load balancing was demonstrated in [Sto16] for Hewlett-Packard (HP) switches running the *ProVision* operating system. The support for other SDN switches is currently under active development.

7.3.2 Hardware Testbed and Experiment Setup

The validation experiments are conducted in a representative data center. The environment is presented in Figure 7.6. The experimental testbed includes the following elements:

- Three types of heterogeneous SDN-enabled switches: two $HPE\ 5700$, four $HP\ 5130$ (all run under control of the *Comware* switch operating system), and $HP\ 3500$ (running under control of the *ProVision* switch operating system). The switches $HP\ 3500$ and $HP\ 5130$ represent top-of-the-rack (ToR) switches and able to connect up to 24 servers each.
- Two separate VLANs: one for measurements and experiment control (VLAN 1) and second for the experiment network (VLAN 100). In selected scenarios, VLAN 100 runs under SDN control.
- An SDN controller deployed in node *C16*. Depending on the scenario, I use *Ryu* [Tea14] or *HP SDN VAN Controller* [HP13].
- Nine commodity servers, each equipped with four-core CPU, 32GB of memory, and 1Gbps Ethernet network interfaces. Eight servers are connected to the $HP\ 5130$'s (each switch connects two servers) and one is connected to the $HP\ 3500$ switch. All servers are connected to the experiment VLAN and the production VLAN using separate network interfaces. The connection runs over a Cat6 Ethernet cable.
- Four 10Gbps links connecting the $HPE\ 5700$ switches to the $HP\ 5130$ switches (*SW4x*). The connections run over the ten gigabit SFP+ DAC copper cables. The two $HPE\ 5700$ switches are connected with each other using copper quad small form-factor pluggable direct attach cable (QSFP+ DAC) with maximum bandwidth of $40Gbps$.

- A separate experiment controller server (*C00*) that stores the experiment scenarios and manages the *L7sdntest* software deployed on the servers. The experiment controller gathers the statistics from all switches by polling the data over SNMP using the production VLAN.

The servers used in this part of validation differ from those used in Section 7.2. Servers C10-C17 are HP DL160 Gen9 with Intel E5-2630v3 CPU (8 cores), 32GB Memory, and two 1Gbps network cards, whereas C36-C39 are HP DL360 Gen9 Server with Intel E5-2640v3 CPU (8 cores), 32GB RAM, and four 1Gbps network cards.

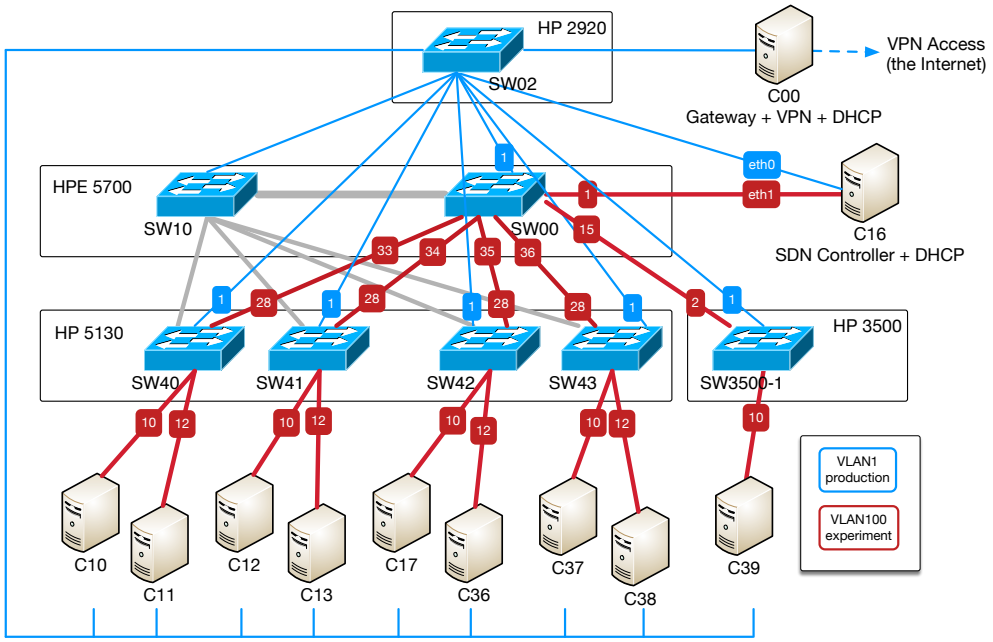


Figure 7.6: Experimental testbed used for SDN experiments. The gray links (connected to SW10) are disabled by Spanning Tree Protocol (STP) if not stated otherwise.

For each experiment conducted in the following scenarios, the experiment controller gathers and processes the performance data to serve as a reference for the prediction accuracy study. The SNMP counters data are polled every second during the experiment for each switch, network interface, and transmission direction. An experiment consist of at least 30 repetitions of an experiment scenario. Single repetition of an experiment scenario takes approximately between 3 and 60 minutes, depending on the scenario. The measured data consist of a time series where for each second an average throughput is calculated. Then, for all experiment repetitions, the so called, warm-up and cool-down periods are removed and the average throughput in each second is calculated. This yields a single

time series representing the average situation on the monitored network interface. Finally, the average steady-state throughput is calculated including such statistics as confidence intervals and percentiles.

This measurement procedure is accurate for steady-state scenarios where the momentary average throughput is stable or represents a repetitive pattern. Nevertheless, the full data set is available for detailed analysis of scenarios with dynamic throughput fluctuations and temporal workload spikes.

7.3.3 Modeling

The DNI models are built mainly manually according to the procedure presented in Section 7.2.2. The standard modeling steps include:

1. building the network topology including nodes, links, and network interfaces; specifying performance for each element of the structure according to the vendor data sheet or tailored measurements (discussed in detail in respective scenarios);
2. specifying and deploying communicating applications on the end nodes;
3. adding traffic sources to communicating applications and specifying traffic workloads (with either manual calibration or with help of the approach presented in Section 6.2);
4. modeling network protocols using standard payloads and overheads;
5. specifying paths and routes in the network according to current VLAN and routing configuration.

Additionally, in this part of validation, I represent SDN-specific elements using the modeling abstractions introduced in Section 4.2. This includes the following steps:

1. For each SDN node, I add the *SDN* `IType` aspect and specify three performance descriptions: (1) for non-SDN mode, (2) for hardware SDN mode, and (3) for software SDN mode. Each description applies to different forwarding mode of the switch: non-SDN, using hardware flow table, or software flow table respectively. A controlled experiment or an estimation is required if no performance specifications for hardware or software forwarding modes is provided by the hardware vendor. A simple method for estimation of forwarding delays is proposed in Section 6.3, whereas more sophisticated methods were surveyed by Spinner et al. in [SCBK15a].
2. An `SdnController` is deployed onto an `EndNode`.
3. For each SDN controller, a default an `SdnControllerApplication` is deployed. I install a standard *learning switch* application possibly in multiple versions (e.g., L2 learning switch, L3 learning router, broken switch) depending on scenario. All versions of the `SdnControllerApplication` are similar and are described with the `PerformanceSdnApplication` entity. The response delay of the `PerformanceSdnApplication` is measured using controlled experiments and calibrated manually (analogically to the calibration of node forwarding delays presented in Section 6.3).

4. Each SDN node becomes an *openFlowEndPoint* for communication with the SDN controller. A dummy `CommunicatingApplication` is deployed to serve as *openFlowEndPoint*.
5. Additional `Directions` are added for paths between each SDN node and the SDN controller assigned to the node.
6. For each triple of: {SDN node, flow, SDN controller application} an `SdnFlowRule` is added to the `NetworkConfiguration`. Probability parameters of `SdnFlowRules` are set arbitrarily based on the scenario (usually one of the probabilities is defined as 1.0 and rest as 0.0) or calibrated according to the procedure presented in Section 6.3.
7. For scenarios with load balancing (scenario #7 and #8), the *probability* parameter of the `Direction` objects are set arbitrarily or based on measurements from a controlled experiment.

The DNI models are then transformed using two model transformations that are focus of this section: DNI-to-QPN and DNI-to-*OMNeT++generic*. Scenario-specific modeling steps and calibration techniques are discussed in the sections of the respective scenarios.

7.3.4 Scenario #4: Upgrading Hardware to SDN

In scenario #4, I investigate the influence of the network hardware setup on the network capacity. I analyze three cases. In scenario #4A, the network handles three flows of client requests operating non-SDN switches. Each client represents one user who issues a request to a predefined server to download file resources. Then, the number of users is increased gradually to analyze the maximal capacity of the network in terms of the maximal number of customers that can be served without exceeding the network capacity.

Next, in scenario #4B, I analyze the impact of the network upgrade, which consist of enabling SDN mode on the switches. First, I enable SDN on all switches and preinstall the flow rules into the hardware flow tables before the experiment starts. The flow rules match the Internet Protocol (IP) addresses of the customers and direct the flows to the respective ports. Next, I change the SDN rules configuration and redefine the flow rules, so that the matching is conducted based on the L2 Media Access Control (MAC) addresses of the flows. This forces the switch to operate in the SDN software forwarding mode. I analyze the influence of the reconfiguration on the capacity of the network in terms of the number of users.

Finally, in scenario #4C, I increase the number of clients and differentiate the size of the requested file resources. I compare the network behavior and capacity for non-SDN and SDN hardware (i.e., using the hardware flow table) forwarding modes.

Scenario #4A: Predicting non-SDN Network Capacity

In scenario #4A, I assume three client applications deployed on servers C39, C38, and C17. The clients request files from predefined servers: C10, C12, and

C36 respectively. The clients request 100 times the same resource of size $20MB$, where each request is issued every 5 seconds. The breaks between requesting the resources are deterministic. The communication pattern is presented in Figure 7.7, where each gray arrow represent the reply of a server to the respective client. Each server reacts to the client requests immediately and starts transmission of the requested resource. The three pairs of servers communicate simultaneously and share the network infrastructure between each other.

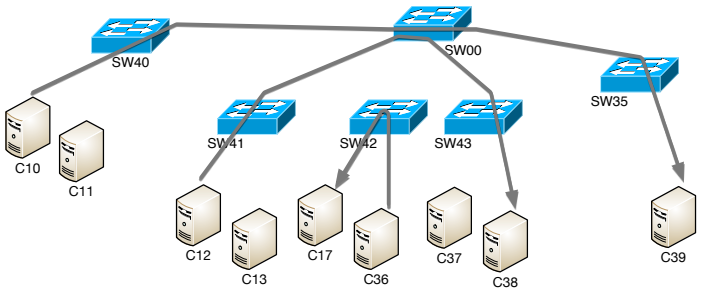


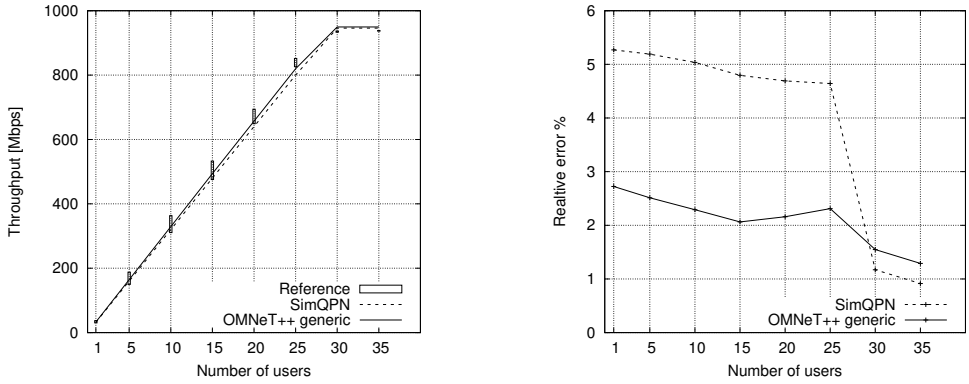
Figure 7.7: Scenario #4A: Flows of server replies (denoted by arrows).

For each communicating pair, I increase the number of users gradually starting from one and observe the capacity of the switch interfaces. I investigate the network capacity for 1, 5, 10, 15, 20, 25, 30, and 35 users. Confidence intervals are calculated for a significance level of $\alpha = 0.05$.

Scenario #4A: Results

In this scenario, I measure prediction accuracy of two predictive models: *OMNeT++generic* and QPN. Based on the obtained predictions, I observe that the network reaches its maximal capacity at the level of about $25 - 30$ users. Throughput does not increase beyond the maximum of $942Mbps$ for 30 and 35 users. The measurements of reference throughput are obtained using SNMP-based monitoring procedures implemented in *L7sdntest* software according to the presented measurement procedure. The results are stable and the size of confidence intervals do not exceeds $60Mbps$. The measured and predicted throughput on a selected network interface (SW35→C39) is presented in Figure 7.8a, whereas the relative prediction error in Figure 7.8b. The predictions are given as averages, whereas the reference measurements as confidence intervals.

Both generated predictive models provided very good performance prediction accuracy. *OMNeT++generic* delivered predictions with lower error (maximally 3%) than *SimQPN* (maximally 5.2%). The maximal absolute prediction error was below $40Mbps$ for *SimQPN* and $20Mbps$ for *OMNeT++generic*. For the case of $1 - 25$ users, both solvers underestimated the throughput, whereas for fully saturated network, the predictions were overestimating the measured capacity by maximally $14Mbps$.



(a) Measured and predicted throughput.

(b) Relative prediction error.

Figure 7.8: Scenario #4A: Network throughput measured on link SW35→C39.

Scenario #4B: Upgrading the Switches to SDN

In this scenario, I reconfigure the switches to work in SDN mode and compare the prediction accuracy for SDN-based network. The SDN switching is based on flow rules that match the IP addresses of the traffic flows. I assume proactive flow insertion, so the flow rules are inserted by the SDN controller before the arrival of the traffic. I use the experiment setup presented in scenario #4A.

Next, I measure the differences in the performance of the reference network setups operating in non-SDN and in hardware SDN modes. I assert that all flow rules are inserted into hardware flow tables of the respective switches and the table capacity is not exceeded. To analyze the performance in SDN hardware mode, I adapt the DNI models by adding the respective `SdnFlowRules` and setting their probabilities to *probabilityHardware=1.0*, whereas the other probabilities are set to 0.0.

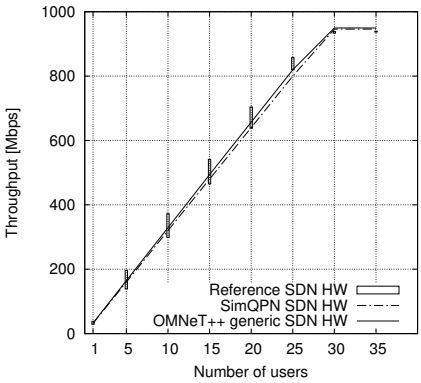
Finally, I model the next SDN reconfiguration, where the flow tables are configured to match the incoming traffic based on the L2 MAC addresses instead of IP addresses (which in fact represents routing). This forced the $SW35$ switch to install the rules into the software flow table. The rest of the switches installed the new rules in the hardware tables. Thus, for the switch $SW35$, I investigate the offered network capacity in the SDN software mode. The other switches contain only hardware flow tables, so their performance in SDN software mode cannot be analyzed. The software SDN forwarding mode is modeled in DNI by setting the probabilities of respective `SdnFlowRules` to *probabilitySoftware=1.0*, whereas the other probabilities are set to 0.0.

Each DNI node of `IType` SDN was described with one (or two for $SW35$) additional performance descriptions. The official vendor data sheets do not include the performance of SDN, so I conducted a series of controlled experiments to

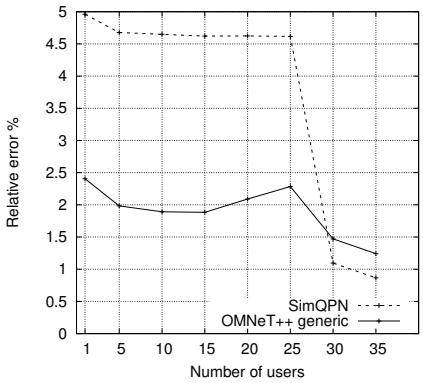
investigate their performance. The forwarding performance and other performance-relevant characteristics of the switches have been published in [RSKK16].

Scenario #4B: Results

Similarly to scenario #4A, I observe low prediction errors for both analyzed solvers. The predictions for SDN hardware mode provide almost identical capacity prediction. The network gets saturated for 30 users and offers maximum throughput of about $942 Mbps$. The generated predictive models provide high accuracy with maximum prediction errors of 5% for *SimQPN* and 2.5% for *OMNeT++generic* respectively. The measured and predicted throughputs on the network interface SW35→C39 are presented in Figure 7.9a, whereas the relative prediction error in Figure 7.9b. Predictions are given as averages, whereas the reference throughput as confidence intervals.



(a) Measured and predicted throughput.

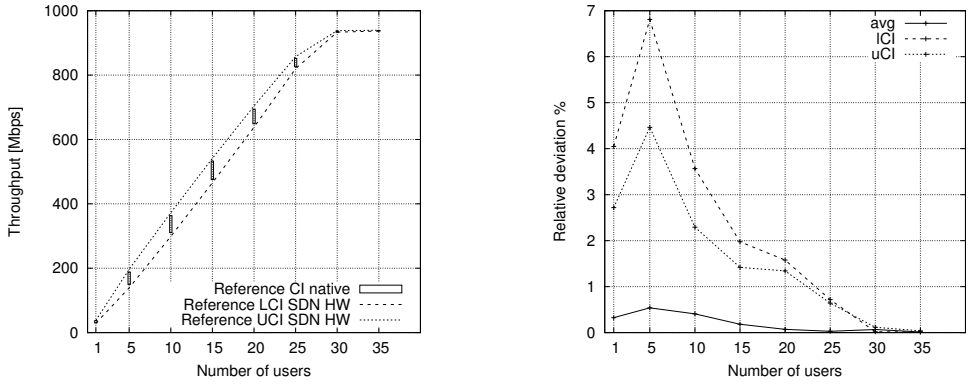


(b) Relative prediction error.

Figure 7.9: Scenario #4B: Network throughput measured on link SW35→C39. The switches operate in SDN hardware forwarding mode.

Based on the obtained predictions, I observe almost identical reference performance for native (non-SDN) and SDN hardware forwarding modes. I analyze the differences in offered throughput performance between the reference measurements. The results for throughput are presented in Figure 7.10a, whereas the relative differences between throughputs (for average throughput and the bounds of the confidence intervals) in Figure 7.10b.

The results presented in Figure 7.10 confirm that there is no significant difference in performance between the native and SDN hardware mode for the analyzed switches. The maximum relative deviation of offered throughput does not exceed 1%, however the bounds of confidence intervals differ up to 7% for the measurement with 5 users. The deviations generally stem form the measurement errors.



(a) Reference throughputs.

(b) Relative performance difference.

Figure 7.10: Comparison of reference throughputs for non-SDN (scenario #4A) and SDN hardware modes (scenario #4B).

Figure 7.10 depicts measurements for switch $SW35$, however the measurements obtained for other switches are similar. Other performance characteristics of the analyzed switches are presented in [RSKK16].

Software SDN forwarding mode. In the next experiment, I reconfigure the SDN switches to forward the incoming traffic based on MAC addresses instead of IP addresses. This forced the $SW35$ switch to install the rules into the software flow table, whereas the other switches installed the rules in the hardware flow tables. For such configuration, I investigate the capacity of the network in terms of maximal number of the users that can be served without overloading the network.

Setting the SDN forwarding mode to *software* caused a drastic drop of the offered throughput. The switch $SW35$ was able to deliver maximally $62Mbps$ of throughput which corresponds to 6.5% of the maximal throughput in the SDN hardware mode.

There are two main factors that contribute to the observed performance drop. First, the switch operating system limits the maximum switching capacity to $10\ 000$ packets per second. This limit is configurable and can be set to lower values, however for the maximum setting, the switch consumes almost 90% of its CPU resources (as presented in [RSKK16]).

Second, the software flow table is usually implemented using general-purpose synchronous dynamic random-access memory (SDRAM), so the lookup procedure consumes additional time to find a matching rule in the flow table. This incurs additional forwarding delay that needs to be estimated empirically.

In Figure 7.11, I depict the throughput measurements of the *HP 3500yl* switch operating in the SDN software mode under various settings of the switching capacity limit. Additionally, I depict performance predictions for arbitrarily selected

values of forwarding delay. Moreover, I present the prediction of the maximal theoretically achievable throughput of $118Mbps$ (for no forwarding delay) assuming the data payload size of 1500 bytes.

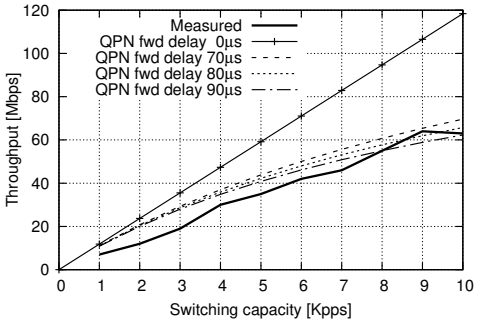


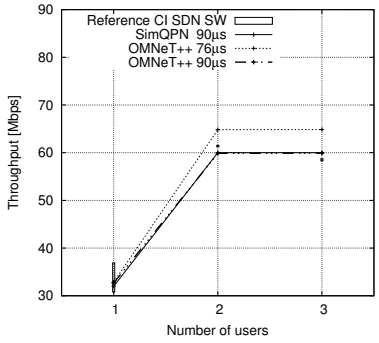
Figure 7.11: Measurement (and exemplary QPN-based prediction) of *HP 3500yl* performance in SDN Software mode for different settings of switching capacity. Excerpted from [RSK16].

In scenario #4B, I modeled the configuration of $SW35$ in DNI by specifying the `softwareSwitchingPerformance` object. I set the *switchingCapacityPPS* parameter to $10\ 000$ packets per second. This represents the maximal switching capacity offered by the switch operating system. I estimated experimentally the *forwardingDelay* in SDN software mode to $90\mu s$. Additionally, I analyze the performance for *forwardingDelay*s set to $76\mu s$ (as estimated using the method presented in Section 6.3). I depict the measured and predicted throughputs (measured on the network interface SW35→C39) in Figure 7.12a, whereas the relative prediction errors in Figure 7.12b. Predictions are given as averages, whereas the reference throughput as confidence intervals.

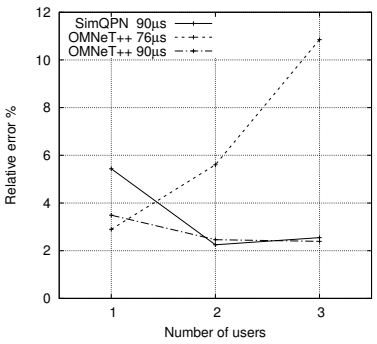
Despite the drastic performance degradation, the predictive models provided accurate predictions and estimated the maximal network capacity to handle the traffic workload of a single user. *OMNeT++generic* simulation predicted the average throughput accurately wit maximal prediction error below 4% (for forwarding delay $90\mu s$). *SimQPN* performed similarly, however the prediction error reached about 6% for a single user. The alternative method for calibrating the forwarding delay resulted in higher inaccuracies and the prediction error for *OMNeT++* reached 11%. This is still considered as acceptable prediction accuracy under the assumption that the forwarding delay was calculated a priori using an analytical formula.

Scenario #4C: Diversified Resource Requests

In scenario #4C, I assume that multiple users request the resources of diversified sizes: $r_1 = 1000$, $r_2 = 100$, and $r_3 = 10MB$ respectively (selected arbitrarily). The deployment of clients and severs, as well as the flows of file transfers (server replies) are presented in Figure 7.13. Each set of file resources is requested by



(a) Measured and predicted throughput.



(b) Relative prediction error.

Figure 7.12: Scenario #4B: Network throughput measured on link SW35→C39. The switches operate in the SDN software forwarding mode.

a single user. The consecutive resources are requested every 5 seconds until 100 requests are served. I configure the switches to operate in the SDN hardware mode. The SDN part of the DNI model is built analogically to the model presented in scenario #4B.

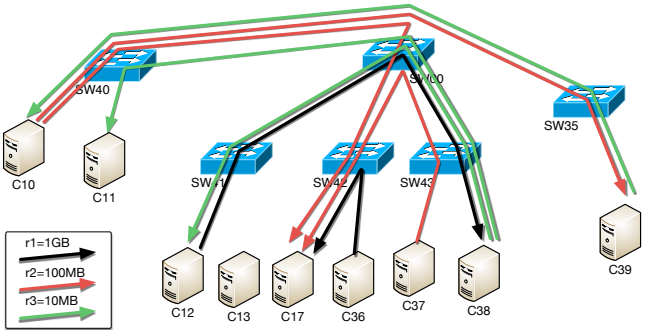


Figure 7.13: Scenario #4C: Experimental testbed and file transfer flows.

For the network capacity analysis, I selected representative network links described in the following format: "A→ • B". This means, that I measure the throughput on the receiving side (marked as •) of the link connecting the nodes A and B. The results obtained in scenario #4C are presented in Table 7.5.

In the experiments, both solvers provided accurate performance predictions. For the first time, the QPN model provided more accurate predictions compared to *OMNeT++generic*. For the fully saturated links (i.e., SW42 • → C17, SW00 → • SW43, SW43 • → C38), the relative prediction error was below 1%. The link SW00 → • SW42 was lightly loaded and the relative error was higher, however the

Table 7.5: Scenario #4C: Measured and predicted network capacity.

Measured port			Reference *Mbps*		QPN *Mbps*	Relative error %	OMNeT *Mbps*		Relative error %
			lCI	uCI	avg		lCI	uCI	
SW42	• →	C17	917	941	927	0.2	948	954	2.4
C36	→ •	SW42	788	814	924	15.4	945	954	18.6
SW00	→ •	SW42	153	161	160	1.9	136	193	4.8
SW00	→ •	SW43	911	945	924	0.4	945	954	2.3
SW43	• →	C38	911	942	924	0.3	945	954	2.5

absolute error was below $10Mbps$. *OMNeT++generic* overestimated all predictions by slightly exceeding the maximum network capacity at the data level by up to $25Mbps$.

An interesting phenomenon is observed for the link C36 → • SW42. Both solvers overestimated the throughput reporting full link utilization. This observation was expected and is caused by TCP used by the *L7sdntest* software, whereas the support for TCP is not available in the predictive models.

In the reference scenario, the capacity of the link SW42 • → C17 was exceeded by three flows: C10-C17, C37-C17, and C36-C17. The two former flows require in total about $160Mbps$ of throughput, whereas the latter consumes full capacity of a $1Gbps$ network interface (see, for example, SW00 → • SW43). As a result, the limited capacity of SW42 • → C17 provided less capacity than required by the three aggregated flows.

In such situations, TCP congestion control informs the sender to decrease the transmission rate to avoid packet losses and retransmissions. The predictive models, however, do not implement the congestion control algorithms, so in the simulation the node C36 was limited only by the throughput of the network interface ($1Gbps$). This lead to excessive queueing on the $SW42$ and possibly packet losses caused by the overrun of the SW42 transmitting queue. In fact, such situation could be observed for case studies based on UDP. Thus, for similar scenarios with TCP, the *OMNeT++INET* solver shall be used (with awareness of its limitations as presented in Section 7.2.2). Unfortunately, *OMNeT++INET* does not support SDN currently.

7.3.5 Scenario #5: Physical and Virtual Nodes

In scenario #5, I introduce server virtualization on node C13 and modify the services deployment to obtain new configuration of flows. The server $C13$ becomes a Xen [BDF+03] hypervisor that hosts two virtual machines $C13a$ and $C13b$. Each virtual machine is assigned with four CPU cores and 8GB memory. I redeploy *L7sdntest* services to leverage the high bandwidth of the 10Gbps small form-factor pluggable (SFP+) links between the switch $SW00$ and switches *SW4x*. Additionally, I configure a new flow of file resources that connects the virtual machines $C13b$ and $C13a$ to investigate the influence of the hypervisor on the network capacity. I diversify the sizes of file resources as in scenario #4C, that is: $r_1 = 1000$, $r_2 = 100$,

and $r_3 = 10MB$. Each user requests 100 copies of a resource, each every five seconds. The testbed configuration used in scenario #5 is presented in Figure 7.14.

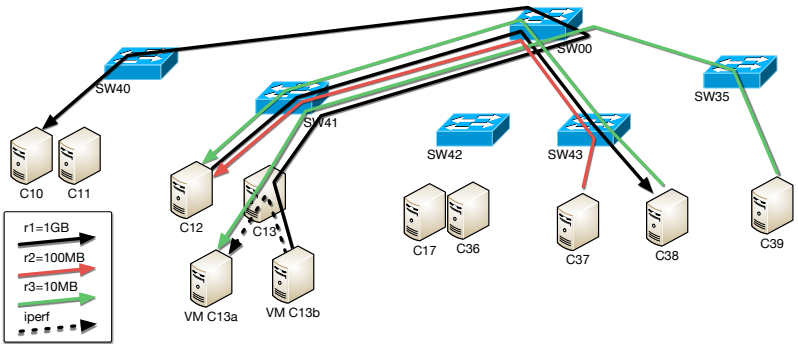


Figure 7.14: Scenario #5: Experimental testbed and file transfer flows.

I follow the same measurements procedure as described in scenario #4 (presented in Section 7.3.4) for all flows traversing a physical switch. Unfortunately, the SNMP implementation in the Xen hypervisor (node $C13$) cannot provide stable reports from virtual interface byte counters. For this reason, I measure flow C13b-C13a separately using *iperf* [ipe16] to estimate the maximal capacity of the VM-to-VM connection. Therefore, I divide this scenario into #5A and #5B. Scenario #5A contains all flows but the C13b-C13a, whereas scenario #5B measures the isolated maximal network capacity of the flow C13b-C13a.

Modeling

The modeling of the server virtualization consisted of defining the VMs, a virtual switch (a bridge running on the hypervisor), and virtual links connecting the VMs to the hypervisor bridge. I assume that the links and the VM network interfaces have infinite capacity. This resulted in setting the bandwidth of links and network interfaces to $1000Gbps$ in both predictive models (as an approximation of infinity). The maximal bandwidth of the virtual hypervisor bridge was specified using forwarding delay. The value of forwarding delay was estimated using a controlled experiment and set to $1\mu s$ for the *iperf* workload.

The precise estimation of forwarding delays is challenging. It is difficult to measure the delays at the microsecond level without dedicated equipment. Moreover, the delays incurred in the hypervisor strongly depend on the configuration and utilization of the physical host and the VMs deployed on it.

The assignment of insufficient resources to a VM (in terms of CPU cores or memory) may also affect the transmission bandwidth at the respective VM. For example, deployment of *L7sdntest* software components onto the VMs utilized more CPU resources and caused about 10% of internal network performance degradation. Larger degradations were observed by under-provisioning of CPU or memory resources. Thus, a DNI model is incapable of precise modeling of the influence

of computing performance on the network unless the integration of performance prediction with DML is used. The results obtained for scenario #5B assume that the DNI model was calibrated under repeatable constant level of computing load on node $C13$.

Results

The prediction results obtained in scenarios #5A and #5B are presented in Table 7.6. In scenario #5A, I observe prediction accuracy similar to scenario #4C. Both predictive models delivered predictions with low errors. *SimQPN* solver performed better than *OMNeT++generic* and provided up to 2% more accurate predictions. Absolute prediction error of the high-bandwidth link ($SW41 \rightarrow SW00$) was low and did not exceed $40Mbps$ on average.

Table 7.6: Scenario #5: Measured and predicted network capacity.

Measured port			Reference *Mbps*		QPN *Mbps*	Relative error %	OMNeT *Mbps*		Relative error %
			lCI	uCI	avg		lCI	uCI	
Scenario #5A (no flow C13b → C13a)									
SW41	• →	SW00	1834	1888	1848	0.7	1879	1924	2.2
SW00	• →	SW41	171	196	192	4.9	163	232	7.9
SW41	• →	C12	152	174	176	7.8	154	208	11.0
SW41	• →	C13	18	22	16	19.6	14	19	17.4
SW00	• →	SW40	920	941	924	0.7	940	962	2.2
SW00	→ •	SW43	917	944	924	0.7	938	961	2.1
SW43	• →	C38	917	942	924	0.6	945	954	2.2
Scenario #5B (flow C13b → C13a with *iperf*)									
C13b	→ •	C13	11052	11149	17040	54	16823	17987	57
C13	• →	C13a	11052	11149	10878	2.0	9999	10256	8.8

Anomalies can be observed for three of the monitored network interfaces. The flows sharing the path $SW00 \rightarrow SW41 \rightarrow C12$ and $C13$ consumed less network resources than predicted by the models. The total consumption of capacity on the link $SW00 \rightarrow SW41$ is expected in theory as maximally $192Mbps$. Despite the ideal prediction of *SimQPN*, the reference measurements provided larger confidence intervals and thus the average is reported below the expected $192Mbps$. *OMNeT++generic*, however, measured higher variation of the average consumed capacity and thus the prediction is provided with higher accuracy error. This phenomenon propagates further to the links $SW41 \rightarrow C12$ and $SW41 \rightarrow C13$, so higher prediction errors are observed. Note that the absolute prediction errors are low and do not exceed $25Mbps$ and $5Mbps$ for links $SW41 \rightarrow C12$ and $SW41 \rightarrow C13$ respectively.

Despite the challenging calibration procedure, the performance predictions in scenario #5B returned accurate results with 2% and 8.8% prediction error for *SimQPN* and *OMNeT++generic* respectively. The flow $C13b \rightarrow C13$, however, was affected by the TCP-UDP modeling gap, that is, the predictive models analyze the traffic in an UDP-fashion, whereas the reference communication runs over

TCP and the throughput of the flow is limited by congestion control algorithms according to the bandwidth of a bottleneck resource.

7.3.6 Scenario #6: SDN Switch Misconfiguration

In scenario #6, I assume that an error in SDN flow rule configuration causes all switches to misinterpret the rules located in the flow tables and forward all traffic via the SDN controller. In this way, I indirectly examine the performance of the SDN controller in handling excessive *packet-in* traffic.

Each switch can be configured to execute a default action if no rule in the flow tables can be matched. There are three available default actions: (1) forward to controller, (2) drop packet, or (3) broadcast to all but incoming ports (as in non-SDN switches). In this scenario, I configure the switches to execute the first action. Additionally, I modify the SDN controller application to return the *flow-mod* messages that do not install any rules in the flow tables. Instead, the switch is instructed to forward each packet directly to the proper outgoing port.

To demonstrate such behavior, I assume that two servers—$C12$ and $C38$—are communicating via network path containing three switches with SDN support: $SW41$, $SW00$, $SW43$. Next, I enable SDN on switch $SW41$, whereas the other switches work in native mode. In this scenario, *Ryu* SDN controller is connected directly to $SW41$ over a dedicated network link. I configure node $C38$ to represent a single user requesting a $20MB$ resource from $C12$ every five seconds. The testbed configuration used in this scenario is presented in Figure 7.15.

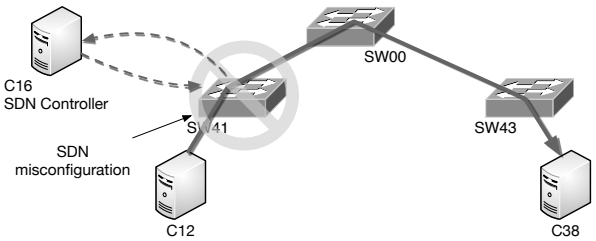


Figure 7.15: Scenario #6: Experimental testbed and file transfer flows.

Unfortunately, in this scenario, *L7sdntest* could not establish a stable connection between the client and the server. The software requires stable network conditions to conduct batch experiments. Note, that in each of 30 experiment repetitions, the software handles 100 consecutive file transfers and reports an experiment failure if any of the experiment repetition fails. For this reason, I used *Iperf* to emulate the user behavior for this scenario.

Modeling

In this scenario, I build the DNI model similarly as in the previously presented SDN scenarios. The modeling consists of defining a flow between nodes $C10$ and

$C38$ with respective `SdnFlowRule`. The parameter *probabilityController* is set to 1.0. The modeling of a `SdnControllerApp` faces similar challenges to the calibration of forwarding delay of the SDN switch working in SDN software mode (as presented in Section 7.3.4). I empirically estimate the per-packet processing delay of `SdnControllerApp` to $8ms$.

Results

The reference measurements obtained in this scenario originate from three various *iperf* settings. First, I run default *iperf* command that measures the maximal capacity of the network connection. Second, I switch the transport protocol to UDP and measure the available bandwidth. This however, does not allow *iperf* to send more data than $1Mbps$ unless the receiving side confirms successful receptions. This mechanism is implemented internally by *iperf* and does not refer to any of UDP features. Finally, I force *iperf* to send maximally $32Mbits$ of data without waiting for the confirmation of the receiving side.

All three types of reference workloads are compared against the prediction results provided by *SimQPN* and *OMNeT++generic*. I use the same DNI model for all three reference measurements. The results are presented in Table 7.7.

Table 7.7: Scenario #6: Measured and predicted network capacity.

Measured port	Reference *Mbps*	QPN *Mbps*	Relative error %	OMNeT *Mbps*		Relative error %
	avg	avg		lCI	uCI	
Reference: default TCP *iperf*						
C12 $\rightarrow \bullet$ SW41	1.24	32.00	2481	27.12	38.60	2550
SW43 $\bullet \rightarrow$ C38	1.24	1.39	12	2.14	2.70	95
Reference: default UDP *"iperf –udp"*						
C12 $\rightarrow \bullet$ SW41	1.03	32.00	3007	27.12	38.60	3090
SW43 $\bullet \rightarrow$ C38	1.03	1.39	35	2.14	2.70	135
Reference: modified UDP *"iperf –udp -b 32000000"*						
C12 $\rightarrow \bullet$ SW41	32.00	32.00	0	27.12	38.60	2.6
SW43 $\bullet \rightarrow$ C38	2.19	1.39	57.5	2.14	2.70	10.5

Reference: default TCP. In this scenario, I observe unusual behavior of the network. The average bandwidth of the path connecting node $C10$ and $C38$ is low and does not exceed $1.3Mbps$ of stable traffic throughput. The measurements of the available capacity with *iperf* running in TCP mode provided flattened representation of the network that was relatively correctly represented by the predictive models. Both *SimQPN* and *OMNeT++generic* overestimated the throughput on link $C12 \rightarrow SW41$ due to the differences between TCP and UDP—the applied prediction mimics UDP more closely than TCP, thus the prediction error is high and should not be compared to the reference measurements directly (similarly to scenarios #4C and #5B).

The analysis of the capacity of link $SW43 \rightarrow C38$ is not affected by the TCP-UDP gap. For that link, the relative prediction errors are high as the models cope with low throughputs. *SimQPN* solver mispredicted the capacity by 12%, whereas *OMNeT++generic* by 95%. This corresponds to an absolute error of $0.15Mbps$ and $1.18Mbps$ respectively.

Reference: default UDP. Similar situation was observed for the second reference measurement. The maximal non-interrupted transfer measured by *iperf* limited the throughput to $1.03Mbps$. According to the documentation [ipe16], in its next internal iteration, *iperf* tried to transmit data with $2Mbps$ but it returned to the throughput of $1Mbps$ due to high packet losses. The relative and absolute prediction errors were higher, however they cannot be taken directly as reliable results without further analysis of the situation.

Reference: modified UDP. The third reference measurement represented the expected behavior of the network in terms of bottleneck location and performance. Note, that the DNI model used in this scenario was built to closely represent the expected behavior.

To measure the reference performance, I force *iperf* to send at least 4 megabytes of data per second ($32Mbps$) and increase the data rate by another $32Mbps$ if no packet drops occur. This allowed the traffic to arrive to $SW41$ at the full generation rate of $32Mbps$. Then, the switch $SW41$ forwarded each packet to the SDN controller by issuing a *packet-in* message. After approximately $8ms$, the controller replied with *flow-mod* containing the decision regarding the forwarding of the packet (the decision has not been stored in the flow table). As a result, each packet was delayed by at least $8ms$ (plus additional network interface processing delays) and forwarded to the destination in $C38$.

At the receiving end-point, *iperf* reported maximal throughput of $2.19Mbps$ with $130ms$ jitter and packet loss rate of 92%—$24\ 948$ datagrams were lost out of $27\ 223$ sent in total over ten seconds. The interface packet counters of the switch $SW41$ confirm this measurement reporting approximately ten times more packets received from $C12$ than forwarded to $SW00$.

The difference between the throughputs reported by the default and the modified configurations of *iperf* stem from its internal behavior regarding timeouts. *Iperf* measures the performance at the receiver side until the sender side notifies it that the experiment ends. However, the notification is significantly delayed by the switch and the SDN controller, so it arrives to the receiver later (possibly after several attempts caused by losses). This additional delay causes that an additional part of datagrams queued at the SDN controller arrive to the receiver and are included in the statistics. In this measurement, the receiving side of *iperf* reported arrival of $3.19MB$ in 11.7 seconds what results in average throughput of $2.19Mbps$.

For this reference traffic, the utilized network capacity has been ideally predicted by *SimQPN*, whereas *OMNeT++generic* mispredicted it with 2.6% relative error ($0.86Mbps$ absolute error). The relative prediction errors on link $SW43 \rightarrow C38$

were higher—57% relative and $0.8Mbsp$ absolute error for *SimQPN*, whereas for *OMNeT++generic* 10.5% relative and $0.23Mbps$ absolute.

7.3.7 Scenario #7: Network Load-balancing with ECMP

In this scenario, I demonstrate the load balancing capabilities based on Equal-Cost Multi-Path Routing (ECMP) [TH00, CWO+12] for the generated predictive models. I do not provide reference values and thus do not evaluate the prediction accuracy against the measurements conducted on real hardware. However, the selected scenarios represent realistic configurations used in big data centers in practice [RBP+11].

Note that SDN-enabled switches are not required for the ECMP scenarios if the devices implement ECMP directly. Nevertheless, SDN can be used to implement similar behavior for SDN-enabled switches that do not provide support for ECMP.

In ECMP routing, I assume that there are multiple paths between two switches supporting ECMP. The switches are configured to automatically load-balance the traffic between the redundant links. An example is presented in Figure 7.16.

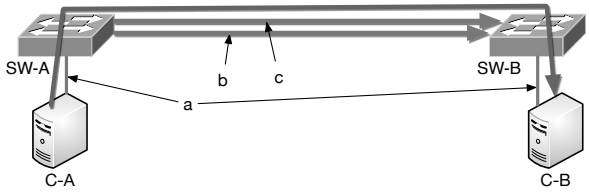


Figure 7.16: Scenario #7: Exemplary testbed. The values of a, b, c, and load balancing ratios are defined in the Table 7.8 for respective sub-scenarios.

Modeling

In this scenario, I model a single `Flow` between two applications deployed on nodes C-A and C-B. The network load balancing is defined by specifying the value of the *probability*< 1.0 parameter of respective `Direction` objects. An example is presented in Figure 7.17, where the link connecting switches SW-A and SW-B is load-balanced in a 50/50 ratio.

Results

I assume three sub-scenarios for evaluation of predicted network capacity between *SimQPN* and *OMNeT++generic*. In scenario #7A, the servers are connected to the switches over $a = 1Gbps$ links, whereas switches share two $b, c = 10Gbps$ connections. The link bandwidths are defined as: $a = 10Gbps$, $b = 6Gbps$, $c = 8Gbps$ for scenario #7B, and $a, b, c = 10Gbps$ for scenario #7C. Moreover, scenario #7C assumes load balancing with weights $70/30$, whereas the rest uses $50/50$.

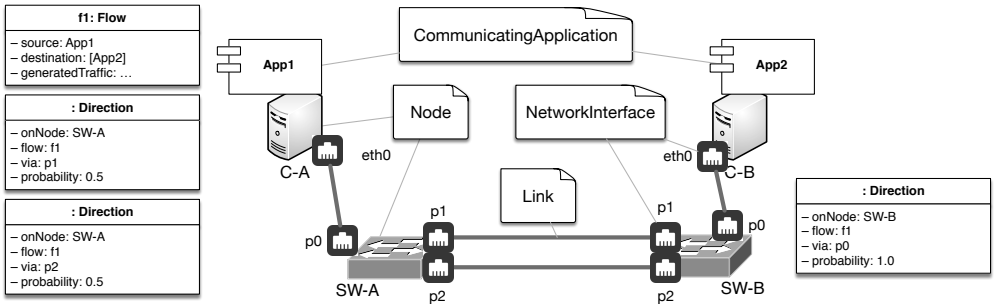


Figure 7.17: DNI model for the ECPM scenario.

The values are selected arbitrarily to represent wide variety of possible realistic configurations. I assume that the application deployed on node C-A transmits a $5GB$ file every second and 100 files are transferred during the experiment. The prediction results are presented in Table 7.8.

Table 7.8: Scenario #7: Predicted network capacity.

Measured port			OMNeT *Mbps* lCI	OMNeT *Mbps* uCI	QPN *Mbps*	Relative difference %
Scenario #7A: a=1Gbps, b,c=10Gbps, ratio 50/50						
C-A	$\rightarrow \bullet$	SW-A	925	957	924	1.8
SW-A.1	$\bullet \rightarrow$	SW-B.1	467	479	456	3.6
SW-A.2	$\bullet \rightarrow$	SW-B.2	462	479	468	0.6
Scenario #7B: a=10Gbps, b=6Gbps, c=8Gbps, ratio 50/50						
C-A	$\rightarrow \bullet$	SW-A	9113	9602	9240	1.3
SW-A.1	$\bullet \rightarrow$	SW-B.1	4556	4800	4632	1.0
SW-A.2	$\bullet \rightarrow$	SW-B.2	4557	4802	4608	1.5
Scenario #7C: a,b,c=10Gbps, ratio 70/30						
C-A	$\rightarrow \bullet$	SW-A	9113	9602	9240	1.3
SW-A.1	$\bullet \rightarrow$	SW-B.1	6380	6722	6473	1.2
SW-A.2	$\bullet \rightarrow$	SW-B.2	2733	2880	2768	1.4

Scenario #7A. In this scenario, the predicted throughput is limited by link C-A→SW-A and drops from $40Gbps$ to the maximal data throughput of about $940Mbps$. The solvers execute ECMP load balancing at the message level (one message has size of $5GB$), whereas in reality, the devices balance the communications at the level of packets or TCP connections. The predicted share of bandwidth between links SW-A.1 and SW-A.2 closely approximates the defined share ratio of $50/50$. The relative difference between the throughputs provided by the predictive models is low and do not exceed 3.6%. *SimQPN* underestimates the maximal theoretically available data throughput by about $16Mbps$.

Scenario #7B. In this scenario, I eliminate the bottleneck by replacing the $1Gbps$ link with a $10Gbps$ SFP+ one. Moreover, the redundant links between switches SW-A and SW-B have been also replaced. They offer now maximally $6Gbps$ and $8Gbps$ respectively. The relative difference between the predicted throughputs remains below 2%. The impact of unequal bandwidths on the load-balancing links has no influence on predicted throughput unless the capacity of a slower link is not exceeded.

Scenario #7C. In this scenario, I assume equal $10Gbps$ bandwidth for all links. The weights of the load-balancing algorithm are changed to simulate a $70/30$ share between the redundant links. The relative difference between predicted throughputs remains below 2%. Despite the message-based load-balancing and high data volumes in a message, the load balancing ratio in the predictions matches the modeled ratio of $70/30$.

7.3.8 Scenario #8: Server Load-balancing as Network Function

In this scenario, I demonstrate the modeling features of DNI and of the generated predictive models. Similarly to scenario #7, I do not provide reference values measured on real hardware. Although *L7sdntest* supports SDN-based load balancing for selected switches, the feature is currently under active development and it does not allow to conduct stable measurements with the assumed scenario setup. The scenario selected for demonstration represents realistic configurations used in nowadays data centers in practice [KK12].

In this scenario, I assume that six users request a file resource from server $C37$ every 5 seconds. The client components are deployed on nodes $C10 - C36$ and communicate with $C37$ over SDN-based network. Switch $SW43$ implements load-balancing, which is controlled from the SDN controller that is deployed on $C16$. Node $C38$ mirrors $C37$. SDN controller monitors the load on servers $C37$ and $C38$ and instructs switch $SW43$ to modify the flow rules to balance the distribution of requests in a round-robin fashion. The users are not aware of load balancing as the switch transparently modifies the requests on-the-fly. The experimental testbed and the configuration of reply flows are presented in Figure 7.18.

Modeling

In this scenario, I model traffic patterns that are not known a priori. It is not known beforehand whether a request will be redirected by $SW43$ and thus from where the reply will originate. DNI is unable to model such scenario completely due to the lack of support for modeling closed workloads. The prediction however, can be divided into two phases, so that the complete traffic information is available.

In the first phase, I model the requests issued by the users and the behavior of the load balancing algorithm. As a result, the mapping of user requests to the destination servers is obtained. Next, I model the reply flows knowing precisely the source node and starting time of each reply flow. This allow to overcome the

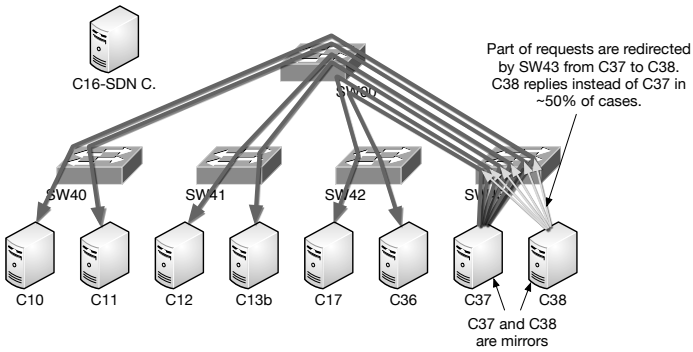


Figure 7.18: Scenario #8: Testbed and reply flows.

limitations of DNI and predict the performance of the network. I describe the modeling steps for both phases in the following.

Request phase. For each client application (deployed on nodes $C10 - C36$), I add a DNI `Flow` and set its respective source application in the client node. For each flow, there are two destination applications deployed on nodes $C37$ and $C38$. This means that the requests may be load-balanced in an intermediate node. Next, I define the node where the load balancing takes place by specifying the *probability* parameter in the respective `Direction` objects. I build two direction objects that reference the same flow and node, but different network interfaces (parameter *via*). Moreover, for each `Direction`, I specify the probability based on the load balancing ratios. In this scenario, I use round-robin load balancing, so both probabilities are set to 0.5. The modeling is presented graphically in Figure 7.19; there are two `Direction` objects specified for flow *f1* and node *SW43*.

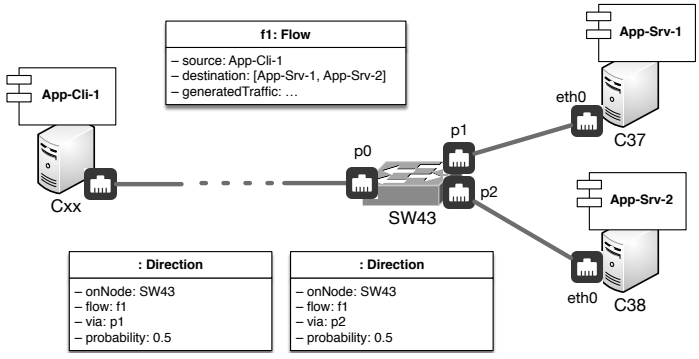


Figure 7.19: DNI model for SDN load-balancing scenario—request phase.

Reply phase. The modeling of the reply phase leverages the information from the request phase. A `Flow` is build for each pair of communicating applications, so that every server can respond to every client. The mapping of server replies to clients is obtained from the request phase, based on the end node where a request arrived. For example, if a request req_1 from node $C10$ arrived to server $C38$ in the first phase phase, a reply flow from $C38$ to $C10$ is built in the second phase.

Moreover, each reply flow is configured to start in a defined moment that matches the arrival time of the respective request. The file transfer is delayed by adding a `WaitAction` at the beginning of each traffic `Workload`. The value of the delay assigned to the `WaitAction` is equal to the request arrival time and increased by the value of *softwareLayersDelay* for each `EndNode`. For example, if a request $req_2 1$ from node $C13$ arrived to server $C37$ in the 32nd second in the first phase, the reply workload from $C37$ to $C13$ will be delayed by 32 seconds plus the *softwareLayersDelay* of node $C37$ (counting from the beginning of the experiment).

The modeling is presented graphically in Figure 7.20. There are 12 `Flow` objects, each for a pair of potential sender and receiver—six client applications (on nodes $C10 - C36$) and two server applications (on nodes $C37$ and $C38$).

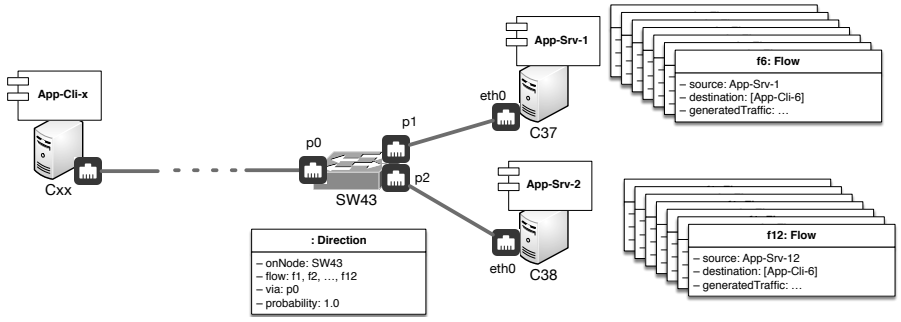


Figure 7.20: DNI model for SDN load-balancing scenario—reply phase.

Results

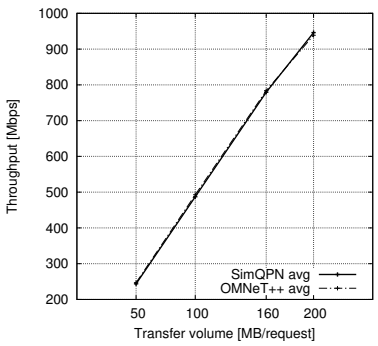
To demonstrate the prediction capabilities, I consider four simulation runs. I solve four models for arbitrarily selected size of the requested resources for the reply phase: $50, 100, 160, 200$ megabytes respectively. First, I transform the DNI model representing the request phase into both predictive models. I do not include the prediction results for this phase. Instead, I analyze the output of *OMNeT++generic* to extract the mapping of flows to the destination nodes and their arrival times. This data is used as an input for the reply phase, where I analyze the predictions delivered by both solvers. I build and transform five DNI models in total: one for request phase and four for the reply phase, each for four file resource sizes. The prediction results are presented in Table 7.9 and in Figure 7.21 for a selected link.

The relative difference between delivered predictions is low and does not exceed 2%. Both predictive models equally distribute the load between nodes $C37$ and

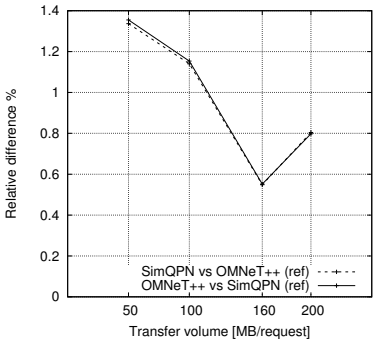
$C38$ in the request phase, thus the imbalance in reply phase is also low—maximally $7Mbps$ variation. For the file resources of size $100MB$ and higher, the models took advantage of the $10Gbps$ bandwidth of the link $SW43 - SW00$. This allowed to increase the network capacity and double the maximal throughput at the network bottleneck ($C37/C38 \rightarrow SW43$).

Table 7.9: Scenario #8: Predicted bandwidth on selected network interfaces in the reply phase.

Measured port			OMNeT *Mbps*		QPN *Mbps*	Relative diff. %	OMNeT *Mbps*		QPN *Mbps*	Relative diff. %
			lCI	uCI			lCI	uCI		
			File resource size 50MB				File resource size 100MB			
SW00	• →	SW40	140	189	164	0.5	293	365	327	0.5
SW00	• →	SW41	143	186	163	1.0	297	361	326	1.1
SW00	• →	SW42	145	184	163	1.0	298	359	326	0.8
SW43	→ •	SW00	435	553	489	0.9	923	1048	979	0.7
C37	→ •	SW43	214	280	244	1.3	455	531	487	1.1
C38	→ •	SW43	214	280	246	0.5	456	532	491	0.5
			File resource size 160MB				File resource size 200MB			
SW00	• →	SW40	483	566	524	0.1	578	673	633	1.2
SW00	• →	SW41	486	567	521	1.1	580	672	629	0.5
SW00	• →	SW42	484	561	521	0.3	572	666	630	1.8
SW43	→ •	SW00	1529	1608	1566	0.2	1843	1878	1892	1.7
C37	→ •	SW43	754	813	779	0.5	929	948	946	0.8
C38	→ •	SW43	761	819	786	0.5	918	942	946	1.8



(a) Throughput.



(b) Relative difference between predictions.

Figure 7.21: Scenario #8: Network throughput and difference between predictions for link $C37 \rightarrow SW43$.

The summary of the SDN-based evaluation scenarios is provided in Section 7.7. The analysis of the SDN hardware performance was published in [RSKK16], whereas the *DNI-to-QPN* transformation (with support for SDN) was published in [RSK16].

7.4 Flexibility of Performance Prediction

In this section, I evaluate the flexibility of the performance prediction process focusing on the duration of the solving. Based on a DNI model, multiple predictive models are obtained and each of them can be solved by at least one solver. In Section 5.6.2, I presented a method for selecting an optimal predictive model and its solver based on the limitations and features of the DNI model transformations. Here, I extend this analysis by evaluating solving times of obtained models. By offering multiple simulation models in parallel, I provide flexibility in trading-off between the modeling accuracy and the simulation overhead.

I evaluate the solving times in two parts. First, in Section 7.4.1, I evaluate solvers for non-SDN DNI models. This includes models generated by three transformations: DNI-to-*OMNeT++INET*, DNI-to-QPN, miniDNI-to-QPN that are solved with two solvers: *OMNeT++INET* and *SimQPN*. Next, in Section 7.4.2, I evaluate the prediction methods that support SDN-based networks. This includes two transformations: DNI-to-QPN and DNI-to-*OMNeT++generic* that are solved using *SimQPN* and *OMNeT++generic* solvers respectively.

Note that due to the limitations of the QPN-to-LQN transformation, I do not include analysis for the Layered Queueing Network (LQN) models. A separate analysis of QPN-to-LQN transformation and three LQN solvers is presented in Section 7.6.

7.4.1 Model Solving Time for non-SDN DNI Models

Along with evaluation of performance prediction accuracy in Section 7.2.3, I evaluated the performance of the generated simulation models, that is, the time needed to solve them. Depending on the situation, a less precise but quickly obtained result may be more valuable than precise but late predictions.

I examine the simulation duration in two scenarios. Firstly, in scenario #9, I varied the traffic intensity for the prediction accuracy scenario. The results of simulation time measurements presented in Section 7.4.1 refer to the experiment described in Section 7.2.3. Secondly, in scenario #10, I varied the size of the network by new adding servers, whereas the traffic intensity was constant.

Measurements and Environment

The duration of simulations are measured using the GNU *time* command on a non-virtualized server with Intel Xeon E3-1230 CPU, 16GB RAM, running under the control of 64-bit Ubuntu Linux 12.04 operating system. I compile *OMNeT++INET* in the release mode (*make MODE=release*) and exclude the TCL library (*NO_TCL=1 ./configure*). Simulations are run in the command line mode (*cmdenv*) without graphical overlay to enable repeatable conditions and automated measurement procedure. Measurements for *SimQPN* were conducted for solver version *2.1* running on top of Java in version *1.6.0_81*.

Several simulation durations for *OMNeT++INET* are estimated due to long simulation times. The estimation is conducted as follows. Firstly, I measure wall-clock time required to simulate 30 simulation-seconds of the experiment time (*OMNeT++* parameter: *sim-time-limit*). During this simulation period, I observe the internal metric of *OMNeT++* called *simulation-seconds per second* that describes the performance of the simulation. Secondly, I estimate the duration of the full simulation using extrapolation based on the real *SBUS* experiment duration and the measured duration for 30 simulation-seconds. For selected simulation runs, I verify the estimations by simulating the complete experiment length. The verification shows, that the estimations are precise—the estimation error does not exceed 1%.

Scenario #9: Scaling Traffic Intensity

In this scenario, I assume the network configuration and topology as presented in Section 7.2.3 in Figure 7.4. To recall, I deploy the camera components and the LPR components and configure the communication between the components according to the following plan: S2→VM4.1, S2→VM5.1, S3→VM6.1, and S3→VM5.1. I increase the picture generation rate by decreasing the think time between sending consecutive pictures to: 100, 50, 35, 20, and 10ms respectively. In the experiment, each camera sends 5000 pictures, each of size $2.5MB$.

The simulation durations are presented in Table 7.10 and depicted in Figure 7.22. Table 7.10 additionally presents the durations of the original experiment for reference.

Table 7.10: Scenario #9: model solution duration (in seconds) for growing traffic intensity.

Think time	SBUS (real)	OMNeT 30s	OMNeT full	QPN DNI	QPN mDNI
100ms	1136	92	666	17	3
50ms	670	175	3908	31	3
35ms	528	234	4118	48	3
20ms	416	351	4867	73	3
10ms	348	519	6020	222	3

The *OMNeT++INET* simulations execute slower than the QPNs—up to 100 minutes for $10ms$ think time. I observe exponential growth of the simulation time for *OMNeT++INET* and QPN for growing traffic intensity. The miniDNI QPN model is insensitive to the traffic intensity and offers constant simulation time of 3 seconds. The exponential growth of simulation duration for *OMNeT++INET* and *SimQPN* is caused by the increasing number of events/tokens in the simulation model. The *miniDNI* QPN model abstracts the traffic patterns in the transformation and thus maintains constant number of tokens and constant simulation time.

Scenario #10: Scaling Network Size

In scenario #10, I investigate the influence of the network size on the simulation duration. I assume a classical dumbbell topology with two directly connected

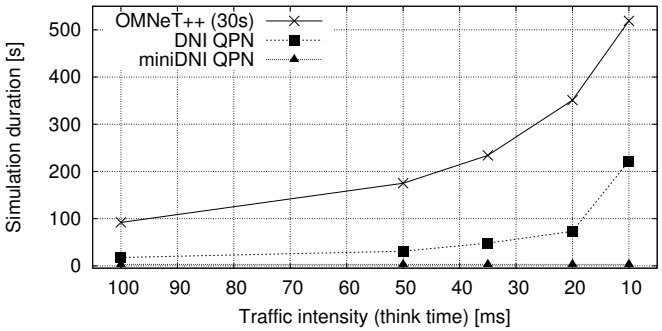


Figure 7.22: Scenario #9: model solving duration (in seconds) for growing traffic intensity.

switches and respective number of servers connected to each of the switches. I increase the number of servers connected to each switch while the traffic characteristics remain the same. Each node is configured to transmit a single $1.6MB$ picture per second. The experiment is ends when each node has finished the transmission of 5000 pictures. I generate 6 DNI models containing 5, 10, 20, 30, 40, and 50 nodes for each switch, that is, 10, 20, 60, 80, and 100 end nodes in total respectively. The simulation durations are presented in Table 7.11 and depicted in Figure 7.23. Additionally, in the column *Transformation*, I present the total time of running the five model-to-model transformations. The code of the transformations was not performance-optimized so the transformation durations can be further reduced as a part of future work.

Table 7.11: Scenario #10: transformation and model solving duration (in seconds) for growing network size.

Number of nodes	OMNeT 30s	OMNeT full	QPN DNI	QPN mDNI	Transformation
2×5	35	2953	21	5	25
2×10	65	5484	56	13	67
2×20	127	10716	183	36	149
2×30	195	16453	376	72	277
2×40	262	22106	630	122	446
2×50	334	28181	884	182	692

In scenario #10, I observe linear growth of simulation time for *OMNeT++INET*, which is caused by the linear growth of the number of events in the simulation engine. This observation confirms the results obtained by Weingartner et al. in [WvLW09]. Similar dependency can be observed for *miniDNI-QPN* where the number of tokens grows linearly with respect to the number of nodes. The *DNI-QPN* model experiences exponential growth of the simulation duration. Despite the linear nature of the *OMNeT++INET* run duration, the *DNI-QPN* outperforms the full length run of *OMNeT++INET* by the factor of 30. In scenario #10, the

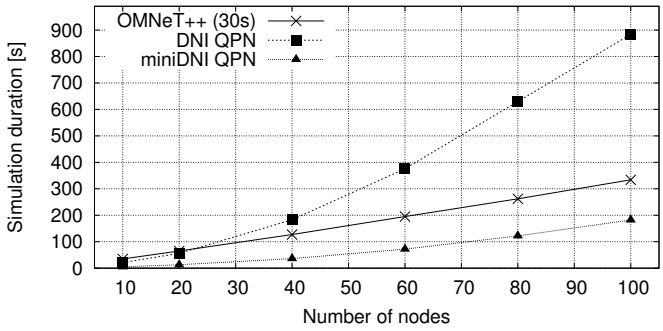


Figure 7.23: Scenario #10: model solving duration (in seconds) for growing network size.

miniDNI-QPN model is solved on average 300 times faster than the respective full-length *OMNeT++INET* simulation. Moreover, *miniDNI-QPN* requires four times less simulation time than the respective *DNI-QPN* model.

7.4.2 Solving Time and Memory Consumption for SDN-based DNI Models

During evaluation of performance prediction accuracy using scenarios from Section 7.3, I evaluated the performance of the generated simulation models, that is, the time needed to solve them.

I examine the simulation duration in for all SDN-based prediction scenarios (see Section 7.3). Based on scenario #4A and #4B, I evaluate the solving times for a constant network configuration and growing number of users. For scenarios #4C–#8, I evaluate solving times for various network setups including SDN and load balancing. Additionally to the solving time, I investigate solvers memory consumption.

Measurements and Environment

The duration and memory consumption of simulations are measured using the GNU *time* command on a non-virtualized server with Intel Xeon E5-2640v3 8-core CPU, 32GB RAM, running under the control of 64-bit Ubuntu Linux 14.04 LTS operating system. Memory consumption is estimated by measuring *minor page faults*. According to Linus Torvalds, each minor page fault truthfully estimates $4KB$ of consumed memory. The analysis for *OMNeT++generic* is conducted twice. First measurement is run for packet-level simulation (assumptions: *dataPayload* modeled as $1480B$, *packetOverhead* as $64B$), whereas in second run, I arbitrarily assume 100 times bigger *dataPayload* and *packetOverhead* to demonstrate simulation performance at a coarser level (denoted in the results as *big packet*).

Each measurement is repeated at least 10 times (30 times for most scenarios) due to long simulation durations in some cases. I compile *OMNeT++generic* in the release mode (*make MODE=release*) and exclude the TCL library (*NO_TCL=1 ./configure*). Simulations are run in the command line mode (*cmdenv*) without graphical overlay to enable repeatable conditions and automated measurement procedure. Both solvers utilize a single CPU core. Measurements for *SimQPN* were conducted for solver version *2.1* running on top of Java in version *1.8.0_91*.

Solving Time

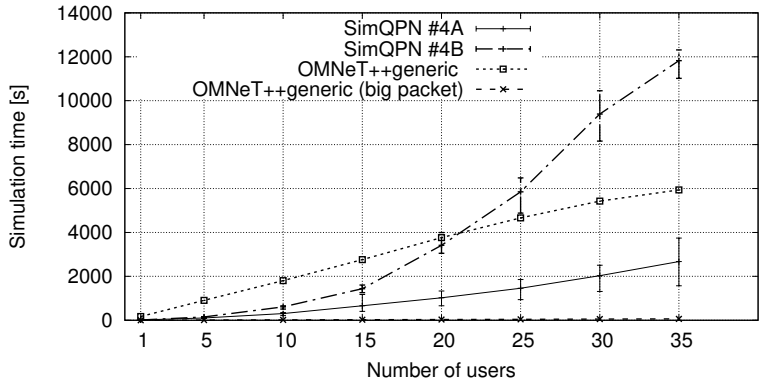


Figure 7.24: Simulation time required to solve scenarios #4A and #4B.

The results of solving time measurements are presented in Figure 7.24 for scenarios #4A and #4B. For *SimQPN*, the non-SDN models (scenario #4A) were solved up to six times faster than the SDN models (scenario #4B). The performance of the *SimQPN* simulation depends on the number of tokens and their colors. SDN scenarios modeled in QPN contain more tokes, as for each flow and each SDN switch, additional flows are generated for the communication of the switch with the SDN controller. Even if the scenario does not actively involve the SDN controller, the additional flows are generated and thus make the models bigger. These results show that there is potential for optimization of the generated QPN models in order to solve them in a shorter time. Such optimization would require to analyze if a token of a given color can be ever produced and if not, the respective firing modes can be removed from transitions.

The normally configured *OMNeT++generic* simulator (i.e., with normal packets size) perform identically for SDN and non-SDN setups. This is caused by the differences between the solvers; *SimQPN* analyzes the complete model as it is, whereas *OMNeT++* is a discrete event simulation. The unscheduled events do not influence the performance of the simulation in *OMNeT++*.

The solving with normally configured *OMNeT++generic* is slower than *SimQPN* for scenario #4A but faster than *SimQPN* for scenario #4B (for majority of cases).

The *OMNeT++* simulation with 100 times larger packets (denoted as *big packet*) outperforms the rest of the solvers. This is caused by drastic reduction of the events in the simulation engine, as every message is converted into smaller number of packets. Such configuration, however, influences the granularity of the reported results. For *dataPayload* of $150KB$, it is difficult to investigate the performance of the network for small traffic volumes, as every message smaller than $150KB$ will be treated as a message of size $150KB$. For the rest of scenarios (#4C–#8), I present the results in Table 7.12. For scenario #8 the duration of the reply phase was measured.

Table 7.12: Model solving duration (in seconds) for SDN scenarios.

	SimQPN			*OMNeT++generic*			*OMNeT++generic* big packet		
Scenario	min	average	max	min	average	max	min	average	max
#4C	51	60	67	4744	4821	5012	63	63	63
#5A	26	32	36	6083	6295	6741	81	82	85
#5B	8	9	10	1646	1704	1773	16	16	17
#6	5	6	7	126	128	131	1	1	1
#7A	4	4	5	900	918	1002	101	104	108
#7B	4	4	5	2449	2527	2595	107	108	110
#7C	7	7	8	2450	2520	2613	102	102	103
#8 50MB	78	103	133	1024	1041	1060	10	10	10
#8 100MB	80	106	190	2048	2093	2182	20	21	21
#8 160MB	78	107	125	3221	3285	3417	32	33	34
#8 200MB	72	128	195	4076	4085	4093	40	40	42

Here, *SimQPN* outperforms both versions of *OMNeT++generic* for scenarios #4C, #5, and #6. Interesting results are observed for scenarios #7 and #8. In scenario #7, *SimQPN* benefited form the ECMP load balancing because no SDN controller was involved and thus no overhead from additional tokens was incurred. *OMNeT++generic* however, was affected by high memory consumption that extended the simulation time over 100 seconds. The memory consumption of both solvers is analyzed in the following section.

For scenario #8, the situation was opposite. QPN model contained the SDN part of the network (as load-balancing was implemented in SDN) and thus the simulation took longer to complete (although no SDN controller was queried during solving) than the *OMNeT++generic* with large packets modification. The modified *OMNeT++generic* was able to cut down the simulation times to less than 40 seconds thanks to the reduced number of larger packets. The unmodified *OMNeT++* was the slowest and ran 10 to 30 times longer than *SimQPN*.

It is important to mention that the simulations were compared in terms of time-to-result. Internally however, *OMNeT++generic* conducts a single simulation run, whereas *SimQPN* repeats the simulation multiple times until the configured stopping criterion is reached. Additionally, *SimQPN* requires to simulate so-called warm-up period to bring the model to steady state. I demonstrated how the *OMNeT++generic* solver can be tuned using larger packets to minimize the

simulation time; similarly, the stopping criteria of *SimQPN* can be adapted (based on scenario) if less accurate results are acceptable.

Memory Consumption

In this section, I analyze the memory consumption of both solvers. The memory requirements play important role in solving as insufficient physical memory causes swapping and thus extends the solving time or even may not return a result due to error if the virtual memory size (physical+swap) is exceeded. The memory consumption measurements are presented in Figure 7.25 for scenarios #4A and #4B, whereas for scenarios (#4C–#8), I present the results in Table 7.13.

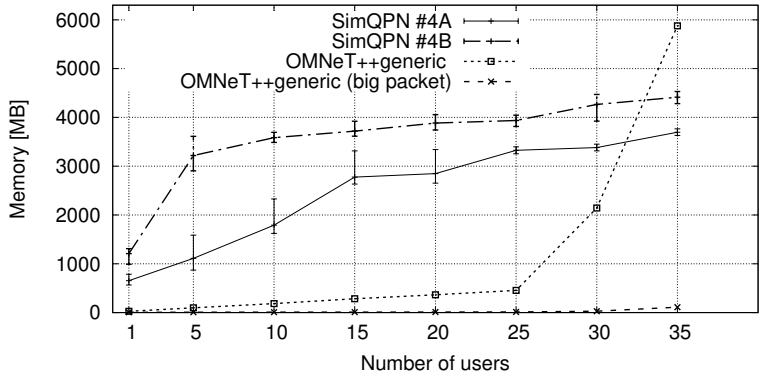


Figure 7.25: Memory consumption during solving scenarios #4A and #4B.

The consumption of memory for *SimQPN* is stable. Even for high number of users and network nodes, the simulation does not exceed $4.5GB$. The more tokens and more colors are used in the model, the more memory is consumed. Moreover, the memory consumption varies depending on a run up to $500MB$. This is caused by Java Virtual Machine (JVM) that manages the memory using garbage collector to clean up the no longer used memory for Java applications.

OMNeT++ (both: modified and unmodified version) consumes less memory than *SimQPN* for scenarios where the network capacity is not saturated. However, for saturated networks, *OMNeT++generic* queues the excessive traffic what causes excessive memory consumption—up to $6GB$ for scenarios #4A and #4B with simulated 35 users. The modified version of *OMNeT++generic* is also affected by queueing (exponential shape of the curve in Fig. 7.25) but to less degree.

High memory consumption and the simulation of UDP-like behavior of unmodified *OMNeT++generic* required to implement the interface queues using *drop-tail queue*. The queues were configured to drop excessive traffic if the utilization of the queue reached $5 \cdot 10^6$ packets. This may cause distortions in the measurements of throughput, so careful selection of *warm-up* and *cool-down* periods (i.e., the periods in which no measurements are taken) was required for the accuracy analysis.

The influence of the limited queue capacity have more impact on the memory consumption for the measurements that are presented in Table 7.13.

Table 7.13: Solver memory consumption (in megabytes) during solving of SDN scenarios.

Scenario	*SimQPN*			*OMNeT++generic*			*OMNeT++generic* big packet		
	min	average	max	min	average	max	min	average	max
#4C	1626	2131	2665	6093	6096	6103	281	281	281
#5A	668	791	901	4199	4207	4217	254	254	254
#5B	706	811	919	4671	4671	4671	51	51	51
#6	583	667	810	1044	1044	1044	6	6	6
#7A	589	669	758	3430	3437	3442	1376	1376	1376
#7B	536	647	787	26089	26092	26095	1093	1093	1093
#7C	547	636	790	26104	26109	26112	1093	1093	1093
#8 50MB	961	1323	1719	100	104	111	8	8	8
#8 100MB	931	1310	1723	176	179	185	8	8	8
#8 160MB	1128	1320	1773	543	571	724	11	11	11
#8 200MB	1086	1241	1667	2023	2035	2047	26	26	26

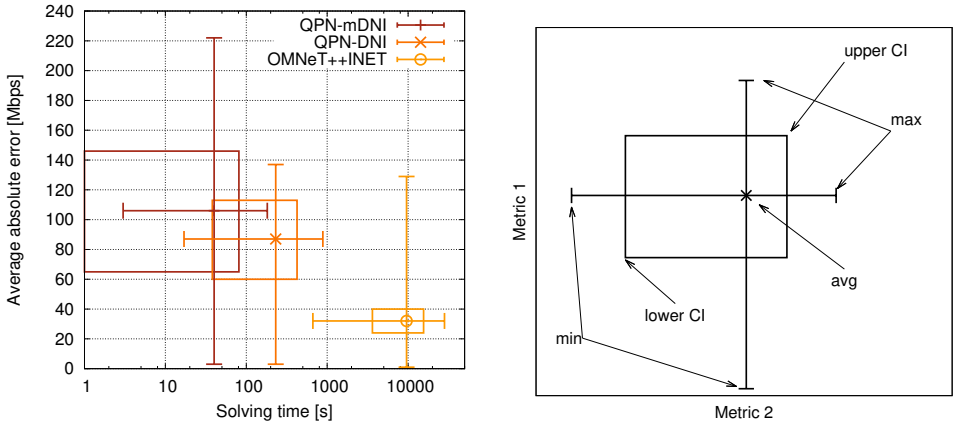
The excessive memory consumption of *OMNeT++generic* affected all scenarios containing bottlenecks and high transmission rates (scenarios #4C–#7C). Interesting situation can be observed for scenarios #7A–#7C (with ECMP load balancing). In scenario #7A, I observe a single bottleneck (of $1Gbps$) between the sender node C10 and the switch SW40. The frequent transmission of large file every second causes the network interface queue of the sender node to overflow which limited the memory consumption to about $3.4GB$. For scenarios #7B and #7C, the sender queue was no longer limiting the transfer (in scenario #7B and #7C, the bottleneck link was replaced from $1Gbps$ to $10Gbps$ one). The high volume of data was transferred to switch SW40 that implemented load balancing. However, the internal queue of the switch does not implement the *drop-tail* behavior and it accumulated the traffic until the interfaces of the switch could transmit it to the next node. Despite applying the *drop-tail* behavior to the queues in the network interfaces, the internal switch queue consumed up to $26GB$ of memory. This did not cause swapping, however the simulation duration was significantly increased. The modified version of *OMNeT++generic* confirms this behavior as the memory consumption results for scenarios #7A–#7C are much higher than in the other observed scenarios.

7.4.3 Discussion

Based on results presented in Section 7.2.3 (prediction accuracy analysis) and 7.4.1 (analysis of solving time), I show that abstraction of selected details in network performance models can lead to only minor prediction accuracy degradation but may significnatly accelerate the performance analysis.

I summarize, the solving time analysis in Figure 7.26. I present the dependency between absolute prediction error and solving time in Figure 7.26a. The figure

presents the average prediction accuracy with respect to the solving time for the three analyzed solvers. Additionally, I depict the minima, maxima, and the bounds of confidence intervals (see explanation in Fig. 7.26b). Based on the experiments presented in Section 7.2.3, I observe that the QPN models obtained form the miniDNI models deliver the highest prediction errors but the solving time is the shortest among the analyzed solvers. The QPN models obtained form the DNI models are solved longer but the predictions more accurate. The *OMNeT++INET* is the slowest solver, however, its accuracy is the highest.



(a) Absolute prediction accuracy vs. solving time.

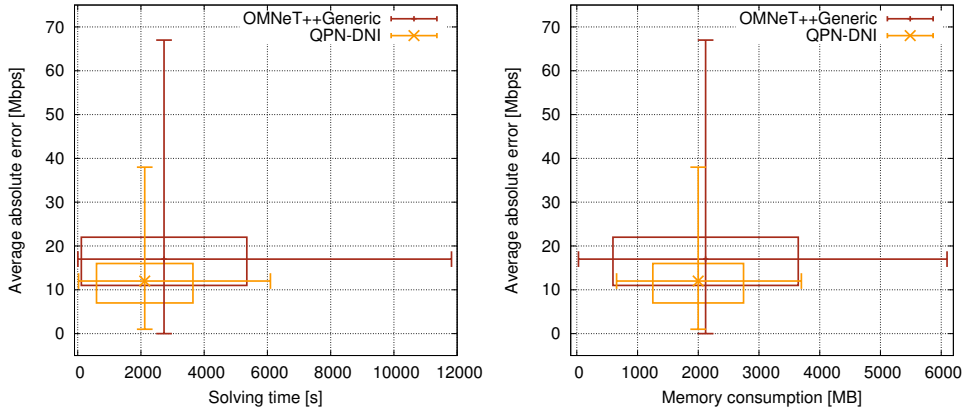
(b) Graph explanation.

Figure 7.26: Flexibility of performance prediction for non-SDN scenarios.

For the SDN-based networks, I analyzed two solvers: *SimQPN* and *OMNeT++generic*. Based on the evaluation conducted in Section 7.3, I observe that both predictive models provide good prediction accuracy with average relative prediction error about 5% for *SimQPN* and 3% for *OMNeT++generic*. The solvers vary in terms of solving time and resource consumption. They use a single CPU core for simulation, however both can be extended to run in parallel. With one exception, *SimQPN* solves the investigated models faster than *OMNeT++generic*—with speed-up between about 3 and 630. This results in similar absolute prediction errors—see Figure 7.27.

The figure presents the summarized results (scenarios #4–#6) after excluding the predictions affected by the TCP–UDP gap. *SimQPN* delivers more accurate predictions in shorter time, however, the difference between in the average absolute prediction accuracy is insignificant. For complex SDN-based network models, *OMNeT++generic* may provide shorter time-to-result with maximal speedup of 2, however at the cost of increased memory consumption—up to twice as much as *SimQPN* (see Fig. 7.27b). For extreme cases, when heavily-loaded network is analyzed, *OMNeT++generic* may cause packet losses due to high memory utilization

(up to $26GB$ in scenario #7). However, for large-scale scenarios, *OMNeT++generic* may be tuned to simulate the traffic with coarser granularity which allows to cut down the solving time by the factor of up to 200 and the memory consumption by the factor of up to 180.



(a) Solving time.

(b) Memory consumption.

Figure 7.27: Flexibility of performance prediction for SDN scenarios.

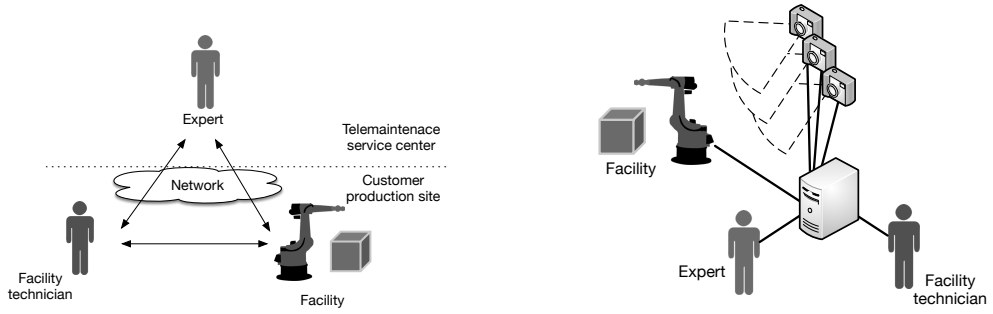
7.5 Traffic Model Extraction

In this section, I evaluate a secondary contribution of this thesis: the approach to traffic model extraction based on the Multi-Scale Decomposition (MSD) algorithm. The accurate modeling of traffic plays important role in optimizing the prediction accuracy as presented in Sections 7.2 and 7.3. On the other hand, the compactness of the DNI model speeds up the simulation as shown in Section 7.4. The evaluation of the MSD approach is conducted concerning the trade-off between both factors: accuracy and compactness of the obtained DNI model. The evaluation presented in this section was developed in cooperation with Viliam Šimko and published in [RSS+16].

In Section 6.2.3, I presented the extraction and optimization method using selected publicly available traces from online repositories to demonstrate the features of the approach. In this section, I evaluate the proposed approach using traces from the *robot telemaintenance* case study. The traces include mainly two types of network traffic: small control instructions or sensor readings and large flows of video streams from cameras that observe the work of a robot. The case study is described in section 7.5.1.

7.5.1 Robot Telemaintenance Case Study

The *Industry 4.0* initiated by German government (similarly to the *Industrial Internet* framed in the USA) comprises, among others, the introduction of the Internet to the manufacturing process. In the research project *MainTelRob* [ASF+15], different industry 4.0 approaches have been investigated for telemaintenance of a plant in a working production line [CA11]. Figure 7.28 depicts the project setting.



(a) Scenario overview.

(b) Network topology.

Figure 7.28: Robot telemaintenance case study. The remote expert and the local facility technician supervise the production facility.

The consumer production site consists of a six-axis Cartesian industrial robot, a two-component injection molding system, and an assembly unit. The plant produces plastic parts for electric toothbrushes. On the upper half of Figure 7.28a, there is a telemaintenance center from which an engineer—the *expert*—provides technical expertise to the local repair personnel—the facility technician. Next to the plant, the facility contains telemaintenance equipment: a computer, multiple cameras for video streaming, and an maintenance access device (e.g., a mobile tablet). The center and the facility are connected over the Internet. The main prerequisite is to provide the expert with a accurate overview about the situation in the facility. This insight can be offered by a specifically orchestrated combination of services: remote access to machinery data in combination with video streaming and communication services, for example, text chat and Voice-over-IP (VoIP). In addition, a visual augmented reality (AR) overlays inserted into the camera pictures or video view are used to provide guidance.

7.5.2 Traffic in the Telemaintenance Case Study

For the presented scenario, the proposed extraction method delivered insight in the patterns of the data exchange between the components of the system. Although the

main goal of the proposed method is to use the extracted models for performance prediction, here, the robot operator can observe the patterns of traffic to investigate anomalies. The analysis of the data exchange patterns may be used (additionally to the performance prediction) for the following purposes: (1) analyzing compact representation of control signals for debugging purposes (instead of, e.g., analyzing textual *wireshark* traces), (2) understanding the data exchange for network capacity planning.

The traces have been captured in a data center from where the devices were controlled. I have recorded *tcpdump* traces from the server that acts as a switch (see Fig. 7.28b). The server was replacing a typical data center switch to enable the traces collection. In a real environment however, the switches provide a monitoring mirroring port, which mirrors the complete traffic and forwards it to a server for analysis and debugging purposes. I capture the traffic traces from a 15 minutes monitoring period, which results in about $6.4GB$ of traffic traces.

7.5.3 Evaluating Model Compactness and Extraction Errors

I evaluate the model extraction accuracy by computing the relative errors for each pair of time intervals in the extracted model instance and the original trace. I stress, that the maximal model accuracy and the size of the extracted model are the trade-offs and the extracted model should not tend towards one of these extremes but provide a flexible means to select the required accuracy and size based on the application scenario. For the evaluation of the size of the extracted models, I assume that the original trace represents as many DNI `transmitActions` as many lines the original *tcpdump* file contains.

For evaluation, I selected 69 representative traffic traces from the robot telemaintenance scenario. Average trace contains about $22\ 000$ data samples. The traces were captured within a 15-minute measurement period. For each trace, I picked arbitrary parameters of the MSD extraction algorithm (smoothing 0, maximal number of clusters 25, cut-off for clustering 0.1, and intervals reduction parameter to 0.0004). I repeated the analysis of the 69 files 30 times and collected the following metrics:

1. median of relative error; the error was calculated between the original and extracted trace for every second (the aggregation function over one-second bins was sum)
2. mean relative error,
3. relative error of the total data transmitted in a trace,
4. compression of the modeled signal (it requires only x% of generators of the original signal).

I calculate the 5^{th}, 50^{th}, and 95^{th} percentiles for all metrics and present the results in Table 7.14. I divide the analyzed traces into four types based on the values of the obtained metrics.

Trace type 1. To the first type, I account the traces with the three relative errors lower than 10%. There are 43 traces of this type what shows that the proposed

Table 7.14: Results for the 69 analyzed traces divided into four groups.

Trace type	Traces	Relative median error %			Relative mean error %			Relative total data error %			Compressed to %		
		percentile			percentile			percentile			percentile		
		.05	.50	.95	.05	.50	.95	.05	.50	.95	.05	.50	.95
1	43	0	0	2.9	0	0.1	8.4	−1.2	0	1.7	0.4	0.8	2.3
2	13	0	0	7.5	12.6	16.4	36.7	−4.4	1.5	6.6	1.6	7.8	15
3	11	13.5	21.8	32.4	13	17.1	23	0	0.3	2.6	0.8	1.6	13.3
4	2	0.8	1.2	1.7	24.1	26.1	28.2	10.9	11.7	12.5	8.1	10.9	13.7
Trace types affected by extraction errors: 1) no error, 2) shifted peaks, 3) extraction parameters, 4) other.													

method can extract models of the most traces with good accuracy and good compression—the extracted models are 40 to 200 times smaller than original (compressed to 0.4%–2.3% of the original size).

Trace type 2. The traces assigned to the second type are characterized with good values of relative median error and relative total data error but higher values of the relative mean error. I investigated the discrepancies among the traces and the extracted models. The higher mean error rates are caused mainly by shift in a extracted signal—a workload peak shifted in time doubles the error: first because a real peak is not discovered and second because an artificial peak is produced whereas no peak exists in a real signal. An example of the described situation is depicted in Figure 7.29, where the x-axis represents time and the y-axis the volume of data in bytes over a second. Few peaks of type 2 were also influenced by non-ideal set of extraction parameters of extracted model (see trace type 3).

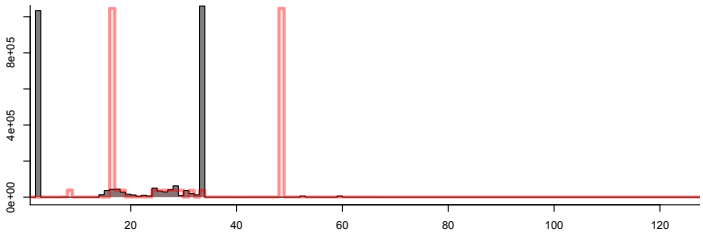


Figure 7.29: Example of a trace of type 2. Reference in dark gray and extracted model in red.

Trace type 3. The traces of the third type are characterized by low errors of relative total data error but higher relative median and mean. I name this type "extraction parameters" as the parameters of the extraction and optimization process were not ideal for the traces (note, I select parameter values arbitrarily). The higher errors are caused mainly by two factors: time-shifted peaks and outliers in the extracted signal caused by lower fit of the extracted model. I selected one trace affected by lower quality extraction and depicted it in Figure 7.30). I observe,

that the extraction was generally correct, but there are several periods where the data is not generated although the original trace behaves differently. This low extraction accuracy is caused by the fixed set of parameters selected for the method. One could optimize the accuracy by fine tuning the algorithm parameters or use less optimization for the compression. Although the median and mean errors vary from 13% to 30% the total amount of transmitted data is accurate and the compression ratio is good and varies between 7 and 125. Comparing the traces of type 2 against type 3, I observed that the ratio of time-shifted peaks to lower quality of extraction is higher in type 2; in type 3 the situation is opposite: there are more cases of lower quality extraction.

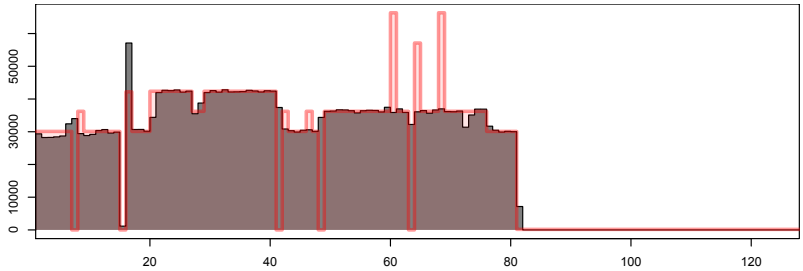


Figure 7.30: Example of a trace of type 3. Reference in dark gray and extracted model in red.

Trace type 4. The traces of type four do not fit to any other type. In this experiment, the relative mean error and total data error are above the arbitrarily selected 10% threshold. The relative median error is low and the compression ratio vary from 7 to 12. There were only two traces of type 4 in the dataset. I depicted a selected trace of type 4 in Figure 7.31). The errors are mainly caused by outliers introduced by suboptimal selection of extraction method parameters. I observe that the main shape of data trace was extracted correctly.

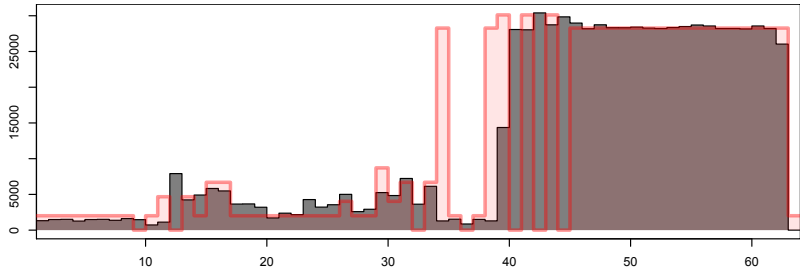


Figure 7.31: Example of a trace of type 4. Reference in dark gray and extracted model in red.

7.6 Transformation QPN-to-LQN and LQN Solvers

In this section, I evaluate the prediction accuracy and the nonfunctional properties of the LQN predictive models obtained in the QPN-to-LQN transformation presented in Section 5.5.2. The transformation is considered as a secondary contribution of this thesis. The transformations was published in [MRSK16] and later extended by Müller in [Mü16].

In contrast to previously presented model transformations, the *QPN-to-LQN* transformation is unable to transform the QPN models obtained from transforming the DNI models because the transformation it depends on the patterns in QPN model and there may exist multiple patterns representing a single construct, e.g., a loop. Moreover, the partial support of QPN models in the *QPN-to-LQN* transformation is caused by the nature of both modeling formalisms—QPN is more general than LQN. Due to that, some QPN scenarios cannot be represented by the LQN formalism.

Thus, to evaluate the *QPN-to-LQN* transformation, I use two QPN models as examples. Example #1 is a simple QPN model that serves as a toy-example. The model presented as example#2 was published in [KB03] and represents SPECjAppServer2001 [SPE02]—Java Enterprise Edition (Java EE) server application benchmark.

7.6.1 Example #1: Simple QPN Model

First example of a QPN model contains three queueing places as depicted in Figure 7.32. Each queueing place has a separate queue with deterministic processing time. The execution is looped to represent a closed workload with no think time.

I transformed it into LQN that is graphically represented in Figure 7.33. Transition t_1 was recognized as firing first based on the initial marking of the *start* place. Transition t_4 is not represented in LQN as it serves only to model the closed workload.

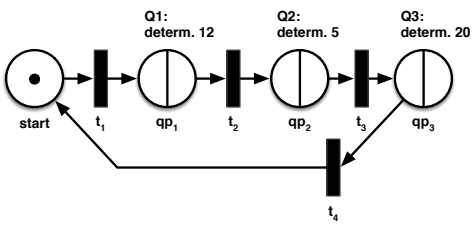


Figure 7.32: Example #1 QPN representation.

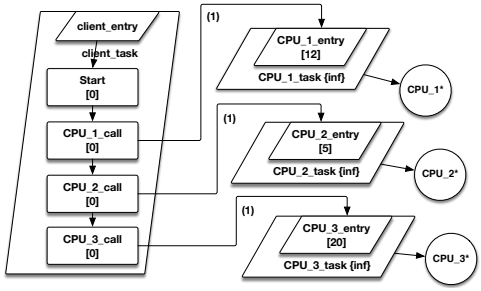


Figure 7.33: Example #1 LQN representation.

The example was solved using four solvers: *SimQPN* for QPN, and LQNS [FMW+09], LQSIM [FMW+09], LINE [PC13] for LQN. In this experiment, I show that the

transformation works correctly for simple QPN models. I examine the utilization of queueing places/processors and throughput. The results are presented in Table 7.15.

Table 7.15: Example #1: processor utilization and throughput.

Solver	Utilization			Throughput		
	CPU_1	CPU_2	CPU_3	CPU_1	CPU_2	CPU_3
SimQPN	0.324	0.135	0.541	0.0270	0.0270	0.0270
LINE	0.32432	0.13513	0.54054	0.027027	0.027027	0.027027
LQNS	0.32432	0.13513	0.54054	0.027027	0.027027	0.027027
LQSIM	0.32585	0.13768	0.53646	0.02738	0.02791	0.0277

The prediction of utilization and throughput was almost identical for all examined solvers. Taking the *SimQPN*'s prediction as a reference, LQSIM solved the model with the highest error mispredicting the utilization by maximally 3%.

I expected higher inaccuracy for LINE because the solving using fluid limits approximation is expected to work better for bigger models and provide higher errors for small. This issue seems to have been addressed by the authors of LINE as the results for small models are also good. I investigate a more complex model in the second example.

7.6.2 Example #2: SPECjAppServer2001

The system represented in this example is based on a Java Enterprise Edition (Java EE) server application benchmark (SPECjAppServer2001) [SPE02]. The application is modeled after a business consisting of four domains: customer domain (customer orders and interactions), manufacturing domain ("just in time" manufacturing operations), supplier domain (interactions with suppliers) and corporate domain (customer, product and supplier domain). The workload is claimed to be big and complex enough to represent a real-world enterprise system [SPE02]. In this scenario, the model is focused on the customer domain including four transaction types: NewOrder, ChangeOrder, OrderStatus and CustomerStatus. The system is deployed on two separate machines, one hosting the application logic and the other running a relational database. Besides the physical resources of the two machines (CPU and disk subsystem of the database), the model contains also logical resources, such as, the thread-pool of the application server, the connection and the process pool of the database server. A complete description of the model and its validation on a real system can be found in [KB03].

The transformed LQN model is depicted in Figure 7.35. The reference layer was selected based on transition t_1 and queueing place *Client*. The place *Client* represents the think time of the closed workload (parameter $Z = 200$ in *clien_7* task in the LQN) and the initial population of clients set to 80 (parameter $[80]$ in *clien_7* task in the LQN). Next, there are three layers that represent three nested critical sections that are limited by the thread pool, database connection pool, and database process poll. Once the activity *DBS-I_2* finishes, the process starts

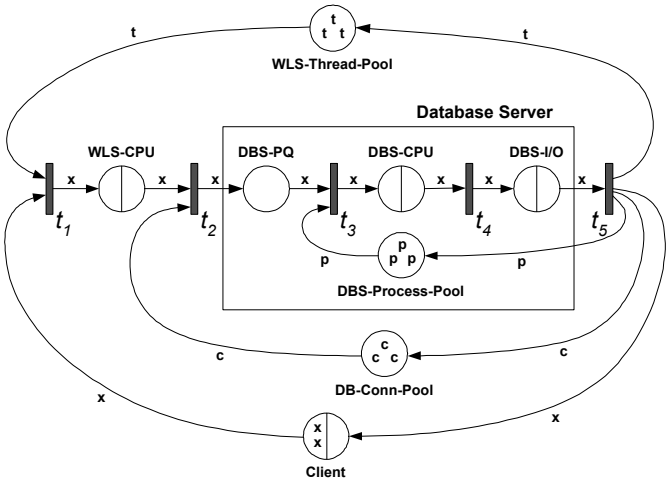


Figure 7.34: QPN representation of example #2. *WLS* stands for WebLogic Server, and *DBS* for a database server. Excerpted from [KB03].

again. I have examined prediction of utilization and throughput. The results are presented in Table 7.16.

Table 7.16: Example #2: processor utilization and throughput for 80 clients.

Solver	Utilization			Throughput		
	DBS-CPU	DBS-I/O	WLS-CPU	DBS-CPU	DBS-I/O	WLS-CPU
SimQPN	0.757	0.171	1	0.014	0.014	0.014
LINE	0.75714	0.17142	1	0.0142857	0.0142857	0.0142857
LQNS	0.75742	0.17149	1.00013	0.0142934	0.0142934	0.0142877
LQSIM	0.75525	0.17508	0.9828	0.01459	0.01425	0.01404

I assume *SimQPN* predictions as a reference. The utilization results show that the *WLS-CPU* is the bottleneck of the modeled system. All solvers reported nearly 100% utilization. LQNS overestimated the utilization, probably due to a rounding error, whereas LQSIM reported the utilization as 1.8% lower than the other solvers. The predicted throughput is affected by the bottleneck resource and is similar for all the solvers. LINE and LQNS overestimated the throughput by up to 2% relatively, whereas LQSIM reported up to 4% higher throughput than the reference.

7.6.3 Analysis of Solving Time and Memory Consumption

Additionally to the performance prediction accuracy, I investigated the solving time of the four examined solvers. I examined the LQN model from example #2. I varied the number of customers in the *clien_7* layer and solved the model for 1, 10, 40, and 80 clients. Then, I scaled up the modeled system and increased the

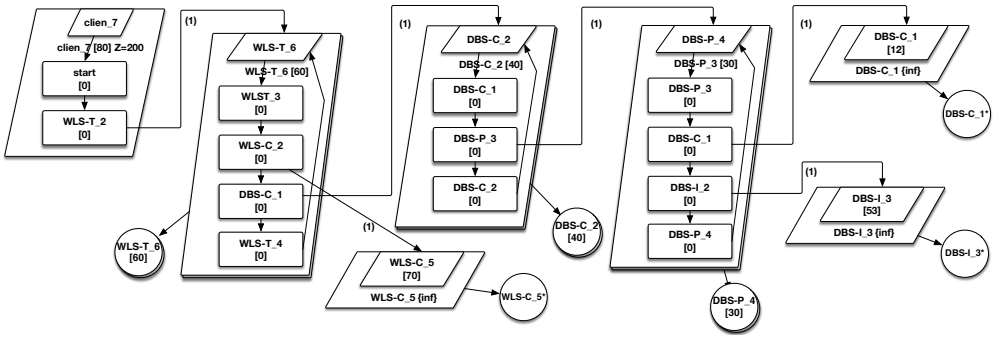


Figure 7.35: LQN representation of example #2.

quantities of the resources 2, 4, 8, 16, and 32 times with respect to the following configuration: 100 clients, 40 database connection pool size, 30 database processes, and 60 WLS threads. I denote these setups as $100 \times \{2, 4, 8, 16, 32\}$ respectively.

I executed the four solvers in Windows 7 virtual machine running on VirtualBox with assigned two CPUs and 4GB memory. Solving times of LQNS and LQSIM were measured using GNU *time* command and included such activities like starting the solver, reading input file, solving the model and writing output. The execution time of LINE was measured in the java code of the solver as LINE source code is [publicly available. *SimQPN* reports the running wall-clock time directly in the results. The time measurements for LQNS and LQSIM may a contain constant additive error because the *time* command includes also the initiation of the solver in the measurement. The results are presented in Table 7.17.

Table 7.17: Solving times of four solvers for varying number of clients in example #2.

Clients:	*SimQPN*	LINE	LQNS (linearizer)	LQNS (exact MVA)	LQSIM
1	$0.48s$	$0.44s$	$0.03s$	$0.05s$	$06.49s$
10	$0.51s$	$0.54s$	$0.09s$	$0.94s$	$1m28.95s$
40	$1.09s$	$0.63s$	$0.06s$	$1.35s$	$2m49.89s$
80	$1.26s$	$0.72s$	$0.06s$	$3.75s$	$4m05.62s$
100×2	$1.12s$	$0.96s$	$0.07s$	$9.67s$	$6m56.36s$
100×4	$2.54s$	$1.34s$	$0.10s$	$2m7.01s$	$12m23.42s$
100×8	$7.15s$	$2.09s$	$0.15s$	$10m8.00s$	$36m08.91s$
100×16	$12.98s$	$3.54s$	crash*	crash*	$100m50.59s$
100×32	$45.78s$	$6.37s$	crash*	crash*	$219m22.36s$
* out of memory ($> 4GB$)					

In this experiment, I expect the analytical solvers (LINE and LQNS) to outperform the simulations (*SimQPN* and LQSIM). The expectation was confirmed experimentally, as LQSIM was the slowest of the solvers and needed 4 minutes to solve the case with 80 clients and about 3.5 hours to evaluate the 100×32

scenario. *SimQPN* uses *batch mean* method which observes the simulation in the steady-state (i.e., after so-called warm-up period). The simulation stops when required precision is reached. The *SimQPN*'s solving time is low, although for big model instances, I observe a non-linear growth. LINE have outperformed the simulators and achieved linear growth of the solving time. Similar observation holds for LQNS which solved the models in linear time and was about 10 times faster than LINE. Unfortunately, LQNS requires much more memory to solve bigger models. During the solving of the 100×16 and 100×32 models, LQNS terminated immediately after the start due to insufficient of memory (reported error: *std::bad_alloc*).

Regarding the memory consumption, LQSIM uses a constant amount of memory during the simulation—about 75MB, 150MB, and 300MB for 100×8, 100×16, and 100×32 scenario respectively. LQNS consumes memory very fast and is unable to solve bigger models in the given configuration of the experimental machine. *SimQPN* has larger memory footprint due to Java virtual machine, however it scales good and can effectively handle simulations with up to several million tokens on a machine equipped with 16GB memory (for comparison, I observed ≈ 7300 tokens in experiment #2 for the 100×32 model). LINE uses Matlab libraries for computation, so there exists a memory footprint as well. It is difficult to observe precise memory consumption for LINE due to short solving times. More experiments are needed to provide an insight into memory complexity of LINE, however, I expect low consumption as LINE uses analytical methods.

7.7 Summary

In this chapter, I evaluated the primary and secondary contributions of this thesis, that is: novel modeling abstractions for virtualized network infrastructures in the form of the DNI meta-model, flexible model solving with model transformations, traffic model extraction (secondary contribution), and the transformation to layered queueing networks (secondary contribution). I evaluated the contributions according to the requirements specified in Section 1.2.1 and the goals specified in Section 7.1.

The evaluation was conducted in a realistic context of a data center network using two representative case studies. The first case study—*SBUS-PIRATES*—represents a distributed system of road traffic monitoring where pictures of cars are transmitted to the data center for further processing. The second case study—*Cloud file backup*—represents distributed cloud backup scenarios where file resources are stored on multiple mirror servers and the users may request to download or upload a file batch at anytime.

In the *SBUS* case study, I evaluated the modeling capabilities and prediction accuracy for native (non-SDN) data center networks. I used 11 scenarios in total to evaluate the modeling and performance prediction of four predictive models. The evaluation showed that **the DNI approach allows to capture the most important performance influencing factors and provide accurate performance predictions**

with prediction error up to 20% for uncalibrated models and 10% for partially calibrated models depending on the solving technique. Moreover, the changes in the experiment scenarios were correctly reflected in the predictions for all investigated cases. This confirms that **the DNI approach provides support for *what-if* performance analysis** in the data center network context. Moreover, the evaluation showed the importance of the model calibration using run-time measurements; this was a direct incentive to provide an approach to traffic model extraction, which I evaluates as a secondary research contribution.

The *Cloud file backup* case study served as source of 14 scenarios in total for evaluation of the modeling capabilities and prediction accuracy for SDN-based data center networks. In this context, I evaluated two predictive models obtained using two new model transformations: *DNI-to-*OMNeT++generic and *DNI-to-QPN*. The evaluation confirmed that calibrated DNI model can deliver prediction accuracy with errors below 4% depending on the modeling granularity and selected solver.

Moreover, I demonstrated that **the DNI models can accurately represent SDN-specific network features**. I showed how various SDN configurations impact the performance of the network. The SDN-specific network features cover: hardware and software flow tables in SDN switches, the SDN controllers, the ECMP-based load balancing, and the load balancing that shares the context of SDN with NFV. I observed, that the performance of a switch operating in non-SDN and SDN hardware mode is indistinguishable. Moreover, the misconfiguration of SDN—both, regarding the forwarding using software flow table and via the SDN controller—caused significant performance degradation. Nevertheless, such scenarios can be predicted correctly with the absolute prediction error not exceeding $10Mbps$. The load-balancing does not decrease the prediction accuracy, and even for high throughputs, the relative prediction errors are lower than 3%. The ratios of weighted load-balancing algorithms are reproduced by the predictive models correctly with less than 1% deviation. The evaluated scenarios confirm that DNI can represent network features that have the strongest influence on the network performance.

During the evaluation of performance prediction accuracy, I measured and analyzed the nonfunctional properties of the solvers that solved the transformed predictive models. I analyzed the solving time and resource consumption (memory and CPU cores) to provide an insight in the cost of solving. This allows to tailor the prediction performance process according to the user's needs and the evaluated scenario. I showed that **using less accurate predictive models may lead to small performance prediction degradation (of about 4%) whereas the solving time may be shortened by the factor of up to** 300. Moreover, the analysis of memory consumption allows to suggest an optimal solver for the user based on the resources available for solving. I formulated a set of recommendations regarding the selection of the feasible solvers based on the nonfunctional requirements of the user. Additionally, in Section 7.4.3, I presented an example of a DNI model that leverage the modeling flexibility of DNI where different modeling granularities (e.g., packet sizes) may lead to the savings of solving time and memory resources.

As a secondary research contribution, I addressed the problem of deterministic analysis of the network traffic traces for performance modeling purposes. I provided a flexible algorithm MSD for extracting traffic profiles from any time series (e.g., *tcpdump* traces). I showed that **the extracted traffic models can be optimized to reduce the size of the model but sill accurately model the characteristics of the original trace**. I showed that the model with reduced size can be as small as about 0.5% of the original while still accurately representing the original traffic characteristics (relative errors can be as low as 0.1%). The parametrized extraction causes that the procedure is flexible with respect to the demanded level of detail in the extracted models. The extraction process may be tuned to balance between extraction accuracy (more details in the model) and compactness of the extracted model (less details in the model). Furthermore, the extracted model in form of the *traffic generator* is not bound to DNI and may be used for general traffic analysis purposes.

The QPN-to-LQN transformation aimed at **bridging the gap between simulation models and analytical solving methods**. Unfortunately, the application of LQNs in DNI context is limited due to the limitations of the solvers currently available for the LQN formalism. The analysis of the nonfunctional properties of three investigated LQN solvers opens promising possibilities for future solving of the DNI models with good prediction accuracy (based on the investigated examples), short solving times, and low resource consumption of the solvers.

Chapter 8

Concluding Remarks

This chapter summarizes the contributions of this thesis and describes possible directions of future work.

8.1 Summary

In this thesis, I proposed the Descartes Network Infrastructure (DNI) approach to flexible run-time performance prediction in data center networks. In the flowing, I summarize the main contributions and their benefits.

To address the shortcomings of existing work (presented in Chapter 3), I proposed the **network performance abstractions for virtualized data center networks for run-time use**. The modeling abstractions are presented in form of the Descartes Network Infrastructure (DNI) meta-model—a new modeling language for performance modeling of modern data center networks.

The novel data center network modeling performance abstractions include:

(a) the DNI meta-model—a new approach for a medium-detailed descriptive modeling of data center networks, including modeling entities capable of representing SDN-based networks (Sections 4.1 and 4.2),
(b) the miniDNI meta-model—a minimal version of the modeling language for the coarsest modeling granularity (Section 4.3.2),
(c) the network deployment meta-model that serves as an integration interface for integration between the DNI and Descartes Modeling Language (DML) meta-models (Section 4.4.5).

The DNI modeling formalism **captures system behavior at run-time and offers different modeling granularities** in order to represent a system at the required level. By offering flexible medium-detailed modeling granularity, the modeling formalism abstracts too fine details and focuses on the relevant performance-influencing factors. The DNI meta-model has a generic character and is not bound to a defined network technology or a protocol. The meta-model offers flexibility by allowing to specify the system using different ways according to available data and user needs.

For coarse modeling of data center networks, I proposed **the miniDNI meta-model**, which provides a solid basis for coarse performance prediction and allows to abstract most details. The miniDNI meta-model is a minimal version of the DNI modeling language. Modeling with miniDNI can be applied for extreme cases

where a minimum amount of information about the modeled system is available. The minimal modeling language defines a lower boundary of the modeling granularity, for which feasible performance predictions with low accuracy can be delivered. Moreover, a DNI model can be automatically transformed to its miniDNI equivalent using the provided model transformation.

The DNI meta-model provides generic modeling entities to **support the majority of existing and future network technologies** including Software-Defined Networking (SDN) and Network Function Virtualization (NFV). The formalism provides generic support for NFV and medium-detailed support for SDN. The medium-detailed support for SDN supports novel prediction scenarios (e.g., using software SDN forwarding tables), at the same time abstracting the factors that have minor influence on the overall system performance.

The proposed **DNI network deployment model allows to connect the DNI and DML meta-models**. It allows mapping the network models onto software architecture-level descriptive models, so that a complete data center landscape can be modeled. The integration with DML allows capturing behaviors from the computing and software domain, which is not included in DNI model directly. The examples of such behaviors include: software bottlenecks, server virtualization and middleware overheads. The integrated descriptive models offer a richer view over the system and are prepared for future integration of transformations and solvers. The contributions described above were published in [RZK13, RKZ13, RK14a, RKTG15, RSK16].

DNI models have a descriptive nature, that is, they store information about the network infrastructure, however, without providing any means to predict the network performance under different conditions. To enable performance prediction, I provided a **flexible way to transform a DNI model into multiple predictive models using model-to-model transformations**.

Each model transformation (or a chain of multiple transformations) contributed in this thesis enables solving of a DNI model by generating a predictive model. The predictive models vary in size and complexity depending on the amount of data abstracted in the model and the transformation process.

I contributed *six* model transformations that transform DNI models into various predictive models based on the following modeling formalisms: (a) *OMNeT++* simulation, (b) Queueing Petri Net (QPN), (c) Layered Queueing Network (LQN). For these formalisms, multiple predictive models are generated (i.e., models having different levels of detail): (a) two for *OMNeT++*, (b) two for QPNs, and (c) two for LQNs. Moreover, some predictive models can be solved using multiple solvers leading to **up to *ten* different automated solving methods for a single DNI model**. In evaluation, I focused mainly on the *OMNeT++* and QPN predictive models as the applicability of the LQN formalism for data center networks is limited.

The main incentive for supporting various modeling formalisms is the difference in their characteristics, which is a prerequisite for the flexibility of the approach. Discrete-event simulation allows modeling the performance at a relatively fine-granular level including, for example, protocol-level details and the traffic at the

packet level. QPNs abstract low-level protocol information and support modeling of queueing and synchronization effects. The proposed set of available model transformations (and thus predictive models) is by no means exhaustive. The approach does not limit the amount and type of model transformations, so the set of supported transformations can be further extended.

The DNI solvers contributed in this thesis are evaluated in terms of solving times and the accuracy of network capacity prediction. The network capacity is expressed as the maximum possible network traffic that can be sustained, that is, the amount of consumed network bandwidth (or throughput). Despite the validation focuses on network throughput, the solvers deliver other performance metrics as well, for example: end-to-end transmission delays, packet losses. Future integration of the DNI and DML solvers will enable deeper insight into data center performance. Currently, pure network point-to-point delays (e.g., port-to-port latency) have limited applications in the context of a complex data center, where usually the end-to-end processing time of a service is more interesting than a low-level switch-to-switch latency.

The predictive models obtained using contributed model transformation were evaluated in terms of prediction accuracy and the resources (time, CPU, memory) required for the solving process. This allowed to characterize the nonfunctional properties of the solvers and thus provide a variety of solving methods that interpret a DNI model in different ways. For most of the evaluated DNI models, multiple solvers can be applied in parallel, so the user can select the most appropriate result. To support the user in selection of a feasible solving method, I proposed a **solver recommendation method** that suggest a set of feasible solvers based on the user requirements concerning the features modeled in DNI. The contributions regarding model transformations and flexible solving process were published in [RSK16, RKTG15, RK14a, RKZ13].

To build a DNI model manually, a modeler needs to acquire data about the network. This includes: topology, configuration and traffic. Some of this information may cumbersome to obtain and model in DNI manually. Mainly the traffic profiles are difficult to extract manually due to the high amount of data transmitted over the network in a short period. Fortunately, traffic models can be extracted automatically from traffic traces. Therefore, as a secondary contribution of this thesis, I provided **methods for supporting the user in the modeling process by semi-automated extraction of the network traffic part of a DNI model**.

In the proposed approach, the traffic profiles are represented as time-series of the data volume for a network interface. They contain only model-relevant data in the form of simplified time series. The new traffic profile abstraction and optimization method—called Multi-Scale Decomposition (MSD)—allows to balance between the size of the traffic model and the compactness of the extracted traffic profiles. The approach was published in [RSS+16].

The DNI approach was **evaluated in a realistic context of a data center network** using two representative case studies. The first case study—*SBUS-PIRATES*—represents a distributed system of road traffic monitoring, in which pictures of cars are transmitted to the data center for further processing. The second case study—

Cloud file backup—represents distributed backup scenarios, where file resources are stored on multiple mirror servers and the users may request to download or upload a file batch at anytime.

In the *SBUS* case study, I evaluated the modeling capabilities and prediction accuracy for native (non-SDN) data center networks. I used 11 scenarios in total to evaluate the modeling and performance prediction of four predictive models. The evaluation showed that the DNI meta-model allows capturing the most important performance influencing factors and **provide accurate performance predictions with prediction error up to** 20% **for uncalibrated models and** 10% **for calibrated models** depending on the solving technique. The changes in the modeled scenarios were correctly reflected in the predictions for all investigated cases. This confirms that the DNI meta-model is suitable for run-time *what-if* performance analysis in data center network context. Moreover, the evaluation showed the importance of the model calibration using run-time data.

The *Cloud file backup* case study served as source of 14 scenarios in total for evaluation of the modeling capabilities and prediction accuracy for SDN-based data center networks. There were two predictive models evaluated in this context. The evaluation confirmed that calibration improves the prediction accuracy providing low prediction errors (below 4% depending on the modeling granularity and selected solver). Moreover, I demonstrated how DNI model can represent SDN-specific network features and how various SDN configurations impact the performance of the network. The SDN-specific network features cover: hardware and software flow tables in SDN switches, SDN controllers, and load balancing scenarios that share the context with NFV. The evaluated scenarios confirm that DNI can properly represent network features that have the strongest influence on the network performance.

During the evaluation of performance prediction accuracy, I measured and analyzed the nonfunctional properties of the solvers that solved the transformed predictive models. I analyzed the solving time and resource consumption (memory and CPU cores) to provide an insight in the cost of solving. This allows tailoring the prediction performance process according to the user needs. I showed that **using less accurate predictive models may lead to small performance prediction degradation (of about** 4%**), whereas the solving time may be shortened by the factor of up to** 300**.** Moreover, the analysis of memory consumption allows suggesting an optimal solver based on the resources available for solving. I formulated a set of recommendations regarding the selection of the feasible solvers based on the nonfunctional requirements of the user. Additionally, I presented examples of the DNI models that leverage the flexibility at the level of DNI model, where different modeling granularities may lead to the savings of solving time and memory resources. The results of the evaluation were published in [RKZ13, RK14a, RKTG15, RSK16]. Evaluation results obtained in the *Cloud file backup* have not been published yet.

The DNI approach is the **first approach that flexibly bridges the gap between the high-level black-box performance models and the fine-grained, low-level network simulations**. I filled this gap by proving different gray-box models

that capture the internal system structure. This allows to investigate various system reconfigurations at different granularities simultaneously benefiting from the variety of solving times without interrupting the operation of the modeled system. The DNI approach provides benefits to the users by enabling the impact analysis of changes in: (a) the network structure, (b) the network configuration, and (c) the workload profile and intensity. Considering the benefits, the approach proposed in this thesis provides a solid basis for analysis and management of data center network resources in an automated resource management process that may reconfigure the system dynamically to adapt to the future expected conditions.

8.2 Open Challenges and Directions for Future Work

The results of this thesis offer a great potential for further research. In the following, I propose the possible directions of future work dividing them (similarly to the contributions) into research (primary and secondary), and technical.

8.2.1 Primary Research Directions

Integration of Solving Methods for DNI and DML

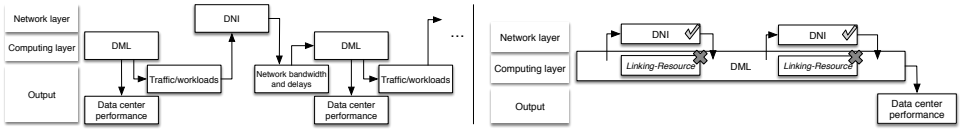
As presented in Section 4.4, the model solvers (i.e., transformations and resulting predictive models) for DML and DNI are separate and their integration is a challenging task. The complexity stems mainly from the wide scope of the integrated model—the automatically generated performance model would need to cover all system components: software, middleware, servers, virtualization, and network. The generated instances of predictive models (e.g., queueing networks or QPNs) may be too big to be solved in a reasonable time. To tackle this issue, I sketch potential approaches to solve the integrated models: iterative solving and modular solving.

The idea of iterative solving is to generate the predictive models for DML and DNI separately and then run the solving iteratively, so that the output of one model is used as input to the other and vice-versa. The iterative solution is repeated until the predicted performance metrics are stable. The iterative approach is presented conceptually in Figure 8.1a.

The modular solving approach assumes that the DML model is solved once, however the solving process of DML can be interrupted calling the DNI solver whenever the network behavior needs to be predicted for a given system state. The DNI solver provides network performance metrics back to the DML solver, and the solving is continued until the stopping-criterion of the DML solver is reached and the performance metrics are returned. Figure 8.1b depicts the modular solving approach.

The differences between modular and iterative solving can be summarized as follows. In the iterative solving approach, DML and DNI are solved separately multiple times and their partial solutions are used as in each subsequent iteration to refine the model parameters until the prediction accuracy converges. The models

are solved multiple times until the required accuracy (or other stopping criterion) is reached. In the modular solving approach, I assume that I build a single DML model where all instances of `LinkingResource` are replaced with small DNI models. In fact, there is only one "solving" process, however, in that process, the DNI solver is called multiple times.



(a) Iterative. (b) Modular.

Figure 8.1: Approaches to integrate DML and DNI model solving.

Fully integrated models would in fact provide a single language to describe complete data center landscape and increase the accuracy of performance prediction as the two models complement each other. Moreover, thanks to the flexibility of DNI and the tailored solving of DML [Bro14a], the integrated models may be tuned to put more focus on the computing part for computing-intensive workloads or on the networking part in traffic-heavy scenarios.

Complete integration of DNI and DML is expected to benefit from the more accurate prediction of network throughput and will also enable other performance metrics like, for example, end-to-end service response time. Additionally, DML complements DNI by providing the support for modeling the computing part of SDN controller applications and complex network functions deployed on a NFV node.

Another open challenge concerns the *automatic extraction and fine tuning* of the model parameters. It is generally desired that models are prefilled automatically and the modelers make relatively small corrections (with respect to building the model from scratch). Extraction and calibration of models is usually challenging (e.g., as shown in [RKTG15]), however calibration of two or more integrated models opens new questions that need to be addressed. Examples of such questions may include: (1) Are the network overheads already included in the DML model? (2) Is it possible to measure software and network overheads separately? (3) May calibration of DNI cause decalibration of DML and vice-versa?

Model-based Data Center Network Management using DNI

The topic of this thesis suggest directly the purpose for future applications of the DNI approach—network capacity management. The network capacity management is stated as one of the application areas of the approach (see Section 1.4). The approach can be used to analyze multiple network configurations and select the optimal one based on the forecast of traffic, optimization time horizon, and goals. Development of an autonomic performance-management framework for data centers is another example of possible future research directions.

The vision of *Self-Aware Computing Systems* [KKMZ17] assumes that the systems are able to "learn models capturing the knowledge about them selves and their environments (...) on an ongoing basis and reason using the models (for example, predict, analyze, consider, plan) enabling them to act based on their knowledge and reasoning (...) in accordance with higher-level goals, which may also be subject to change." DNI fits into this vision delivering the methods and tools to store the knowledge about the data center network and provide reasoning in form of performance prediction. Moreover, the models can be extracted at run-time to support the processes of online resource management in data centers [SFK+17].

Quality of Service in Networks

Providing world-wide access to a Quality of Service (QoS)-Internet is still an open challenge. However, in many private networks, delivering quality guarantees in terms of guaranteed bandwidth, prioritization of traffic flows, and fairness play important role nowadays. Internet of Things (IoT) may is example of application area where QoS in networks should not be neglected [ŚTN15, ŚJB+12].

Thanks to its generic character, DNI can support multiple application scenarios, including IoT. However, currently, the performance is modeled per device (physical or virtual) and not on a per-flow basis. Despite the possibility of modeling a set of virtual nodes (e.g., each handling a single traffic flow), providing the native support for per-flow QoS policies (e.g., fair scheduling, traffic prioritization, per-class bandwidth management) would greatly extend the applicability of DNI in the context of Future Internet (FI) and Next Generation Networks (NGN) and their application areas.

8.2.2 Secondary Research Directions

Automated Extraction of Complete DNI Models

In Section 6.1, I provided concepts of methods for automated extraction of complete DNI models from a running system. Further analysis of monitoring data sources and implementation of extraction tooling would drastically decrease the amount of work needed to build a DNI model. Such extraction suite would change the role of a human operator form the modeler to a supervisor, who eventually correct and calibrate the extracted models. This would provide an important milestone towards application of DNI in autonomic modeling, management, and analysis of self-aware computing and networking systems.

Traffic Model Extraction

I see several possible directions to improve the traffic model extraction approach presented in Section 6.2. First is the improvement of the automatic selection of extraction parameters, so that more traces can be analyzed with lower extraction errors. Second, the approaches to automatic model extraction should be compared

against the manual-built models crafted by humans. In such evaluation, the human-caused errors could be compared with the imperfections of the MSD extraction algorithm. Such evaluation is challenging and time consuming (mainly due to the involvement of the human factor), however the results would provide deeper insight of the model building process. Finally, the proposed approach was tailored (but not limited) to extract traffic models in a DNI-friendly format. The scope could be extended and the extraction of other traffic models could be added.

Optimization of the Transformations and Predictive Models

As presented in Section 7.4, the model transformations and the predictive models that they produce are not always optimal in term of size and their solving time. The transformations were built with focus on the functionality and correctness, however they were not optimized for performance. Moreover, the generated predictive models represent full information included in DNI (according to the level of abstraction implemented in the transformation). Yet, the complete set of information is not always needed to conduct tailored performance prediction. For example, the analysis of utilization of a selected networking device does not require to simulate the complete network, but only the traffic that directly influences the analyzed device.

Providing an approach to selectively ignore parts of the predictive model would decrease the solving time and consumption of resources needed for solving. The optimizations conducted at the transformation level would decrease the transformation time as well. Moreover, the tools for optimization of the predictive models (e.g., QPNs) could be applied to manually-built models as well providing the improvements to wider community.

Support for Analytical Predictive Models

DNI supports currently an arbitrarily selected set of model transformations and predictive models. Providing further model transformations would contribute towards the flexibility of the prediction. Especially, the support for analytical solving methods could be extended to support the state-of-the-art methods for analytical network performance prediction.

Support for Further Performance Metrics

The set of supported performance metrics can be extended as well to support, for example, end-to-end communication delays, packet losses, and jitter. In fact, the currently available predictive models deliver some of these metrics by default. Yet, they have not been evaluated for prediction accuracy in this context. Integration of DNI and DML should allow to predict the end-to-end delays. Measurements of further reference metrics would allow judge about the prediction accuracy delivered by DNI for those metrics.

Support for Closed Workloads

The support for modeling of closed workloads can be added to the DNI meta-model and the model transformations. As shown in scenario #8 in Section 7.3.8, the current support for closed workloads require to split the analysis into at least two separate steps: request and reply. This makes the analysis of closed workloads cumbersome, so providing a native support for closed workloads would extent the usability and applicability of the approach even more.

Modularization of Model Transformations

A DNI model can be transformed into with various predictive models. This however requires a separate transformation for each of them. While the transformations differ greatly in some aspects, for example, the configuration of the solver, other parts (e.g., the calculation of the performance parameters) are similar. It would therefore be reasonable to encapsulate parts of the transformations into reusable modules and extract them as independent units [SZC16]. Such modularization would improve the maintainability of the transformation code and limit the work required for developing new transformations to implementing only the parts that are specific for the output model. The modularization could be achieved, for example, using the *Reuseware* tool [HHJZ09].

8.2.3 Technical Directions

Further work on providing DNI tooling and an encapsulation of its components into a unified software product concerns mainly two directions. First, is related to the integration of DNI and DML. It is natural that the integrated approach should be available as a single framework, where the user can freely choose form the available modeling entities, transformations, solving methods, and tools for analysis of the results. Implementation of such a unified framework would widen the audience of the approach, as more users would be able to use it without manual installation and configuration procedure that is required currently.

Furthermore, the components of the approach could be wrapped and provided as web services to the users (i.e., prediction-as-a-service), so that the installation and maintenance of the software is shifted to the provider, while the users are offered a well defined application programming interface (API) to submit their models for solving and performance analysis.

List of Figures

1.1 Division of the performance model space into analytical and simulation models. Models with high level of detail (gray-striped area) are out of scope. . . . 3
1.2 DNI Approach: areas, contributions, and focus. . . . 13
1.3 Core benefit of the DNI approach: DNI supports a range of models with different modeling granularities without requiring expertise in each of them. . . . 14

2.1 Categorization of generic data center network virtualization techniques. 24
2.2 Comparison of typical forwarding pipelines of classical (non-SDN) and SDN devices. . . . 28

3.1 Related work divided into domains and relations between them. . . . 42

4.1 Root of the DNI meta-model. . . . 51
4.2 Core of the DNI meta-model: Entities. . . . 51
4.3 In the DNI meta-model, Dependencies are used to model numeric values and functions. . . . 52
4.4 DNI meta-model of network structure. . . . 53
4.5 Example of various `Node IPositions` combinations: a) intermediate, b) end, c) end+intermediate (receiver), d) end+intermediate (sender), and e) end+intermediate (forwarder). . . . 54
4.6 Relation between `NetworkInterface` and `NetworkProtocol` in the DNI meta-model. Dotted entities originate from the NetworkConfiguration part of the DNI meta-model (presented in Section 4.1.3). . . . 55
4.7 DNI meta-model of network traffic. . . . 56
4.8 Example of the DNI model presenting the workload. Depicted using the UML object diagram. . . . 57
4.9 DNI meta-model of network configuration. Entities in dotted boxes represent the entities form the other parts of the DNI meta-model. 58
4.10 Example presenting a fragment of a DNI instance using both: unofficial graphical notation and object diagram elements. The Direction *d* defines that half of the traffic from *App1* to *App2* should be routed via port *p1* on node *N2*. . . . 59
4.11 Meta-model of network structure with SDN entities included (in gray). Dotted entities represent other parts of DNI. . . . 60
4.12 Flow diagram for forwarding in a DNI Node. . . . 61
4.13 Flow diagram for processing a *packet-in* request in an SDN controller. 62

4.14 The DNI meta-model of network traffic with SDN entities included (in gray). The dashed entity `PerformanceSdnApplication` is a simplified representation of software component performance description that can be modeled using DML. 63
4.15 The DNI meta-model of network configuration with SDN entities included (in gray). 63
4.16 Flexibility in building DNI models. 64
4.17 miniDNI meta-model with SDN entities included (in gray). 66
4.18 Overview of DML's structure. Excerpted from [Bro14b]. 68
4.19 DML's Resource Landscape meta-model. Updated and redrawn based on [Hub14]. 69
4.20 DML's Resource Landscape meta-model: ConfigurationSpecification. Updated and redrawn based on [Hub14]. 69
4.21 Assembly Context in DML. Excerpted from [Bro14b]. 70
4.22 Different meaning of workloads in DML and DNI. 71
4.23 DML Example. Excerpted from [Bro14b]. 72
4.24 DML's FineGrainedBehavior. Excerpted from [Bro14b]. 72
4.25 Service behaviors in DML. Excerpted from [Bro14b]. 73
4.26 `CallParameter` in DML. Updated and redrawn based on [Bro14b]. . 73
4.27 Usage Profile Model in DML. Updated and redrawn based on [Bro14b]. 74
4.28 *NetworkDeployment* meta-model as extension of DML's *Deployment*. Respective parts are presented using distinguished colors. 75
4.29 Example DML model (from Fig. 4.23) mapped onto DNI. Mapping presented using object diagram. 76
4.30 Example DML model (from Fig. 4.23) extended by defining *Service Behavior Description* using `CoarseGrainedBehavior`. The deployment part of the example is abstracted for brevity. 77
4.31 Extracting DNI Traffic Flows based on DML model. Object diagram. Gray entities are highlighted to show the new objects with respect to the previous step (Fig. 4.29). 77
4.32 Example DML model (from Fig. 4.23 and 4.30) extended by defining *Usage Profile Model* (presents using graphical notation based on [BKR09]). The deployment and the Service Behavior Description parts of the example are abstracted for brevity. 78
4.33 Extracting DNI Traffic Workload based on DML model. Object diagram. Gray entities are highlighted to show the new objects with respect to the previous step (Fig. 4.31). 79
4.34 Extraction of the DNI traffic workload model from a DML model. . 80

5.1 Model transformations in the process of performance prediction. . . 85
5.2 Overview of models, transformations, and solvers. 86
5.3 In-place transformation building `FlowRoutes` based on `Directions`. Demonstrated using example presented formerly in Fig. 4.10. 91

5.4 Example demonstrating in-place DNI transformation removing `Branches` from `Workloads`. Left: workload with a branch, right: two workloads without branches. . . . 91
5.5 Notation used in QPN diagrams. Excerpted from [KBB+11]. . . . 92
5.6 Notation used in QPN transitions. . . . 93
5.7 QPN representation of network nodes and links. The internal structure of `Node` subnets is presented in Fig. 5.8. . . . 94
5.8 QPN representation of DNI's `Node`. Figure depicts the internal structure of a `Node` subnet place. . . . 94
5.9 Internal structure of transitions for non-load-balanced scenario. Token colors: *t* traffic, *f* flow-mod. . . . 95
5.10 Internal structure of the *SdnSwitching* subnet place. Token colors: *t* traffic, *f* flow-mod, *p* packet-in. . . . 95
5.11 Internal structure of the *switching*, *hwSdnSwitching*, and *swSdnSwitching* subnet places. . . . 96
5.12 QPN representation of DNI's `SdnController`. . . . 97
5.13 QPN representation of a `Node` with "60/40" load balancing for a single flow. . . . 98
5.14 QPN representation of a `Node` hosting virtual nodes. . . . 98
5.15 QPN representation of the TrafficSource. Token colors: gray workload execution token (WE) (workload-execution), white traffic. . . . 99
5.16 QPN representation of the a traffic workload example containing actions: wait, transmit, loop. Token colors: gray WE, white traffic. . 100
5.17 QPN representation of the workload loop action with three iterations. Token colors: gray *WE*, white traffic. . . . 101
5.18 QPN representation of the workload fork and branch actions. Token colors: gray *WE*, white traffic. . . . 101
5.19 QPN representation of miniDNI `Network` including `Links` and `Nodes` in *mDNI-to-QPN* transformation. . . . 105
5.20 Internal structure of miniDNI `Node` subnet place including `TrafficSource` in *mDNI-to-QPN* transformation. . . . 106
5.21 Example of a top-level *OMNeT++* network including modules: `StandardHost` (e.g., *relate3*), `Switch` (e.g., *sw1*), `VMM` (e.g., *relate4*). . . . 109
5.22 Internal structure of the custom `VMM` *OMNeT++* module. . . . 109
5.23 Example of a top-level *OMNeT++generic* network. . . . 111
5.24 Internal structure of network interface in *OMNeT++generic*. . . . 112
5.25 Transformation of queueing places with: (a) single place, queue, and color; (b) single place, queue, and two colors; (c) two places, single queue and single color. Processing times are modeled with the exponential distribution with a mean value defined in seconds. 118
5.26 Transformation of QPN ordinary places depending on context. . . . 119
5.27 Transformation of QPN ordinary places depending on context: branch in workload. . . . 119

5.28 Transformation of QPN transitions. QPN transition t_1 contain modes that consume and produce tokens on fire. LQNs representation is simplified (no processors) for brevity. 121
5.29 Exemplary QPN containing the fork and join pattern. 122
5.30 Exemplary LQN containing the fork and join representation of the QPN shown in Figure 5.29. 122
5.31 Example of a QPN loop representation with probabilistically modeled number of iterations. Excerpted from [BMB+15]. 123
5.32 Example of a QPN loop representation with deterministically modeled number of iterations (repetition of Fig. 5.17). 123
5.33 LQN loop representation with deterministically modeled number of iterations. 124
5.34 Example QPN containing a critical section. The pool contains, so maximally three tokens can enter the subnet. 124
5.35 LQN representation of the critical section corresponding to the QPN in Figure 5.34. 125
5.36 Example of QPN containing a second internal loop. 125

6.1 Model of a simple traffic generator. 139
6.2 A toy example: decomposition into traffic generators at different scales and amplitudes and the corresponding instance of a DNI traffic model. 140
6.3 Four examples of typical traffic dumps (with duration of 60, 10, 10, 10 minutes respectively). X-axis represents time in seconds, y-axis represents bytes transferred per second. 141
6.4 Normalized kernel density plots of transferred bytes from the example traffic dumps. 142
6.5 Overview of the extraction pipeline (rectangles represent data, ovals represent actions). 142
6.6 Multi-scale decomposition algorithm. Rectangles represent data, ovals represent actions. 144
6.7 Example decomposition explained step-by-step. 145
6.8 Decomposition example #1: 10 minutes of traffic without optimization. 147
6.9 Decomposition example #2: 10 minutes of traffic with moderate optimization. 148
6.10 Differences between OFF period durations in the simulation (upper part) and in the real system (lower part). 153

7.1 Experimental environment and network topology for *SBUS-PIRATES* case study. Dashed links are used for monitoring and measurements, solid links for experiment data traffic. 161
7.2 Scenario #1A: Confidence intervals for the mean throughput for think time 50–500ms (left, experiment A) and for 5–40ms (right, experiment B). 164

7.3 Scenario #1B: Confidence intervals for varying number of cameras and think time p. For Uperf, 1st and 3rd quartiles are shown additionally. . . . 165
7.4 Network topology used in the experiment. Dashed links are used for monitoring, solid links for experiment data traffic. Server $S1$ is the experiment controller. . . . 172
7.5 Scenarios #3A–#3E: relative prediction errors. Solid line compares two reference measurements using *SBUS* and Uperf. . . . 176
7.6 Experimental testbed used for SDN experiments. The gray links (connected to SW10) are disabled by Spanning Tree Protocol (STP) if not stated otherwise. . . . 181
7.7 Scenario #4A: Flows of server replies (denoted by arrows). . . . 184
7.8 Scenario #4A: Network throughput measured on link SW35→C39. . 185
7.9 Scenario #4B: Network throughput measured on link SW35→C39. The switches operate in SDN hardware forwarding mode. . . . 186
7.10 Comparison of reference throughputs for non-SDN (scenario #4A) and SDN hardware modes (scenario #4B). . . . 187
7.11 Measurement (and exemplary QPN-based prediction) of *HP 3500yl* performance in SDN Software mode for different settings of switching capacity. Excerpted from [RSK16]. . . . 188
7.12 Scenario #4B: Network throughput measured on link SW35→C39. The switches operate in the SDN software forwarding mode. . . . 189
7.13 Scenario #4C: Experimental testbed and file transfer flows. . . . 189
7.14 Scenario #5: Experimental testbed and file transfer flows. . . . 191
7.15 Scenario #6: Experimental testbed and file transfer flows. . . . 193
7.16 Scenario #7: Exemplary testbed. The values of a, b, c, and load balancing ratios are defined in the Table 7.8 for respective sub-scenarios. . . . 196
7.17 DNI model for the ECPM scenario. . . . 197
7.18 Scenario #8: Testbed and reply flows. . . . 199
7.19 DNI model for SDN load-balancing scenario—request phase. . . . 199
7.20 DNI model for SDN load-balancing scenario—reply phase. . . . 200
7.21 Scenario #8: Network throughput and difference between predictions for link $C37 \rightarrow SW43$. . . . 201
7.22 Scenario #9: model solving duration (in seconds) for growing traffic intensity. . . . 204
7.23 Scenario #10: model solving duration (in seconds) for growing network size. . . . 205
7.24 Simulation time required to solve scenarios #4A and #4B. . . . 206
7.25 Memory consumption during solving scenarios #4A and #4B. . . . 208
7.26 Flexibility of performance prediction for non-SDN scenarios. . . . 210
7.27 Flexibility of performance prediction for SDN scenarios. . . . 211
7.28 Robot telemaintenance case study. The remote expert and the local facility technician supervise the production facility. . . . 212

7.29 Example of a trace of type 2. Reference in dark gray and extracted model in red. 214
7.30 Example of a trace of type 3. Reference in dark gray and extracted model in red. 215
7.31 Example of a trace of type 4. Reference in dark gray and extracted model in red. 215
7.32 Example #1 QPN representation. 216
7.33 Example #1 LQN representation. 216
7.34 QPN representation of example #2. *WLS* stands for WebLogic Server, and *DBS* for a database server. Excerpted from [KB03]. 218
7.35 LQN representation of example #2. 219

8.1 Approaches to integrate DML and DNI model solving. 228

List of Tables

1.1 Denotation of the real and modeled entities used in the formal problem description. . . . 10
1.2 Further notation used in the formal problem description. . . . 10

2.1 Selected key differences between QPN and LQN formalisms. . . . 36

4.1 Comparison of the differences in modeling granularity between the DNI and the miniDNI meta-models. . . . 67
4.2 Overlapping entities in DNI and DML—data center structure. . . . 70

5.1 Matrix of DNI support of transformations and solvers. . . . 87
5.2 *DNI-to-miniDNI* transformation rules. . . . 104
5.3 *DNI-to-Omnet-INET* transformation rules. . . . 108
5.4 Key rules used in the QPN-to-LQN transformation. . . . 117
5.5 Selected features of the predictive models resulting from model transformations. . . . 127

7.1 Prediction errors and goodness of fit for scenarios #1A and #1B. . . 165
7.2 Scenarios #2A and #2B: Measured and predicted bandwidth between network nodes. . . . 166
7.3 Scenario #2C and #2D: Measured and predicted bandwidth between network nodes. . . . 168
7.4 Scenarios #3A–#3E: measured and predicted throughput. All values in mega-bits per second. . . . 175
7.5 Scenario #4C: Measured and predicted network capacity. . . . 190
7.6 Scenario #5: Measured and predicted network capacity. . . . 192
7.7 Scenario #6: Measured and predicted network capacity. . . . 194
7.8 Scenario #7: Predicted network capacity. . . . 197
7.9 Scenario #8: Predicted bandwidth on selected network interfaces in the reply phase. . . . 201
7.10 Scenario #9: model solution duration (in seconds) for growing traffic intensity. . . . 203
7.11 Scenario #10: transformation and model solving duration (in seconds) for growing network size. . . . 204
7.12 Model solving duration (in seconds) for SDN scenarios. . . . 207
7.13 Solver memory consumption (in megabytes) during solving of SDN scenarios. . . . 209
7.14 Results for the 69 analyzed traces divided into four groups. . . . 214

7.15 Example #1: processor utilization and throughput. 217
7.16 Example #2: processor utilization and throughput for 80 clients. . . 218
7.17 Solving times of four solvers for varying number of clients in example #2. 219

Acronyms

API application programming interface.

AR augmented reality.

ARP Address Resolution Protocol.

ASIC application-specific integrated circuit.

BCAM binary content-addressable memory.

BNF Backus–Naur form.

Bps bytes per second.

CGSPN Colored Generalized Stochastic PN.

CIM Common Information Model.

COTS commercial off-the-shelf.

CPU central processing unit.

DiffServ Differentiated Services.

DML Descartes Modeling Language.

DNI Descartes Network Infrastructure.

DoS denial of service.

DSML domain-specific modeling language.

DSP digital signal processing.

DWT Discrete Wavelet Transform.

ECMP Equal-Cost Multi-Path Routing.

EMF Eclipse Modeling Framework.

EMOF Essential Meta-Object Facility.

EOL Epsilon Object Language.

ETL Epsilon Transformation Language.

EVL Epsilon Validation Language.

FCFS first come first serve.

FI Future Internet.

FIFO first in first out.

GRE Generic Routing Encapsulation.

HFSC Hierarchical Fair Service Curve.

HP Hewlett-Packard.

HPE Hewlett-Packard Enterprise.

HTB Hierarchical Token Bucket.

HUTN Human Usable Textual Notation.

I/O input/output.

IETF Internet Engineering Task Force.

IntServ Integrated Services.

IoT Internet of Things.

IP Internet Protocol.

ISO/OSI International Standard Organization/Open Systems Interconnection.

Java EE Java Enterprise Edition.

JVM Java Virtual Machine.

LLTD Link Layer Topology Discovery.

LPR license plate recognition.

LQN Layered Queueing Network.

MAC Media Access Control.

MAN metropolitan area network.

MOF Meta-Object Facility.

MOL Method of Layers.

MPLS Multi-Protocol Label Switching.

MSD Multi-Scale Decomposition.

MTU maximum transfer unit.

MVA mean-value analysis.

NFV Network Function Virtualization.

NGN Next Generation Networks.

OCL Object Constraint Language.

OMG Object Management Group.

OVS Open vSwitch.

PCM Palladio Component Model.

PN Petri Net.

Pps packets per second.

QML/CS quality-modelling language for component-based systems.

QN Queueing Network.

QoS Quality of Service.

QPME Queueing Petri Net Modeling Environment.

QPN Queueing Petri Net.

QSFP+ quad small form-factor pluggable.

QSFP+ DAC quad small form-factor pluggable direct attach cable.

RSVP Resource Reservation Protocol.

SDL Specification and Description Language.

SDN Software-Defined Networking.

SDRAM synchronous dynamic random-access memory.

SFP+ small form-factor pluggable.

SFP+ DAC small form-factor pluggable direct attach cable.

SLA service level agreement.

SNMP Simple Network Monitoring Protocol.

STP Spanning Tree Protocol.

TCAM ternary content-addressable memory.

TCP Transmission Control Protocol.

TIME Transport Information Monitoring Environment.

ToR top-of-the-rack.

UDP User Datagram Protocol.

UML Unified Modeling Language.

Uperf Unified performance tool for networking.

VLAN Virtual Local Area Network.

VM virtual machine.

VMM virtual machine monitor.

VNFs virtualized network functions.

VoIP Voice-over-IP.

VPN Virtual Private Network.

WE workload execution token.

XMI Extensible Markup Language (XML) Metadata Interchange.

XML Extensible Markup Language.

XSD XML Schema Definition.

Bibliography

[Ada97] A. Adas. Traffic models in broadband networks. *Communications Magazine, IEEE*, 35(7):82–89, Jul 1997. [see pages 47 and 56]

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010. [see page 23]

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008. [see page 27]

[AMX15] I. Alsmadi, M. Munakami, and D. Xu. Model-based testing of sdn firewalls: A case study. In *Trustworthy Systems and Their Applications (TSA), 2015 Second International Conference on*, pages 81–88, July 2015. [see page 30]

[ANP+13] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou. An analytical model for software defined networking: A network calculus-based approach. In *IEEE Global Communications Conference (GLOBECOM)*, pages 1397–1402, Dec 2013. [see pages 2 and 43]

[ASF+15] Doris Aschenbrenner, Felix Sittner, Michael Fritscher, Markus Krauss, and Klaus Schilling. Teleoperation of an Industrial Robot in an Active Production Line. In *Proceedings of 2nd IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT)*, 2015. [see page 212]

[ASLG+14] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, William Snow, and Guru Parulkar. Openvirtex: A network hypervisor. In *Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, March 2014. USENIX Association. [see page 30]

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755 – 768, 2012. Special Section: Energy efficiency in large-scale distributed systems. [see page 1]

[Bal00] Simonetta Balsamo. Product form queueing networks. In *Performance Evaluation: Origins and Directions*, pages 377–401, London, UK, UK, 2000. Springer-Verlag. [see page 34]

[Bau93a] F. Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14–23, 1993. [see page 92]

[Bau93b] Falko Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14–23. IEEE, 1993. [see page 92]

[BBC+98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475 (Informational), 1998. [see page 24]

[BBE+08] Jean Bacon, Alastair Beresford, David Evans, David Ingram, Niki Trigoni, Alexandre Guitton, and Antonios Skordylis. TIME: An open platform for capturing, processing and delivering transport-related data. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas*, 2008. [see pages 22 and 159]

[BBE+12] M.F. Bari, R. Boutaba, R. Esteves, L.Z. Granville, M. Podlesny, M.G. Rabbani, Q. Zhang, and M.F. Zhani. Data center network virtualization: A survey. *IEEE Communications Surveys and Tutorials*, September 2012. [see page 1]

[BBGP10] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. Openflow switching: Data plane performance. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5, May 2010. [see page 43]

[BCH+11] Katherine Barabash, Rami Cohen, David Hadas, Vinit Jain, Renato Recio, and Benny Rochwerger. A case for overlays in dcn virtualization. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, DC-CaVES '11, pages 30–37. ITCP, 2011. [see page 25]

[BCR+09] Marco Bozzano, A. Cimatti, M. Roveri, J. Katoen, Viet Yen Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 121–130, July 2009. [see page 44]

[BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009. [see page 37]

[Bd05] J.P. Britton and AN. deVos. Cim-based standards and cim evolution. *IEEE Transactions on Power Systems*, 20(2):758–764, May 2005. [see page 44]

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003. [see pages 23 and 190]

[BdMIS04] S. Balsamo, A di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295–310, 2004. [see page 45]

[BGdMT98] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, New York, NY, USA, 1998. [see pages 34 and 36]

[BGdMT06] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006. [see pages 2 and 92]

[BH02] Ed Brinksma and Holger Hermanns. Lectures on formal methods and performance analysis. chapter Process Algebra and Markov Chains, pages 183–231. Springer-Verlag New York, Inc., New York, NY, USA, 2002. [see page 36]

[BHK12] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Modeling Parameter and Context Dependencies in Online Architecture-Level Performance Models. In *Proceedings of the 15th ACM SIGSOFT International Symposium on Component Based Software Engineering (CBSE 2012), June 26–28, 2012, Bertinoro, Italy*, June 2012. [see page 68]

[BHK14] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Elsevier Science of Computer Programming Journal (SciCo)*, Vol. 90, Part B:71–92, September 2014. [see page 68]

[BK98] Falko Bause and Pieter S. Kritzinger. Stochastic petri nets: An introduction to the theory. *SIGMETRICS Perform. Eval. Rev.*, 26(2):2–3, August 1998. [see page 35]

[BKLG14] A. Bianco, V. Krishnamoorthi, Nanfang Li, and L. Giraudo. Openflow driven ethernet traffic analysis. In *Communications (ICC), 2014 IEEE International Conference on*, pages 3001–3006, June 2014. [see page 44]

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. [see pages 45, 73, 78, and 234]

[Blo76] Peter Bloomfield. *Fourier Analysis of Time Series: An Introduction*. Wiley, 1976. [see pages 146 and 147]

[BM07] Simona Bernardi and José Merseguer. Performance evaluation of UML design with Stochastic Well-formed Nets. *Journal of Systems and Software*, 80(11):1843–1865, 2007. [see page 45]

[BMB+15] Fabian Brosig, Philipp Meier, Steffen Becker, Anne Koziolek, Heiko Koziolek, and Samuel Kounev. Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures. *IEEE Transactions on Software Engineering (TSE)*, 41(2):157–175, February 2015. [see pages 2, 116, 123, and 236]

[Bro14a] Fabian Brosig. *Architecture-Level Software Performance Models for Online Performance Prediction*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, July 2014. [see pages 37 and 228]

[Bro14b] Fabian Brosig. *Architecture-Level Software Performance Models for Online Performance Prediction*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, July 2014. [see pages 68, 70, 72, 73, 74, and 234]

[CA11] Soumitra Chowdhury and Asif Akram. E-Maintenance: Opportunities and Challenges. In *Proceedings of the 34th Information Systems Research Seminar in Scandinavia (IRIS)*, pages 68–81, 2011. [see page 212]

[CB10] N. M. M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Comput. Netw.*, 54(5):862–876, 2010. [see page 1]

[CCW+12] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Dr. Chunfeng Cui, Dr. Hui Deng, Javier Benitez, Uwe Micheel, Herbert Damker, Kenichi Ogaki, Tetsuro Matsuzaki, Masaki Fukui, Katsuhiro Shimano, Dominique Delisle, Quentin Loudier, Christos Kolias, Ivano Guardini, Elena Demaria, Roberto Minerva, Antonio Manzalini, Diego Lopez, Francisco Javier Ramon Salguero, Frank Ruhl, and Prodip Sen. Network Functions Virtualization (NFV), An Introduction, Benefits, Enablers, Challenges & Call for Action. SDN and OpenFlow World Congress, Darmstadt, Germany, 2012. Accessed 30.08.2016. [see pages 30 and 31]

[CDFH93] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *Computers, IEEE Transactions on*, 42(11):1343–1360, Nov 1993. [see page 92]

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006. [see page 85]

[CPSV08] Vittorio Cortellessa, Pierluigi Pierini, Romina Spalazzese, and Alessio Vianale. Moses: Modeling software and platform architecture in uml 2 for simulation-based performance analysis. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *Quality of Software Architectures. Models and Architectures*, volume 5281 of *Lecture Notes in Computer Science*, pages 86–102. Springer Berlin Heidelberg, 2008. [see page 45]

[CWD11] L. Cheng, C-L. Wang, and S. Di. Defeating network jitter for virtual machines. In *Proc. of the Fourth IEEE International Conference on Utility, Cloud Computing*, pages 65–72, 2011. [see page 25]

[CWO+12] Y. Cai, L. Wei, H. Ou, V. Arya, and S. Jethwani. Protocol Independent Multicast Equal-Cost Multipath (ECMP) Redirect. RFC 6754, October 2012. [see page 196]

[DDSG07] Isabel Dietrich, Falko Dressler, Volker Schmitt, and Reinhard German. SYNTONY: network protocol simulation based on standard-conform UML2 models. In *Proc. of the ValueTools '07*, pages 21:1–21:11, 2007. [see page 46]

[DLWJ08] Wolfgang E. Denzel, Jian Li, Peter Walker, and Yuho Jin. A framework for end-to-end simulation of high-performance computing systems. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 21:1–21:10, 2008. [see pages 2, 37, and 43]

[dWK05] Nico de Wet and Pieter Kritzinger. Using UML models for the performance analysis of network systems. *Comput. Netw.*, 49(5):627–642, 2005. [see page 46]

[Eco10] Eclipse Modeling Framework—Interview with Ed Merks. Online, `https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html`, Accessed 28.08.2016, 2010. [see pages 18 and 40]

[ER01] R. Callon E. Rosen, A. Viswanathan. Rfc 3031: Multiprotocol label switching architecture. IETF RFC 3031, 2001. [see page 26]

[Erl09] Agner K. Erlang. The Theory of Probabilities and Telephone Conversations. *Nyt Tidsskrift for Matematik*, 20(B):33–39, 1909. [see pages 32 and 33]

[Erl17] Agner K. Erlang. Solution of some Problems in the Theory of Probabilities of Significance in Automatic Telephone Exchanges. *Elektrotkeknikeren*, 13, 1917. [see page 32]

[FAOW+09] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *Software Engineering, IEEE Transactions on*, 35(2):148–161, March 2009. [see page 35]

[FHH02] A. J. Field, Uli Harder, and Peter G. Harrison. Network Traffic Behaviour in Switched Ethernet Systems. In *MASCOTS 2002, 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 32–42, October 2002. [see page 57]

[Fin13] Graham Finnie. Policy control & sdn: A perfect match? White Paper, 2013. Online, Accessed: 30.08.2016, `https://www.sandvine.com/downloads/general/analyst-reports/analystreport-heavy-reading-policy-control-and-sdn-a-perfect-match.pdf`. [see page 30]

[FM94] V.S. Frost and B. Melamed. Traffic modeling for telecommunications networks. *Communications Magazine, IEEE*, 32(3):70–81, March 1994. [see pages 33 and 47]

[FMW+09] G. Franks, P. Maly, M. Woodside, D.C. Petriu, and A. Hubbard. Layered Queueing Network Solver and Simulator User Manual. Manual, Real-Time and Distributed Systems Lab, Carleton Univ., Canada, 2009. [see pages 17, 35, 115, 116, 118, 124, and 216]

[FRZ14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, April 2014. [see pages 24 and 31]

[FWW13] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3):18:1–18:47, 2013. [see page 116]

[GABF06] Arijit Ganguly, Abhishek Agrawal, P. Oscar Boykin, and Renato Figueiredo. Wow: Self-organizing wide area overlay networks of virtual workstations. In *In Proc. of the 15th International Symposium on High-Performance Distributed Computing (HPDC-15*, pages 30–41, 2006. [see page 26]

[GBK14] Fabian Gorsler, Fabian Brosig, and Samuel Kounev. Performance Queries for Architecture-Level Performance Models. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, New York, NY, USA, March 2014. ACM. [see page 17]

[GGP12] Hadi Goudarzi, Mohammad Ghasemazar, and Massoud Pedram. Sla-based optimization of power and migration cost in cloud computing. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on*

Cluster, Cloud and Grid Computing (Ccgrid 2012), CCGRID '12, pages 172–179, Washington, DC, USA, 2012. IEEE Computer Society. [see page 1]

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability: A guide to NP-completeness*. W. H. Freeman, 1979. [see page 116]

[GLL+09] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, 2009. [see page 27]

[GMS07] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems. *J. Syst. Softw.*, 80(4):528–558, April 2007. [see page 45]

[Gor78] Geoffrey Gordon. The development of the general purpose simulation system (gpss). *SIGPLAN Not.*, 13(8):183–198, August 1978. [see page 37]

[GRSS12] Alexander Gouberman, Martin Riedl, Johann Schuster, and Markus Siegle. A modelling and analysis environment for lares. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, volume 7201 of *LNCS*, pages 244–248. Springer, 2012. [see page 44]

[GS08] A. Grzech and P. Świątek. Parallel processing of connection streams in nodes of packet-switched computer communication systems. *Cybernetics and Systems*, 39(2), 2008. [see pages 24 and 47]

[GSR10] Adam Grzech, Pawel Świątek, and Piotr Rygielski. Adaptive Packet Scheduling for Requests Delay Guaranties in Packet-Switched Communication Network. *Systems Science*, 36(1):7–12, 2010. [see page 24]

[GSTH08] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. *Fundamentals of Queueing Theory*. Wiley-Interscience, New York, NY, USA, 4th edition, 2008. [see pages 33 and 34]

[GWT+08] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):75–86, 2008. [see page 27]

[GYG13] A. Gelberger, N. Yemini, and R. Giladi. Performance analysis of software-defined networking (sdn). In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, pages 389–393, Aug 2013. [see page 44]

[HBK12] Nikolaus Huber, Fabian Brosig, and Samuel Kounev. Modeling Dynamic Virtualized Resource Landscapes. In *Proceedings of the 8th ACM SIGSOFT International Conference on the Quality of Software Architectures (QoSA 2012)*, pages 81–90, New York, NY, USA, June 2012. ACM. [see page 68]

[Hei07] Frank Heimburger. Performance Modelling of Java EE Applications using LQNs and QPNs. Master's thesis, Technische Universitaet Darmstadt, Germany, 2007. [see page 35]

[Her01] Ulrich Herzog. Formal methods for performance evaluation. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *Lectures on Formal Methods and PerformanceAnalysis: First EEF/Euro Summer School on Trends in Computer Science Bergen Dal, The Netherlands, July 3–7, 2000 Revised Lectures*, pages 1–37. Springer Berlin Heidelberg, 2001. [see page 36]

[HGJL15] Bo Han, V. Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, Feb 2015. [see page 31]

[HHAZ14] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. Flowguard: Building robust firewalls for software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 97–102, New York, NY, USA, 2014. ACM. [see page 30]

[HHJZ09] Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE*, 5560:39–82, 2009. [see page 231]

[HHK02] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1):43–87, 2002. [see page 36]

[HHKA14] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. *Concurrency and Computation - Practice and Experience, John Wiley and Sons, Ltd.*, 26(12):2053–2078, March 2014. [see page 84]

[HHSDK14] Evangelos Haleplidis, Jamal Hadi Salim, Spyros Denazis, and Odysseas Koufopavlou. Towards a network abstraction model for sdn. *Journal of Network and Systems Management*, pages 1–19, 2014. [see page 44]

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996. [see page 36]

[HLT09] Jens Happe, Hui Li, and Wolfgang Theilmann. Black-box performance models: Prediction based on observation. In *Proceedings of the 1st International Workshop on Quality of Service-oriented Software Systems*, QUASOSS '09, pages 19–24, New York, NY, USA, 2009. ACM. [see page 33]

[HP93] Peter G. Harrison and Naresh M. Patel. *Performance Modelling of Communication Networks and Computer Architectures*. Addison-Wesley, 1993. [see pages 41 and 42]

[HP13] HP. Hp SDN controller architecture. Technical report, Hewlett-Packard Development Company, L.P., September 2013. [see page 180]

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008. [see page 170]

[HsSL+14] Y. Han, S. s. Seo, J. Li, J. Hyun, J. H. Yoo, and J. W. K. Hong. Software defined networking-based traffic engineering for data center networks. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–6, Sept 2014. [see page 30]

[Hub14] Nikolaus Huber. *Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, July 2014. [see pages 69 and 234]

[Ing09a] David Ingram. PIRATES Data Representation. `http://www.cl.cam.ac.uk/research/time/pirates/docs/datarepr.pdf`, 2009. Accessed July 11, 2013. [see page 159]

[Ing09b] David Ingram. Reconfigurable middleware for high availability sensor systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 20:1–20:11, New York, NY, USA, 2009. ACM. [see pages 22 and 159]

[ipe16] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool. Online, `https://iperf.fr/`, Accessed September 2016. [see pages 180, 191, and 195]

[ITU00] SDL combined with UML. ITU-T Z.109, 2000. [see page 44]

[Jac88] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM. [see pages 167 and 170]

[Jar14] Michael Jarschel. *An Assessment of Applications and Performance Analysis of Software Defined Networking*. PhD thesis, May 2014. [see pages 2 and 30]

[JK09] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009. [see page 117]

[JOS+11] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Teletraffic Congress (ITC), 2011 23rd International*, pages 1–7, Sept 2011. [see pages 2 and 43]

[Kan09] Krishna Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Computer Networks*, 53(17):2939 – 2965, 2009. Virtualized Data Centers. [see page 1]

[KB03] Samuel Kounev and Alejandro Buchmann. Performance Modeling of Distributed E-Business Applications using Queueing Petri Nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003*, pages 143–155, Washington, DC, USA, March 2003. IEEE Computer Society. [see pages 92, 216, 217, 218, and 238]

[KB06] Samuel Kounev and Alejandro Buchmann. SimQPN—a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394, 5 2006. [see pages 35, 37, 92, and 115]

[KBB+11] Samuel Kounev, Konstantin Bender, Fabian Brosig, Nikolaus Huber, and Russell Okamoto. Automated Simulation-Based Capacity Planning for Enterprise Data Fabrics. In *4th International ICST Conference on Simulation Tools and Techniques*, pages 27–36, 2011. [see pages 92 and 235]

[KBM+16] Samuel Kounev, Fabian Brosig, Philipp Meier, Steffen Becker, Anne Koziolek, Heiko Koziolek, and Piotr Rygielski. Analysis of the Trade-offs in Different Modeling Approaches for Performance Prediction of Software Systems (Talk Extended Abstract). In *Software Engineering 2016 (SE 2016), Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. März 2016, Vienna, Austria*, Lecture Notes in Informatics (LNI), pages 47–48, Vienna, Austria, February 2016. GI. [see page viii]

[Ken53] David George Kendall. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953. [see page 34]

[KF13] H. Kim and N. Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, February 2013. [see page 30]

[KHBZ16] Samuel Kounev, Nikolaus Huber, Fabian Brosig, and Xiaoyun Zhu. A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. *IEEE Computer*, 49(7):53–61, July 2016. [see pages 12, 29, and 97]

[KJ13] Dominik Klein and Michael Jarschel. An openflow extension for the omnet++ inet framework. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, pages 322–329, ICST, Brussels, Belgium, Belgium, 2013. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). [see page 43]

[KK12] M. Koerner and O. Kao. Multiple service load-balancing with openflow. In *2012 IEEE 13th International Conference on High Performance Switching and Routing*, pages 210–214, June 2012. [see pages 30 and 198]

[KKMZ17] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors. *Self-Aware Computing Systems*. Springer Verlag, Berlin Heidelberg, Germany, 2017. [see page 229]

[KL12] J.M. Kunkel and T. Ludwig. Iopm – modeling the i/o path with a functional representation of parallel file system and hardware architecture. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 554–561, Feb 2012. [see page 46]

[KMF04] Thomas Karagiannis, Mart Molle, and Michalis Faloutsos. Long-range dependence: Ten years of internet traffic modeling. *IEEE Internet Computing*, 8(5):57–64, 2004. [see page 57]

[KO01] Ingemar Kaj and Jörgen Olsén. Throughput modeling and simulation for single connection tcp-tahoe. In Nelson L.S. da Fonseca Jorge Moreira de Souza and Edmundo A. de Souza e Silva, editors, *Teletraffic Engineering in the Internet Era*, volume 4 of *Teletraffic Science and Engineering*, pages 705–718. Elsevier, 2001. [see page 46]

[Kou05] Samuel Kounev. *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction*. PhD thesis, Technische Universität Darmstadt, Germany, December 2005. [see page 35]

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67(8):634–658, August 2010. [see page 45]

[KPK15] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. What You Need to Know About SDN Flow Tables. In *Proceedings of the 16th International Conference on Passive and Active Measurement*, pages 347–359, 2015. [see pages 7, 20, and 29]

[KSG+09] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM. [see page 47]

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. [see pages 7 and 84]

[KWS+13] Xiangxin Kong, Zhiliang Wang, Xingang Shi, Xia Yin, and Dan Li. Performance evaluation of software-defined networking with real-life isp traffic. In *Computers and Communications (ISCC), 2013 IEEE Symposium on*, pages 000541–000547, July 2013. [see page 44]

[LAD+14] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, GehanM.K. Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software & Systems Modeling*, pages 1–38, 2014. [see page 85]

[LCE10] Hui Li, Giuliano Casale, and Tariq Ellahi. Sla-driven planning and optimization of enterprise applications. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 117–128, New York, NY, USA, 2010. ACM. [see page 1]

[LGS+15] Stanislav Lange, Steffen Gebert, Joachim Spoerhase, Piotr Rygielski, Thomas Zinner, Samuel Kounev, and Phuoc Tran-Gia. Specialized Heuristics for the Controller Placement Problem in Large Scale SDN Networks. In *27th International Teletraffic Congress (ITC 2015)*, pages 210–218, Ghent, Belgium, September 2015. [see page viii]

[LHY12] Fei Liu, M. Heiner, and Ming Yang. An efficient method for unfolding colored Petri nets. In *Proceedings of the 2012 Winter Simulation Conference (WSC)*, pages 1–12, 2012. [see pages 116 and 117]

[Lju01] L. Ljung. Black-box models from input-output measurements. In *Instrumentation and Measurement Technology Conference, 2001. IMTC 2001. Proceedings of the 18th IEEE*, volume 1, pages 138–146 vol.1, May 2001. [see page 33]

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984. [see page 34]

[MAB+08a] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008. [see page 4]

[MAB+08b] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008. [see page 27]

[Mal08] Stphane Mallat. *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way*. Academic Press, 3rd edition, 2008. [see page 143]

[Man65] B. Mandelbrot. Self-similar error clusters in communication systems and the concept of conditional stationarity. *IEEE Transactions on Communication Technology*, 13(1):71–90, March 1965. [see page 33]

[MDA04] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. [see page 157]

[MIK+13] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Piotr Rygielski, Jason Ding, Walfredo Cirne, and Florian Rosenberg. Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios. Technical Report SPEC-RG-2013-001 v.1.0.1, SPEC Research Group - Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, May 2013. [see page ix]

[MIK+16] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Diane E. Mularz, Jonathan A. Curtiss, Jason J. Ding, Florian Rosenberg, and Piotr Rygielski. CUP: A Formalism for Expressing Cloud Usage Patterns for Experts and Non-Experts. *IEEE Cloud Computing*, 2016. To Appear. [see page vii]

[MLP+13] Y. Mei, L. Liu, X. Pu, S. Sivathanu, and X. Dong. Performance analysis of network i/o workloads in virtualized data centers. *IEEE Transactions on Services Computing*, 6(1):48–63, 2013. [see page 1]

[MOF14] OMG's Meta-ObjectFacility, 2014. [see pages 18 and 40]

[MRSK16] Christoph Müller, Piotr Rygielski, Simon Spinner, and Samuel Kounev. Enabling Fluid Analysis for Queueing Petri Nets via Model Transformation. *Electronic Notes in Theoretical Computer Science*, 327:71–91, 2016. The 8th International Workshop on Practical Application of Stochastic Modeling, {PASM} 2016. [see pages vii, 17, 87, 115, and 216]

[MST+05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM. [see page 1]

[MTMC99] Andreas Mitschele-Thiel and Bruno Müller-Clostermann. Performance engineering of SDL/MSC systems. *Comput. Netw.*, 31(17):1801–1815, 1999. [see page 46]

[MV16] Klaus Müller and Tony Vignaux. SimPy: Discrete Event Simulation for Python. Online, 2016. Accessed 30.08.2016. [see page 37]

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006. [see page 85]

[Mü16] Christoph Müller. Fluid Analysis of Queueing Petri Nets Models using Model-to-Model Transformations. Master Thesis, University of Würzburg, Am Hubland, 97074 Würzburg, Germany, April 2016. [see pages 17 and 216]

[Net11] Juniper Networks. Network simplification with juniper networks virtual chassis technology. Whitepaper, 2011. [see page 25]

[Omn16] INET Framework—An open-source OMNeT++ model suite for wired, wireless and mobile networks. Online, `https://inet.omnetpp.org/Introduction.html`, Accessed 14.07.2016, 2016. [see pages 2 and 107]

[ope11] Openvswitch. Online, `openvswitch.org`, Accessed 28.08.2016, 2011. [see page 25]

[PAB+05] Ruoming Pang, Mark Allman, Mike Bennett, Jason Lee, Vern Paxson, and Brian Tierney. A first look at modern enterprise traffic. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association. [see page 140]

[Pat13] Prayson Pate. NFV and SDN: What's the Difference? Online, 2013. Accessed 30.08.2016. [see page 32]

[PC13] J.F. Perez and G. Casale. Assessing sla compliance from palladio component models. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 409–416, Sept 2013. [see pages 17, 115, 124, 129, and 216]

[Pet62] C. A. Petri. Kommunikation mit Automaten. Technical report, 1962. [see page 35]

[Pid03] M Pidd. *Tools for Thinking: Modelling in Management Science*. John Wiley and Sons Ltd, 2nd edition, 2003. [see page 40]

[PS06] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006. [see page 29]

[PTC12] Arun Prakash, Zoltán Theisz, and Ranganai Chaparadza. Formal methods for modeling, refining and verifying autonomic components of computer networks. In *Transactions on Computational Science XV*, pages 1–48. Springer, 2012. [see page 44]

[Pui03] Ramon Puigjaner. Performance modelling of computer networks. In *Proc. of the 2003 IFIP/ACM Latin America conf. on Towards a Latin American agenda for network research*, LANC '03, pages 106–123, New York, NY, USA, 2003. ACM. [see pages 41 and 43]

[QG14] P. Quinn and J. Guichard. Service function chaining: Creating a service plane via network service headers. *Computer*, 47(11):38–44, Nov 2014. [see page 30]

[QKW+04] Lie Qian, A. Krishnamurthy, Yuke Wang, Yiyan Tang, P. Dauchy, and A. Conte. A new traffic model and statistical admission control algorithm for providing qos guarantees to on-line traffic. In *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, volume 3, pages 1401–1405 Vol.3, Nov 2004. [see pages 33 and 47]

[RB94] S. Shenker R. Braden, D. Clark. Integrated services in the internet architecture: an overview. RFC 1633, 1994. [see page 24]

[RBB+11] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, 2011. [see pages 29 and 45]

[RBP+11] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM. [see page 196]

[Res13] Transparency Market Research. Software Defined Networking (SDN) Market Global Industry Analysis, Size, Share, Growth, Trends, and Forecast, 2012–2018. Online, `http://www.transparencymarketresearch.com/software-defined-networking-sdn-market.html`, Accessed 28.08.2016, 2013. [see page 4]

[RH10] GeorgeF. Riley and ThomasR. Henderson. The ns-3 network simulator. In Klaus Wehrle, Mesut Günes, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer Berlin Heidelberg, 2010. [see pages 1, 2, 37, and 42]

[RK96] A. Rueda and Kinsner. A survey of traffic characterization techniques in telecommunication networks. In *Electrical and Computer Engineering, 1996. Canadian Conference on*, volume 2, pages 830–833 vol.2, May 1996. [see page 33]

[RK13] Piotr Rygielski and Samuel Kounev. Network Virtualization for QoS-Aware Resource Management in Cloud Data Centers: A Survey. *PIK — Praxis der Informationsverarbeitung und Kommunikation*, 36(1):55–64, February 2013. [see page vii]

[RK14a] Piotr Rygielski and Samuel Kounev. Data Center Network Throughput Analysis using Queueing Petri Nets. In *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Workshops). 4th International Workshop on Data Center Performance, (DCPerf 2014)*, pages 100–105, June 2014. [see pages viii, 16, 87, 172, 224, 225, and 226]

[RK14b] Piotr Rygielski and Samuel Kounev. Descartes Network Infrastructures (DNI) Manual: Meta-models, Transformations, Examples. Technical Report v.0.3, Chair of Software Engineering, University of Würzburg, Am Hubland, 97074 Würzburg, September 2014. [see pages ix, 15, 16, and 87]

[RKTG15] Piotr Rygielski, Samuel Kounev, and Phuoc Tran-Gia. Flexible Performance Prediction of Data Center Networks using Automatically Generated Simulation Models. In *Proceedings of the Eighth International Conference on Simulation Tools and Techniques (SIMUTools 2015)*, pages 119–128, August 2015. [see pages viii, 15, 16, 87, 172, 224, 225, 226, and 228]

[RKZ13] Piotr Rygielski, Samuel Kounev, and Steffen Zschaler. Model-Based Throughput Prediction in Data Center Networks. In *Proceedings of the 2nd IEEE International Workshop on Measurements and Networking (M&N 2013)*, pages 167–172, October 2013. [see pages viii, 15, 16, 87, 107, 160, 161, 224, 225, and 226]

[Rot09] J. Rothschild. High performance at massive scale lessons learned at facebook. Onine, 2009. Onlin, `http://video-jsoe.ucsd.edu/asx/JeffRothschildFacebook.asx`, Accessed 06.2012. [see page 27]

[RS10] Piotr Rygielski and Pawel Świątek. Graph-fold: an Efficient Method for Complex Service Execution Plan Optimization. *Systems Science*, 36(3):25–32, 2010. [see page 120]

[RSK16] Piotr Rygielski, Marian Seliuchenko, and Samuel Kounev. Modeling and Prediction of Software-Defined Networks Performance using Queueing Petri Nets. In *Proceedings of the Ninth International Conference on Simulation Tools and Techniques (SIMUTools 2016)*, August 2016. [see pages vii, 15, 16, 29, 87, 150, 188, 201, 224, 225, 226, and 237]

[RSKK16] Piotr Rygielski, Marian Seliuchenko, Samuel Kounev, and Mykhailo Klymash. Performance Analysis of SDN Switches with Hardware and Software Flow Tables. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2016)*, October 2016. Paper accepted for publication. [see pages vii, 29, 149, 186, 187, and 201]

[RSS+16] Piotr Rygielski, Viliam Simko, Felix Sittner, Doris Aschenbrenner, Samuel Kounev, and Klaus Schilling. Automated Extraction of Network Traffic Models Suitable for Performance Simulation. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*, pages 27–35. ACM, March 2016. Acceptance rate (Full Paper): 19/57 = 33%. [see pages viii, 16, 138, 211, and 225]

[RZK13] Piotr Rygielski, Steffen Zschaler, and Samuel Kounev. A Meta-Model for Performance Modeling of Dynamic Virtualized Network Infrastructures. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, pages 327–330, New York, NY, USA, April 2013. ACM. [see pages viii, 15, 87, 107, and 224]

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. [see page 47]

[SCBK15a] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51 – 71, October 2015. [see pages 33 and 182]

[SCBK15b] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51 – 71, October 2015. [see page 137]

[Sch09] G. Schmied. *Integrated Cisco, UNIX Network Architectures*. Cisco Press, 2009. [see page 25]

[Sei03] Ed Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, September 2003. [see page 40]

[SFK+17] Simon Spinner, Antonio Filieri, Samuel Kounev, Martina Maggio, and Anders Robertsson. Run-time Models for Online Performance and Resource Management in Data Centers. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware*

Computing Systems. Springer Verlag, Berlin Heidelberg, Germany, 2017. [see page 229]

[SGkY+09] Rob Sherwood, Glen Gibb, Kok kiong Yap, Martin Casado, Nick Mckeown, and Guru Parulkar. Flowvisor: A network virtualization layer. Technical report, 2009. [see page 30]

[ŚJB+12] Paweł Świątek, Krzysztof Juszczyszyn, Krzysztof Brzostowski, Jarosław Drapała, and Adam Grzech. *Supporting Content, Context and User Awareness in Future Internet Applications*, pages 154–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. [see page 229]

[SJLW11] M. Zubair Shafiq, Lusheng Ji, Alex X. Liu, and Jia Wang. Characterizing and Modeling Internet Traffic Dynamics of Cellular Devices. *SIGMETRICS Perform. Eval. Rev.*, 39(1):265–276, June 2011. [see pages 33 and 47]

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003. [see page 84]

[SKM12a] Simon Spinner, Samuel Kounev, and Philipp Meier. Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In Serge Haddad and Lucia Pomello, editors, *Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012)*, volume 7347 of *Lecture Notes in Computer Science (LNCS)*, pages 388–397, Berlin, Heidelberg, June 2012. Springer-Verlag. [see pages 47 and 138]

[SKM12b] Simon Spinner, Samuel Kounev, and Philipp Meier. Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In *Proc. of the 33rd Int. Conf. on Application and Theory of Petri Nets and Concurrency*, pages 388–397. Springer-Verlag, 2012. [see page 92]

[Soc05] IEEE Computer Society. IEEE 802.1Q virtual bridged local area networks. Technical report, IEEE, 2005. [see page 26]

[SPE02] Standard Performance Evaluation Corporation SPEC. SPECjAppServer2001 Documentation. Technical report, Sep 2002. Online, `http://www.spec.org/osg/jAppServer/`, Accessed 28.08.2016. [see pages 216 and 217]

[SSHC+13] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43, July 2013. [see pages 30 and 31]

[ŚTN15] Paweł Świątek, Halina Tarasiuk, and Marek Natkaniec. Delivery of e-health services in next generation networks. In Thanh Ngoc Nguyen, Bogdan Trawiński, and Raymond Kosala, editors, *Intelligent Information and Database Systems: 7th Asian Conference, ACIIDS 2015, Bali, Indonesia, March 23-25, 2015, Proceedings, Part II*, pages 453–462. Springer International Publishing, Cham, 2015. [see page 229]

[Sto16] Jonathan Stoll. SDN-basierte Lastverteilung für Schicht-7 Anfragen (SDN Rechenzentrum Fallstudie). Bachelor Thesis, University of Würzburg, Am Hubland, 97074 Würzburg, Germany, March 2016. [see pages 21, 178, and 180]

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001. [see pages 23 and 25]

[SWHB06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association. [see page 74]

[SWK16] Simon Spinner, Jürgen Walter, and Samuel Kounev. A Reference Architecture for Online Performance Model Extraction in Virtualized Environments. In *Proceedings of the 2016 Workshop on Challenges in Performance Methods for Software Development (WOSP-C'16) co-located with 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*, March 2016. [see page 135]

[SZC16] Rick Salay, Steffen Zschaler, and Marsha Chechik. Correct reuse of transformations is hard to guarantee. In Pieter van Gorp and Gregor Engels, editors, *Proc. 9th Int'l Conf Model Transformations (ICMT'16)*, pages 107–122. Springer International Publishing, 2016. [see page 231]

[Tay14] Martin Taylor. A Guide to NFV and SDN. White Paper, Online, 2014. Accessed 30.08.2016. [see page 30]

[TC98] Christopher Torrence and Gilbert P. Compo. A practical guide to wavelet analysis. *Bulletin of the American Meteorological Society*, 79:61–78, 1998. [see page 147]

[Tea14] Ryu Project Team. *RYU SDN Framework - English Edition Release 1.0*. 2014. eBook. [see page 180]

[TGG+12] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *2nd Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Berkeley, CA, 2012. USENIX. [see page 44]

[TH00] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, 2000. [see page 196]

[upe12] Uperf A network performance tool. Online, `www.uperf.org`, Accessed 28.08.2016, 2012. [see pages 160 and 179]

[Var01] Andras Varga. The OMNeT++ discrete event simulation system. In *Proc. of the European Simulation Multi-conference*, pages 319–324, 2001. [see pages 1, 37, 42, and 107]

[vKHK14] Jóakim Gunnarson von Kistowski, Nikolas Roman Herbst, and Samuel Kounev. LIMBO: A Tool For Modeling Variable Load Intensities. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, pages 225–226. ACM, 2014. [see pages 56 and 128]

[vKHZ+15] Jóakim von Kistowski, Nikolas Roman Herbst, Daniel Zoller, Samuel Kounev, and Andreas Hotho. Modeling and Extracting Load Intensity Profiles. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, May 2015. [see page 47]

[VV06] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 111–122, New York, NY, USA, 2006. ACM. [see pages 47 and 48]

[vxl11] Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks. Online, 2011. [see page 26]

[WGG10] Klaus Wehrle, Mesut Günes, and James Gross, editors. *Modeling and Tools for Network Simulation*. Springer Berlin Heidelberg, 2010. [see pages 37 and 43]

[WHKF12] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM. [see page 33]

[WN10a] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010. [see page 1]

[WN10b] Guohui Wang and T.S.E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010. [see page 2]

[WNPM95] C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *Computers, IEEE Transactions on*, 44(1):20–34, Jan 1995. [see pages 35 and 116]

[WP98] Walter Willinger and Vern Paxson. Where mathematics meets the internet. *Notices of the American Mathematical Society*, pages 961–970, 1998. [see page 33]

[WPP+05] Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by unified model analysis (puma). In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 1–12. ACM, 2005. [see page 45]

[WSK15] Jürgen Walter, Simon Spinner, and Samuel Kounev. Parallel Simulation of Queueing Petri Nets. In *Proceedings of the Eighth EAI International Conference on Simulation Tools and Techniques (SIMUTools 2015)*, August 2015. [see pages 117 and 126]

[WvHK+16] Jürgen Walter, Andre van Hoorn, Heiko Koziolek, Dusan Okanovic, and Samuel Kounev. Asking "What?", Automating the "How?": The Vision of Declarative Performance Engineering. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*, March 2016. [see page 17]

[WvLW09] E. Weingartner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Communications, 2009. ICC '09. IEEE International Conference on*, pages 1–5, June 2009. [see pages 2 and 204]

[Xco16] Xcore: Modeling for Programmers and Programming for Modelers. Online, `https://wiki.eclipse.org/Xcore`, Accessed 28.08.2016, 2016. [see page 18]

[Zho10] S. Zhou. Virtual networking. *SIGOPS Oper. Syst. Rev.*, 44(4), 2010. [see page 25]

[Zsc09] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modelling (SoSyM)*, 9:161–201, 2009. [see page 45]