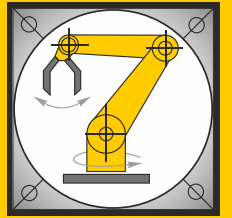


Institut für Informatik  
Lehrstuhl für Robotik und Telematik  
Prof. Dr. K. Schilling  
Prof. Dr. A. Nüchter



Würzburger Forschungsberichte  
in Robotik und Telematik

Uni Wuerzburg Research Notes  
in Robotics and Telematics

Julius-Maximilians-

**UNIVERSITÄT  
WÜRZBURG**

Dissertation an der Graduate School  
of Science and Technology

Veaceslav Dombrovski

Software Framework to  
Support Operations of  
Nanosatellite Formations

**Band 23**



**Doctoral thesis / *Dissertation***  
for the doctoral degree / *zur Erlangung des Doktorgrads*  
**Doctor rerum naturalium (Dr. rer. nat.)**

Software Framework to Support Operations of Nanosatellite  
Formations

*Software Framework für die Unterstützung des Betriebs  
von Nanosatelliten-Formationen*



Submitted by / *Vorgelegt von*  
**Veaceslav Dombrovski**  
from / *aus*  
Chisinau  
Würzburg, 2021





Submitted on / *Eingereicht am*: 11.06.2021

**Members of thesis committee / *Mitglieder des Promotionskomitees***

Chairperson / *Vorsitz*: Prof. Dr. Bert Hecht

1. Reviewer and Examiner / *1. Gutachter und Prüfer*: Prof. Dr. Klaus Schilling
2. Reviewer and Examiner / *2. Gutachter und Prüfer*: Prof. Dr. Sergio Montenegro
3. Examiner / *3. Prüfer*: Prof. Dr. Jens Eickhoff

Day of thesis defense / *Tag des Promotionskolloquiums*: 03.11.2021



# Abstract

Since the first CubeSat launch in 2003, the hardware and software complexity of the nanosatellites was continuously increasing with the aim to realize capabilities that are normally offered by bigger satellites. To keep up with the continuously increasing mission complexity *and* at the same time retain the primary advantages of a CubeSat mission (cost-effectiveness, small development teams, fast development times), a new approach for the overall space and ground software architecture and protocol configuration is elaborated in this work.

The aim of this thesis is to propose a uniform software and protocol architecture as a basis for software development, test, simulation and operation of multiple pico-/nanosatellites based on *ultra-low power* components. In contrast to single-CubeSat missions, current and upcoming nanosatellite formation missions (NetSat, TIM/TOM and CloudCT) require faster and more straightforward development, pre-flight testing and calibration procedures as well as simultaneous operation of multiple satellites.

A dynamic and decentral *Compass mission network* was established in multiple active CubeSat missions, consisting of uniformly accessible *Compass nodes*. *Compass middleware* was elaborated to unify the communication and functional interfaces between all involved mission-related software and hardware components: operation workstations, ground stations, mission servers, test facilities, simulations and ultra-low power satellite subsystems. All systems can access each other via dynamic routes to perform service-based M2M communication. Standard services were implemented to support all required tasks during the software development, testing, simulation and in-orbit operations.

With the proposed *model-based communication* approach, all states, abilities and functionalities of a system are accessed in a uniform way. The *Tiny* scripting language was designed to allow dynamic code execution on ultra-low power components as a basis for constraint-based in-orbit scheduler and experiment execution. The implemented *Compass Operations front-end* enables far-reaching monitoring and control capabilities of all ground and space systems of the mission network. Its integrated constraint-based *operations task scheduler* allows the recording of complex satellite operations, which are conducted automatically during the overpasses.

The outcome of this thesis (Compass middleware, Compass OS, Tiny and Compass Operations front-end) became an enabling technology for UWE-3, UWE-4 and NetSat CubeSat missions in which it was successfully tested in-orbit.



# Zusammenfassung

Seit dem Launch des ersten CubeSats im Jahr 2003, hat die Komplexität der Nanosatelliten stetig zugenommen, mit dem Ziel sie mit Fähigkeiten auszustatten, die zuvor nur größeren Satelliten vorbehalten waren. Um mit den wachsenden Anforderungen Schritt zu halten und gleichzeitig nicht auf die Hauptvorteile einer CubeSat Mission zu verzichten (Kosteneffektivität, schnelle Entwicklung, kleine Entwicklungsteams), wird eine einheitliche Protokoll- und Softwarearchitektur für den gesamten Weltraum- und Bodensegment einer Mission vorgeschlagen.

Diese Arbeit schlägt eine einheitliche Software- und Protokoll-Architektur vor als Basis für Softwareentwicklung, Tests und Betrieb von mehreren Pico-/Nanosatelliten, die auf *extrem energiesparenden* Komponenten aufbauen. Im Gegensatz zu Missionen mit nur einem CubeSat, erfordern künftige Nanosatelliten-Formationen (wie NetSat, TIM/TOM und CloudCT) eine schnellere und einfachere Entwicklung, Vorflug-Tests, Kalibrierungsvorgänge sowie die Möglichkeit mehrere Satelliten gleichzeitig zu betreiben.

Ein dynamisches und dezentrales *Compass Missionsnetzwerk* wurde in mehreren CubeSat Missionen realisiert, bestehend aus einheitlich zugänglichen *Compass Knoten*. Die *Compass-Middleware* wurde entwickelt, um sowohl die Kommunikation als auch funktionale Schnittstellen zwischen allen beteiligten Software und Hardware Systemen in einer Mission zu vereinheitlichen: Rechner des Bedienpersonals, Bodenstationen, Mission-Server, Testeinrichtungen, Simulationen und Subsysteme aller Satelliten. Alle Systeme können aufeinander über dynamische Routen zugreifen, um Service-basierte *Machine-to-Machine Prozess-Kommunikation* zu betreiben. Standardisierte Services wurden definiert, um alle Aufgaben im Bereich der Softwareentwicklung, Tests, Simulationen und des in-orbit Betriebs zu unterstützen.

Mit dem Ansatz der *modellbasierten Kommunikation* wird auf alle Zustände, Fähigkeiten und Funktionen eines Systems einheitlich zugegriffen. Die entwickelte *Tiny Skriptsprache* ermöglicht die Ausführung von dynamischem Code auf energiesparenden Systemen, um so in-orbit Scheduler zu realisieren. Das *Compass Operations Front-End* bietet zahlreiche grafische Komponenten, mit denen alle Weltraum- und Bodensegment-Systeme einheitlich überwacht, kontrolliert und bedient werden können. Der integrierte *Betrieb-Scheduler* ermöglicht die Aufzeichnung von komplexen Satellitenbetrieb-Aufgaben, die dann beim Überflug automatisch ausgeführt werden.

Die Ergebnisse dieser Arbeit (Compass Middleware, Compass OS, Tiny und Operations Front-End) wurden zur Enabling-Technologie für UWE-3, UWE-4 und NetSat Missionen und wurden erfolgreich im Orbit getestet.



## Acknowledgements

This thesis would not happen to be possible without the guidance and the help of several individuals who contributed their assistance in the preparation of this thesis.

I would like to express the deepest appreciation to Prof. Dr. Klaus Schilling, Chair of Robotics and Telematics at the University Würzburg, for giving me the opportunity to be a part of the space-team family. Also special thanks to Prof. Dr. Sergio Montenegro and Prof. Dr. Jens Eickhoff for supporting my thesis and giving me interesting and helpful insights into their fields of research. Many thanks to Dr. Schröder-Köhne for supporting me in administrative affairs and for organizing wonderful and informative get-together events.

I cannot find words to express my gratitude to Stephan Busch whose guidance, patience and encouragement I will never forget. Mr. Busch has been my inspiration during my studies. I share the credit of my work with Dieter Ziegler, for the technical insights he has shared and interesting discussions during the break – and some good laughs. I consider it an honor to have worked with Philip Bangert and Alexander Kramer – at times it was an amazing experience to see how fast and how far we could get in close collaboration backed by our will and stamina. Also special thanks to the ZfT family – Oliver Ruf, Julian Scharnagl, Florian Kempf and other wonderful colleagues who supported me during this thesis.

I am forever indebted to my parents for giving me love and inspiration throughout my life. A very special word of thanks goes to my fiance and best friend Lisa for the unconditional support and for lifting my spirits. Many thanks to my friend Martin for motivation and our numerous adventures in the nature. Though they can't possibly know how much of a help they have been, I'd like to thank Cookie and Ginger for keeping my feet warm during the long thesis writing sessions.





## Acronyms

<b>ADCS</b>	Attitude Determination and Control System
<b>AOCS</b>	Attitude and Orbit Control System
<b>AIT</b>	Assembly, Integration and Test
<b>AIV</b>	Assembly, Integration and Verification
<b>API</b>	Application Programming Interface
<b>ARR</b>	Automatic Record and Report
<b>ATF</b>	Along Track Formation
<b>CAN</b>	Controller Area Network
<b>CDMA</b>	Code Division Multiple Access
<b>CollExp</b>	Collect and Expand
<b>COTS</b>	Commercial Off-The-Shelf
<b>CRC</b>	Cyclic Redundancy Check
<b>CSP</b>	Cubesat Protocol
<b>CSMA</b>	Carrier Sense Multiple Access
<b>DB</b>	Database
<b>DLR</b>	Deutsches Zentrum für Luft- und Raumfahrt e.V.
<b>DCE</b>	Dynamic Code Execution
<b>DevKit</b>	Development Kit
<b>DNS</b>	Domain Name System
<b>DTN</b>	Delay Tolerant Network
<b>ELISA</b>	Electronic Intelligence by Satellite
<b>ECC</b>	Error Correction Code
<b>EPS</b>	Electrical Power System
<b>ESA</b>	European Space Agency
<b>FDMA</b>	Frequency Division Multiple Access
<b>FDIR</b>	Fault-Detection, Fault-Isolation and Recovery
<b>FEC</b>	Forward Error Correction
<b>FFS</b>	Fast File System

<b>FSK</b>	Frequency Shift Keying
<b>GCS</b>	Ground Control Segment
<b>GBO</b>	Goal Based Operations
<b>GEO</b>	Geostationary Orbit
<b>GCC</b>	GNU Compiler Collection
<b>GDS</b>	Generic Data System
<b>GS</b>	Ground Station
<b>GPIO</b>	Generic Purpose Input Output
<b>GRACE</b>	Gravity Recovery and Climate Experiment
<b>GRACE-FO</b>	GRACE-Follow-On
<b>GSN</b>	Ground Station Network
<b>GSS</b>	Ground Station Server
<b>GUI</b>	Graphical User Interface
<b>HAL</b>	Hardware Abstraction Layer
<b>HEO</b>	High Elliptical Orbit
<b>HIL</b>	Hardware in the Loop
<b>HW</b>	Hardware
<b>I2C</b>	Inter-integrated Circuit protocol
<b>IDE</b>	Integrated Development Environment
<b>IoT</b>	Internet of Things
<b>IPN</b>	Inter-Planetary Network
<b>ISL</b>	Inter-Satellite Link
<b>JPL</b>	NASA Jet Propulsion Laboratory
<b>LEO</b>	Low Earth Orbit
<b>LEOP</b>	Launch and Early Orbit Phase
<b>MCC</b>	Mission Control Center
<b>MCS</b>	Mission Control System
<b>MEO</b>	Medium Earth Orbit
<b>MBD</b>	Model Based Design

<b>MCC</b>	Mission Control Center
<b>MDE</b>	Model Driven Development
<b>MIL</b>	Model in the Loop
<b>MLT</b>	Mixed Loop Testing
<b>MMS</b>	Magnetospheric Multiscale Mission
<b>MO</b>	Mission Operations service
<b>MQTT</b>	Message Queuing Telemetry Transport
<b>MS</b>	Mission Server
<b>MTBA</b>	Model Tree Based Architecture
<b>MVC</b>	Model View Controller
<b>NanoFEEP</b>	Nanosatellite Field Emission Electric Propulsion
<b>NASA</b>	National Aeronautics and Space Administration
<b>NetSat</b>	Networked Satellite Formation Flying Mission
<b>NFS</b>	Network File System
<b>NSP</b>	Nanosatellite Protocol
<b>OBDH</b>	On-Board Data Handling
<b>OBC</b>	On-Board Computer
<b>OCS</b>	Orbit Control System
<b>OCC</b>	Operations Control Center
<b>OSI model</b>	Open Systems Interconnection model
<b>OSIRIS</b>	Optical Space Infrared Downlink System
<b>PIL</b>	Processor in the Loop
<b>PPU</b>	Thruster Power Processing Unit
<b>PUS</b>	ECSS Packet Utilisation Standard
<b>Prisma</b>	Prototype Research Instruments and Space Mission technology Advancement
<b>RCP</b>	Rich Client Platform
<b>RTOS</b>	Real Time Operating System
<b>SCOS</b>	Space Control And Operating System

<b>SDMA</b>	Space Division Multiple Access
<b>SDR</b>	Software Defined Radio
<b>SIDS</b>	Simple Downlink Sharing Convention
<b>SIL</b>	Software in the Loop
<b>SOC</b>	System On a Chip
<b>SPB</b>	Subsystem Prototyping Board
<b>SPI</b>	Serial Peripheral Interface
<b>SW</b>	Software
<b>SVN</b>	Apache Subversion
<b>TC</b>	Telecommand
<b>TDMA</b>	Time Division Multiple Access
<b>TIM</b>	Telematics International Mission
<b>TM</b>	Telemetry
<b>TNC</b>	Terminal Node Controller
<b>TLE</b>	Two Line Element set
<b>TOM</b>	Telematics Earth Observation Mission
<b>TT/C</b>	Telemetry, Tracking and Control
<b>TU</b>	Thruster Control unit
<b>UFS</b>	UWE File System
<b>UHF</b>	Ultra High Frequency
<b>UHF GS</b>	Ultra High Frequency Groundstation
<b>UWE</b>	University Würzburg Experimental satellite
<b>UWE-3</b>	University Würzburg Experimental satellite 3
<b>UWE-4</b>	University Würzburg Experimental satellite 4
<b>XTEA</b>	eXtended Tiny Encryption Algorithm
<b>ZfT</b>	Zentrum für Telematik

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Baseline Mission . . . . .	4
1.3	Contributions . . . . .	5
1.4	Thesis Outline . . . . .	7
<b>2</b>	<b>State of the Art</b>	<b>9</b>
2.1	Requirements . . . . .	10
2.1.1	Baseline Mission . . . . .	10
2.1.2	Protocol . . . . .	12
2.1.3	Services . . . . .	15
2.1.4	Space Segment software . . . . .	18
2.1.5	Ground Segment software . . . . .	20
2.2	Formation Missions . . . . .	21
2.2.1	CanX-4 & 5 . . . . .	21
2.2.2	Prisma . . . . .	21
2.2.3	GRACE . . . . .	22
2.2.4	HawkEye 360 Pathfinder . . . . .	22
2.2.5	MMS . . . . .	23
2.2.6	NetSat . . . . .	23
2.2.7	PROBA-3 . . . . .	23
2.2.8	CloudCT . . . . .	23
2.3	On-board Autonomy in Constellations . . . . .	24
2.3.1	SWARM . . . . .	24
2.3.2	TerraSAR-X and TanDEM-X . . . . .	24
2.3.3	Galileo . . . . .	25
2.3.4	Planet . . . . .	26
2.3.5	OneWeb . . . . .	26
2.4	IoT/M2M Missions . . . . .	27
2.4.1	Starlink . . . . .	28
2.4.2	Myriota . . . . .	28
2.4.3	Astrocast . . . . .	28
2.4.4	Kepler . . . . .	29
2.4.5	Swarm . . . . .	29
2.4.6	kineis . . . . .	29
2.5	OPS-SAT and MO services . . . . .	30

2.6	Protocols . . . . .	31
2.6.1	CCSDS Recommendations . . . . .	31
2.6.2	Packet Utilization Standard services . . . . .	35
2.6.3	Ground Segment Protocols . . . . .	36
2.6.4	Ground-Space Protocols . . . . .	38
2.6.5	Inter Satellite Link . . . . .	40
2.7	Considerations on Satellite Development and Operations . . . . .	41
2.7.1	Satellite Software Development . . . . .	41
2.7.2	Testing and Verification . . . . .	43
2.7.3	Distributed Dynamic Mission Network . . . . .	44
2.7.4	Operations . . . . .	44
2.8	Roundup . . . . .	45
<b>3</b>	<b>Approach</b>	<b>47</b>
3.1	Uniform Model Interface . . . . .	49
3.1.1	Model Tree Based Architecture . . . . .	51
3.1.2	Model Tree shadowing and Model-based communication . . . . .	53
3.1.3	Model buffering . . . . .	53
3.1.4	Model swapping . . . . .	54
3.1.5	Implementation . . . . .	54
3.1.6	Testability . . . . .	55
3.2	Uniform Communication . . . . .	57
3.2.1	Addressing . . . . .	59
3.2.2	Channels . . . . .	60
3.2.3	Routing . . . . .	63
3.2.4	Advanced Protocol Functions . . . . .	64
3.3	High-level Functionality . . . . .	66
3.3.1	Services . . . . .	66
3.3.2	Network Features . . . . .	66
3.3.3	Telemetry, Tracking and Control (TT&C) . . . . .	67
3.3.4	Testing and Fault Diagnostics . . . . .	67
3.3.5	File Link . . . . .	68
3.3.6	Dynamic Code Execution . . . . .	69
3.4	Summary . . . . .	71
<b>4</b>	<b>Compass Protocol</b>	<b>73</b>
4.1	Overview . . . . .	74
4.2	Packet Definition . . . . .	75
4.2.1	Addressing . . . . .	77
4.2.2	Payload size . . . . .	78
4.2.3	Time field . . . . .	78
4.2.4	Route Format . . . . .	78
4.2.5	Route . . . . .	80
4.2.6	SGN . . . . .	80
4.2.7	CRC . . . . .	81
4.2.8	Error . . . . .	81
4.2.9	Urgent . . . . .	81

4.2.10	Encryption . . . . .	81
4.2.11	Zip Compression . . . . .	82
4.3	Services . . . . .	83
4.3.1	Network . . . . .	86
4.3.2	Echo . . . . .	90
4.3.3	Command . . . . .	91
4.3.4	Downlink . . . . .	93
4.3.5	Uplink . . . . .	95
4.3.6	Log . . . . .	96
4.3.7	Unit-Test . . . . .	97
4.3.8	Network File System . . . . .	99
4.3.9	Tiny script . . . . .	102
4.3.10	Model . . . . .	107
4.3.11	Recording and Reporting . . . . .	114
4.3.12	Database . . . . .	117
4.3.13	Registry . . . . .	117
4.3.14	Tunnel . . . . .	119
4.4	Channels . . . . .	120
4.4.1	Generic Byte Stream Channels . . . . .	120
4.4.2	TCP or UDP . . . . .	122
4.4.3	I2C . . . . .	122
<b>5</b>	<b>Space Segment</b>	<b>125</b>
5.1	Hardware environment . . . . .	126
5.1.1	Microcontrollers . . . . .	127
5.1.2	Payload . . . . .	127
5.2	Communication . . . . .	128
5.2.1	Satellite bus . . . . .	128
5.2.2	Space-Ground . . . . .	129
5.2.3	Inter Satellite Link . . . . .	130
5.3	Compass OS . . . . .	130
5.3.1	Hardware Abstraction Layer . . . . .	131
5.3.2	Embedded File Systems . . . . .	131
5.3.3	Channels . . . . .	132
5.4	Compass services . . . . .	134
5.4.1	Network service . . . . .	134
5.4.2	Command service . . . . .	136
5.4.3	Model service . . . . .	137
5.4.4	Unit Testing . . . . .	139
5.4.5	File service . . . . .	139
5.5	Dynamic Code Execution with Tiny . . . . .	140
5.5.1	Tiny Language . . . . .	141
5.5.2	Tiny IDE . . . . .	144
5.5.3	External Functions . . . . .	147
5.5.4	Compass bonding . . . . .	147
5.5.5	Remote Function Execution . . . . .	148

<b>6</b>	<b>Ground Segment</b>	<b>151</b>
6.1	Environment . . . . .	152
6.1.1	Before this thesis . . . . .	152
6.1.2	During this thesis . . . . .	153
6.2	Java Implementation . . . . .	154
6.2.1	Fire Framework . . . . .	155
6.3	Ground Station Server . . . . .	156
6.4	Mission Server . . . . .	160
6.5	External Ground Stations . . . . .	161
6.6	Compass Operations front-end . . . . .	162
6.6.1	Node selection . . . . .	166
6.6.2	Nodes View . . . . .	167
6.6.3	Packet View . . . . .	168
6.6.4	Command View . . . . .	171
6.6.5	Model View . . . . .	173
6.6.6	Uplink and Downlink View . . . . .	175
6.6.7	Unit Testing . . . . .	177
6.6.8	Value Monitors . . . . .	177
6.6.9	Schedule View . . . . .	178
6.6.10	Echo View . . . . .	179
6.6.11	Tiny View . . . . .	181
6.7	Auto-Operations . . . . .	183
6.7.1	Task Creation . . . . .	184
6.7.2	Task recording . . . . .	185
6.7.3	File Links . . . . .	186
<b>7</b>	<b>Testing and Live System Experience</b>	<b>187</b>
7.1	Protocol Performance . . . . .	189
7.1.1	Serial communication . . . . .	189
7.1.2	Local TCP . . . . .	189
7.1.3	Remote TCP . . . . .	191
7.1.4	Radio Link . . . . .	191
7.2	TOM Scenario . . . . .	191
7.3	UWE-4 Sensor Calibration . . . . .	193
7.4	In-Orbit Dynamic Code Execution . . . . .	197
7.5	UHF Ground Stations . . . . .	199
7.6	Multi Satellite Operations . . . . .	201
<b>8</b>	<b>NetSat Experience</b>	<b>205</b>
8.1	External Tests and Verification . . . . .	205
8.2	Pre-flight LEOP exercises . . . . .	206
8.3	Launch and Early Orbit Phase . . . . .	208
8.4	Auto-Operations . . . . .	210
8.5	Dynamic Code Execution with Tiny . . . . .	211
8.6	ISL . . . . .	213
8.7	Outlook . . . . .	215



<b>9</b>	<b>Conclusions</b>	<b>219</b>
9.1	Ground Segment . . . . .	219
9.2	Multi-Satellite Operations . . . . .	220
9.3	Space Segment and In-Orbit Autonomy . . . . .	222
9.4	Compass Protocol usage . . . . .	223
9.5	Future Work . . . . .	224
9.6	Publications . . . . .	226
	<b>Appendices</b>	<b>229</b>
	Compass Node Creation . . . . .	231
	Matlab . . . . .	231
	Java . . . . .	232
	Front-End Examples . . . . .	235
	Traffic comparison . . . . .	237
	Examples from NetSat Operations . . . . .	241



# 1 | Introduction

This work addresses challenges of space and ground segment systems of pico- and nanosatellite formation missions with respect to the communication, functional interfaces and operation. A *model-based approach* to unify the access to the functionality of ground systems and ultra-low power satellite subsystems will be presented. After pointing out challenges of single space and ground communication segments, the *Compass protocol* is elaborated. Based on the identified common responsibilities and tasks of every particular system in an exemplary formation mission, a set of segment-agnostic *standard services* is defined with special attention being paid to their suitability for ultra-low power satellite subsystems. The *Compass middleware* was implemented to support both extremely limited micro-controllers and high-power workstations, and it is demonstrated how this framework was used as a basis for all space and ground systems in the University Würzburg Experimental satellite 4 (UWE-4) and Networked Satellite Formation Flying Mission (NetSat) missions to span a uniform and decentralized mission network consisting of satellite subsystems, ground stations, mission servers, test equipment and operator's workstations. It will be shown how the model-approach has enabled model-based machine-to-machine communication and how the entirety of all distributed system models forms a top-level digital twin of the mission. The *Tiny script* service is presented, which enables execution of dynamic code on ultra-low power satellite subsystems and was demonstrated on board the UWE-3, UWE-4 and NetSat satellites to execute experiments and run constraint-based task schedules. Furthermore, the *Compass Operations front-end* has been developed to monitor, control and operate all systems within the mission network without a priori knowledge, and is also presented in this work with examples from the NetSat operations. The inherent IoT capabilities of the Compass middleware and the model-based communication approach open up possibilities for further forward-looking studies, such as anomaly detection based on machine learning methods or *super-missions*, where satellites from different missions perform cooperative tasks.

## 1.1 Motivation

In 1999, Bob Twiggs from Stanford University and Jordi Puig-Suari from California Polytechnic State University elaborated the first *CubeSat reference design* with the aim of promoting skills required for design and implementation of small LEO satellites as a platform for the scientific research and exploration of new space approaches[Pro14]. First CubeSats launched four years later, in June 2003. Initially, the specification was used mainly by the academia – the majority of all launched CubeSats until 2013 was for academic purposes until they have been superseded by commercial and amateur missions (figure 1.1). At this

point also the 3U CubeSats, measuring 10x10x30 cm, began to displace the previously favored 1U size.

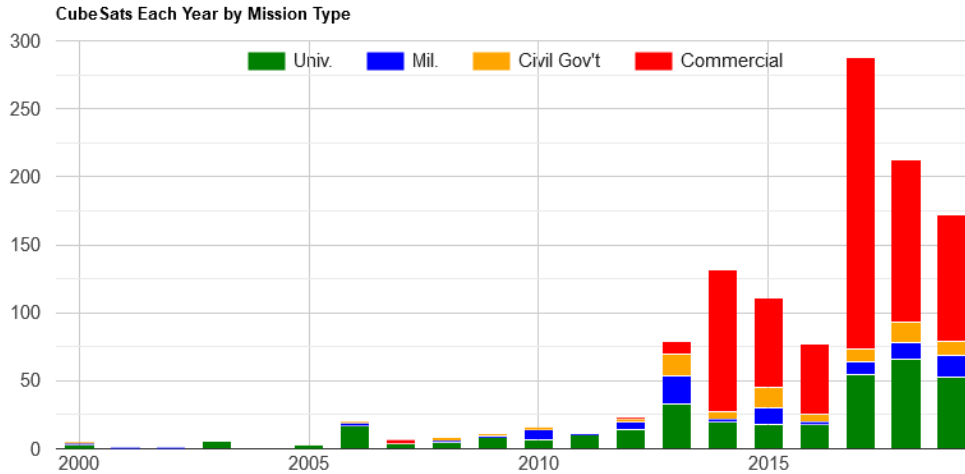


Figure 1.1: Launched CubeSats, grouped by mission types. Image source: [Swa21]

Since the first CubeSat launch, the complexity of the nanosatellites was continuously increasing. Today, some CubeSats offer capabilities that formerly were in the domain of much larger satellites: global communication, Earth observation, formations, etc. In addition to that, the amount of nanosatellite constellations and formations is steadily increasing. At the same time, the majority of all CubeSat missions are based on spacecrafts up to 3U[Swa21], i.e. most of the satellite subsystems (satellite bus) are still constrained by the size and power limitations. To keep up with the continuously increasing mission complexity *and* at the same time retain the primary advantages of a CubeSat mission (cost-effectiveness, small development teams, fast development times), a new approach for the overall space and ground software architecture is elaborated in this work.

The initial motivation of this work originated from the Launch and Early Orbit Phase (LEOP) phase of the University Würzburg Experimental satellite 3 (UWE-3) mission in January 2014. The UWE mission line began in 2005 and is intended for educational purposes at the University of Würzburg, Chair VII Robotics and Telematics, where students can participate on the satellite development and contribute to the scientific, software or hardware work packages and thus gradually obtain a big picture of real satellite missions.

The UWE-1 CubeSat was the first German CubeSat and was successfully launched 2005 from Plesetsk Cosmodrome, Russia, and was used to test Internet protocols in space [SPS07]. The UWE-2 CubeSat was designed to test *attitude determination* techniques and was launched 2009 from Satish Dhawan Space Centre, India [Sch+09]. Its successor, UWE-3, launched 2013 in Dombrovsky Air Base, Russia, was completely redesigned and equipped with *attitude control* capabilities. With UWE-3 a new *UNISEC Electrical Interface Bus* was introduced[BS17], which since then became the standard subsystem

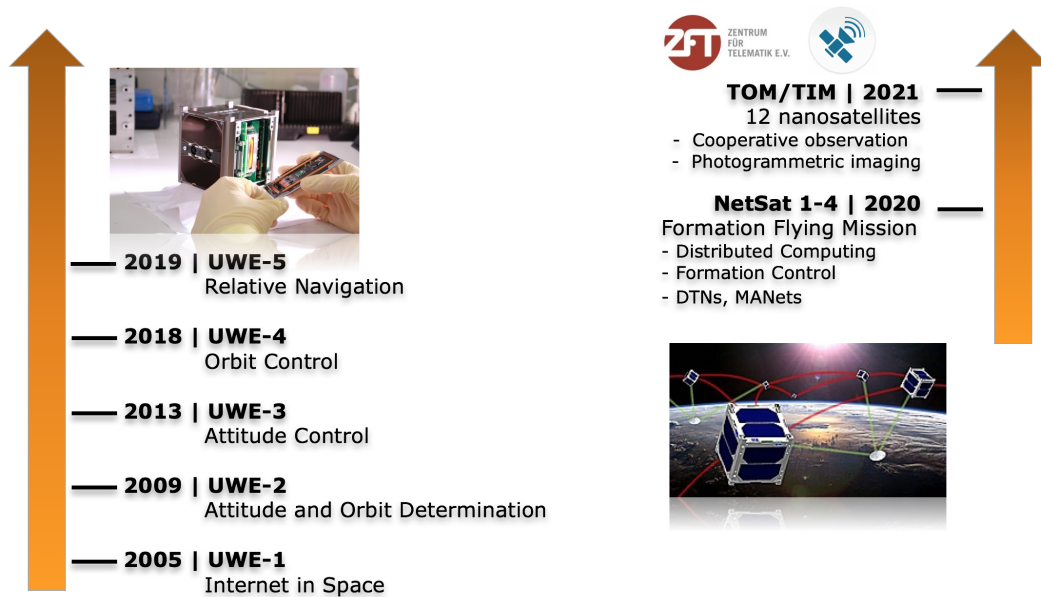


Figure 1.2: Mission roadmap of the University of Würzburg (left) and Zentrum für Telematik (right)

hardware interface for all following satellite missions at Zentrum für Telematik (ZfT). Even though the ground segment and the satellite software of the UWE-3 mission was improved as compared to the former UWE missions, several flaws – with respect to the reusability – were existing:

- space segment software:
  - *Missionlink protocol* used for satellite communication supported only one ground and one space node
  - different custom protocols between the main On-Board Data Handling (OBDH) and other subsystems – panels and Attitude Determination and Control System (ADCS)
  - all satellite subsystems were represented by one OBDH node
  - the OBDH was the only subsystem that could actively initiate packet transmission or communicate with other subsystems
  - the software of all subsystems had no common basis
  - no abstraction of different physical interfaces
- ground segment software:
  - different protocols were used during the development (*DebugComm*, *SixPack*) and in-orbit phase (*Missionlink*)
  - the software of the UHF ground station was based on several components with different monitoring and control interfaces
  - no auto-operation capabilities – operations could only be performed manually
  - no interface for externally received packets – e.g. from radio amateurs

It became evident that to keep up with the continuously increasing mission com-

plexity, a new approach must be elaborated to optimize the entire satellite and ground software architecture of the upcoming missions. More specifically, the software and protocols of the next UWE-4 and *NetSat* missions had to be designed and implemented based on the new approach. The goal of the UWE-4 1U CubeSat mission was to perform orbit maneuvers with four Nanosatellite Field Emission Electric Propulsion (NanoFEEP) thrusters. It contains 11 cooperative microcontrollers: 2x on-board computer (On-Board Computer (OBC)), Attitude and Orbit Control System (AOCS), 2x Thruster Power Processing Unit (PPU) and 6x panels – all of which need to be separately monitored, controlled and updated from ground. UWE-4 was also intended to be used as a test platform to gain knowledge for subsequent nanosatellite missions, such as NetSat [Sch+15], Telematics Earth Observation Mission (TOM) [Sch+17; Sch+18] and CloudCT[KS19a; KS19c; KS19b] missions – all being developed at ZfT, Würzburg.

With formation missions being in queue of the ZfT’s development timeline, it became clear that challenges, as compared to a single satellite mission, will significantly increase with respect to the development, testing, pre-flight sensor calibration, communication and operations. In addition to the increased complexity of the satellite and ground segment software development, auxiliary mission work packages also need to be harmonized in order to remain maintainable by smaller CubeSat teams, such as:

- software-based testing
- hardware-in-the-loop tests
- calibration and sensor testing procedures with test facilities (e.g. turntables, sun simulator)
- orbit and system simulation (Orekit, Matlab)
- utilization of multiple ground stations
- receive data from external ground stations (radio amateurs)
- uniform operability of all mentioned mission components

So, in addition to the established UNISEC hardware interface, a new uniform software and protocol standard for both space and ground systems had to be elaborated.

## 1.2 Baseline Mission

The ZfT’s NetSat formation, consisting of four 3U CubeSats, was defined as an exemplary *baseline mission*, as it contains all common space and ground systems that are present in comparable pico- and nanosatellite constellation and formation missions. A simplified overview of the NetSat space and ground systems is depicted in figure 1.3. Most of the NetSat’s subsystems are based on ultra-low power microcontrollers, thus enforcing very low hardware requirements of the new framework. To prove the feasibility of the proposed approaches, also the UWE-3 and UWE-4 1U CubeSat missions were used for testing – both have microcontroller specifications very similar to NetSat. During the kick-off of this thesis, the UWE-3 CubeSat was already in orbit. Therefore, it could only be used to test single space software components (via in-orbit software update). The launch date of the UWE-4 CubeSat was at the same time the deadline at which the technical realization of this work had to be finished. That is, the software of all UWE-4 subsystems and of the entire ground segment had to be based solely on the developed framework/middleware. The time between the UWE-4 and the NetSat launch could then be used to improve the

overall system. As in UWE-4 CubeSat, the software of all NetSat subsystems is based on the developed middleware. So, by the end of this work five Compass-enabled satellites, with almost 60 separately accessible systems, and numerous ground systems are combined into one dynamic and decentralized mission network.

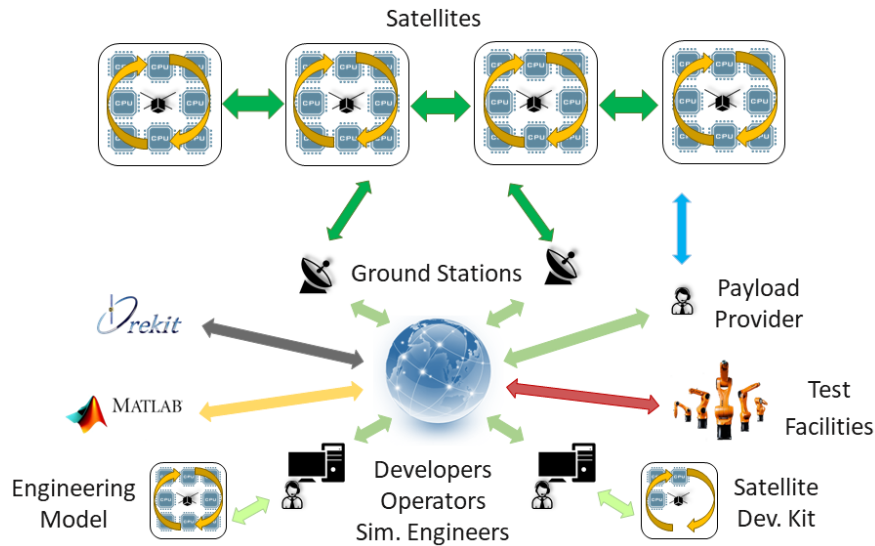


Figure 1.3: Ground and space segment of the exemplary NetSat formation mission

## 1.3 Contributions

This dissertation proposes a novel approach for the unification of common space and ground systems of a nanosatellite mission with respect to the communication protocol configuration, functional interfaces and operation. It is applicable both on very low-power embedded devices and on high-power workstations.

### Compass protocol and services

The elaborated *Compass protocol* unifies the communication between all space and ground systems and forms a decentralized and dynamic *mission network*, consisting of satellite subsystems, test equipment, ground stations, mission servers, simulations, operator's workstations and further Compass-enabled devices. Hence, standard protocol and services are used between the satellite's subsystems (intra satellite link), between the satellites (ISL), for space-ground communication and between the ground nodes. With *Model Tree Based Architecture (MTBA)* approach the functionality of all systems is made uniformly accessible, hence introducing *model-based communication* approach. That is, every node can propagate on-demand its internal states, parameters, functions, sensor and control values as a hierarchically ordered *model* via the *Model service*. With Compass used as a carrier, all remote system models form a *mission model* (i.e. top-level digital twin), which represents the functionality, knowledge and current state of the *entire mission*. Standardized

*Compass services* were implemented to enable commanding, model access and modification, file transmission, unit testing and dynamic code execution on remote systems. The protocol design was inspired by several existing protocols and standards, such as *CCSDS PUS* and *MO* services, *CCSDS File Delivery Protocol* and *MQTT*.

## Tiny scripting service

The *Tiny scripting language* was designed to support dynamic code execution on ultra-low power microcontrollers, thus eliminating the need to perform hazardous in-orbit software updates, which otherwise would be required to enable the execution of on-demand designed experiments. A script can be used to read and modify local or remote model values, execute commands locally or remotely and perform local file system modifications. Due to its very low memory requirements (1 kB), the interpreter was enabled on all subsystems of the UWE-3, UWE-4 and NetSat satellites and is accessed via the *Tiny service*. Tiny scripts are used on the always-on OBC subsystems to enable *constraint-based on-board scheduler* and to execute in-orbit experiments.

## Compass OS and CompassNode

The protocol core and services were implemented for ultra-low power satellite subsystems (*Compass OS*) and for high-level machines (*CompassNode*). Minimum hardware requirements to run the Compass OS are: 16 bit architecture, 2 kB RAM and 10 kB ROM – thus allowing every existing satellite subsystem of the exemplary NetSat mission to enter the network as a discrete node. In the context of Compass, a satellite formation is an interruptive available subnetwork consisting of satellite subsystem nodes. Autonomous in-orbit formation control can be performed by the satellites using model-based communication, i.e. via mutual modification of particular model values (formation parameters).

## Compass Operations front-end

The *Compass Operations front-end* was designed to allow operators to access an existing Compass network and to monitor, control and operate all available nodes in the same way: satellites, ground stations, mission servers, test equipment and simulations. It provides numerous arrangeable views for accessing remote services for model modifications, commanding, file transmission, unit testing, Tiny script execution, etc. The integrated constraint-based *operations task scheduler* allows visual recording of complex satellite operations, which are automatically conducted during the overpasses. The front-end was extensively used during all phases of the UWE-4 and NetSat mission by test engineers, simulation designers, satellite software developers, ground station advisers and satellite operators.

## Realization

Based on the lessons learned from the UWE-3 mission and active work on the UWE-4, NetSat, QUBE, TOM and CloudCT missions, the philosophy of this work was to thoroughly verify *theoretical* considerations and approaches against their *practical feasibility* in an exemplary baseline mission. The modus operandi was a combination of top-down



and bottom-up approaches, i.e. continuous attempts to link the top-down path starting with mission aims (what must be done?) with bottom-up path beginning with low-level feasibility on specific hardware components (what can be done?).

The theoretical work and all implementations were applied to the existing satellite missions at ZfT and University of Würzburg, Chair VII Robotics and Telematics. Throughout this document references are made to the mission in which the mentioned approach, technique or implementation was successfully tested in-orbit (UWE-3, UWE-4 and NetSat). The UWE-4 CubeSat was launched in December 2018 and has successfully demonstrated its orbit control capabilities [KBS20]. Four 3U NetSat CubeSats were successfully launched in September 2020 and are used to date for in-orbit experiments. The satellite and ground software of both missions is based entirely on the technical outcome of this thesis:

- *Compass protocol* with numerous services
- *Compass OS* as C implementation of the Compass middleware for embedded devices; is currently active in-orbit on almost 60 individually accessible nodes
  - UWE-4: all subsystems of one flight and one engineering model – each containing 2x OBC, AOCS, 2x PPU and 6x Panels
  - NetSat: all subsystems of four flight and two engineering models – each containing 2x OBC, 2x AOCS, 2x Thruster Control unit (TU), Computing board and 5x Panels
  - numerous Satellite Development Kits (flat-sat)
  - controllers for ZfT’s high-precision motion simulators (turntables)
- *Tiny interpreter* as a part of the Compass OS used for in-orbit experiment execution and constraint-based task scheduling
- *CompassNode* as Java implementation of the Compass middleware for high-level nodes
  - *Compass Operations front-end*
  - *UHF Ground Station server* – one located at the University of Würzburg and one at ZfT
  - *Mission Server* – one for UWE-4 and one for NetSat mission
  - *MatlabNode* and *OrekitNode* for Matlab and Orekit simulation nodes respectively

## 1.4 Thesis Outline

Following chapters are organized in three groups: theory, implementation and testing&results. In the *State of the Art* chapter issues and challenges in an exemplary CubeSat formation mission (NetSat) are addressed along with existing common solutions for particular areas. Based on the gained insight and the practical constraints given by the ZfT (mission timeline, reuse of existing in-house satellite subsystems, available finances and workforce, etc.), a theoretical background for the new Compass protocol and software framework is elaborated in the *Approach* chapter: unification of communication and access to component’s functionality.

The theory chapters are followed by three implementations chapters and are depicted in figure 1.4. Referring to the stated requirements, the specification of the Compass protocol and standard Compass services is described in *Protocol* chapter. The protocol

specification is used as a basis for the implementation of software for space and ground systems. So, the implementation of Compass middleware for satellite subsystems, which is runnable on ultra-low power microcontrollers, is described in *Space Segment* chapter. The implementation of the Compass functionality for ground systems is described in the *Ground Segment* chapter: UHF ground stations, mission server and eventually Compass Operations front-end.

The eligibility of the proposed framework – both for space and ground systems – for the targeted exemplary CubeSat formation mission is handled in the *Testing and Live System Experience* chapter: protocol performance, in-the-loop testing, ground station performance and experience with single-CubeSat missions (UWE-3 and UWE-4).

The *NetSat Experience* chapter describes how all proposed approaches and implementations were used to enable operations of four 3U NetSats nanosatellites in-orbit: LEOP, recording-based auto-operations and in-orbit scheduler.

Eventually, a summary of all achievements and an outlook are presented in the *Conclusions* chapter.

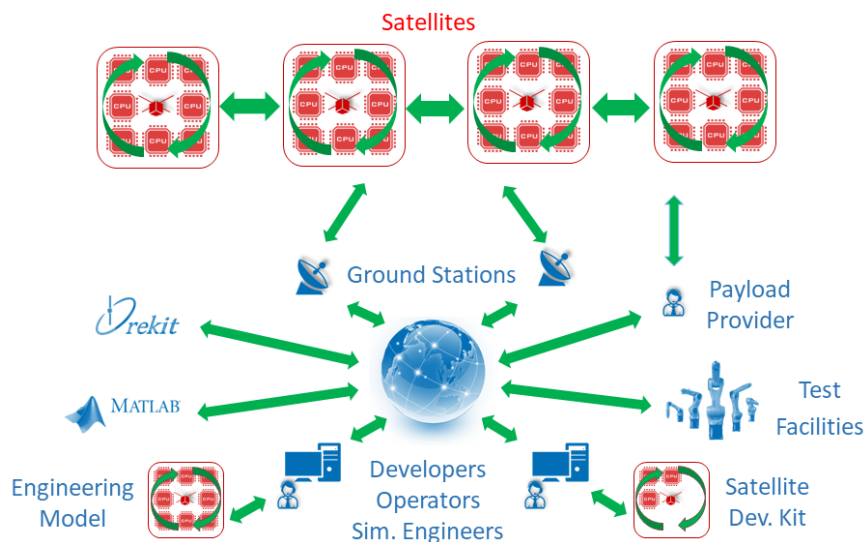


Figure 1.4: Coverage of the implementation chapters: *Protocol Chapter* (green), *Space Chapter* (red) and *Ground Chapter* (blue)

## 2 | State of the Art

In this Chapter challenges in an exemplary baseline satellite formation mission, more specifically the NetSat formation mission, and several selected state-of-the art solutions are discussed.

It is obviously not possible to propose a framework basis for ground and space systems for *any* possible current and future missions. Instead, the NetSat mission is used as a baseline, thus making the proposed solution best suitable for *comparable* missions with respect to the communication requirements, satellite with multiple individually accessible subsystems with one or more subsystems offering radio relay capabilities, suitability of the model-based approach for functional interfaces, and comparable composition of the ground-segment. Furthermore, the proposed framework aims at realization of all ground and space systems *without* the usage of external services, such as external ground station networks, external cloud-based solutions for Telemetry, Tracking and Control (TT/C) operations, etc. The concept is not to exclude external services per se, but to show the self-contained nature of the framework, which can practically be used as a basis for all mission-relevant systems. A framework-enabled infrastructure can nevertheless be modified or extended with further externally offered services, e.g. if a satellite provider does not possess own ground stations and is therefore relying on Ground Station Network (GSN) services.

A formation mission does not only introduce formation-specific issues, instead it also inherits challenges from a multi-satellite and single-satellite missions:

- *single-satellite*
  - Ground Station (GS) set-up: antenna, transceiver, server, mission server
  - Mission Control Center (MCC) set-up
  - satellite hardware development, testing, calibration and verification
  - satellite software development, testing and verification
  - ground-ground and ground-space communication paths
  - LEOP and operations
- *multi-satellite*
  - multiple satellites tracking
  - multiple satellites operation
  - multiple ground stations (optional)
- *formation*
  - formation simulation environment
  - Inter-Satellite Link (ISL)
  - formation-specific operation tasks

- autonomous in-orbit formation handling (depending on mission)

Challenges with respect to the hardware development are not handled in the scope of this thesis. Most of the mentioned challenges cannot be addressed in solely exclusive way, e.g. ISL may intersect with ground-space communication, if the same transceiver module is used for both segments – as a result of hardware (e.g. space, power) or project constraints (e.g. time, funds).

Several smaller and bigger formation and constellation missions have been performed until present. After top-level requirements in an exemplary baseline mission (NetSat) are defined, some existing formation missions will be shown, followed by a short discussion about on-board autonomy in five selected missions. Thereafter, an overview of current Internet of Things (IoT) network providers will be presented to emphasize the fact that IoT approach is gradually moving into the space segment, thus underlying the forward-looking inherent IoT nature of the Compass middleware. Eventually, some selected protocols and approaches used in ground and space segment will be discussed.

## 2.1 Requirements

The aim of the following state-of-the-art research is to find currently existing solutions for particular tasks mentioned in the *Overview* section. The state-of-the-art is ordered in three sections: communication protocols, ground segment and space segment. The suitability of the existing solutions is examined based on the defined requirements of the baseline mission, which must first be stated in more detail.

### 2.1.1 Baseline Mission

Since this work is focused on *nanosatellite* formation missions being conducted by *small teams* (UWE-4 mission was developed on average by three and NetSat formation mission by four full-time co-workers), the NetSat mission will be used as an example in the further discussion. Most of the hardware nodes (systems) involved in the NetSat mission are depicted in figure 2.1:

- A) four satellite flight models, each containing twelve *programmable* (2x OBC, 2x AOCS, 2x TU, Computing board and 5x Panels) and six non-programmable sub-system nodes: Thruster, 2x AX100 Radio and 3x EPS
- B) two satellite engineering models
- C) multiple operator's workstations as a part of the distributed MCC
- D) UHF ground stations, one located at the ZfT and another at the University of Würzburg – each consisting of a server, rotator, UHF transmitter and Terminal Node Controller (TNC)
- E) Orekit-based orbit simulator
- F) Matlab environment for simulating algorithms and post-processing of scientific data
- G) ZfT's dynamic test bench facility consisting of two precise motion simulators

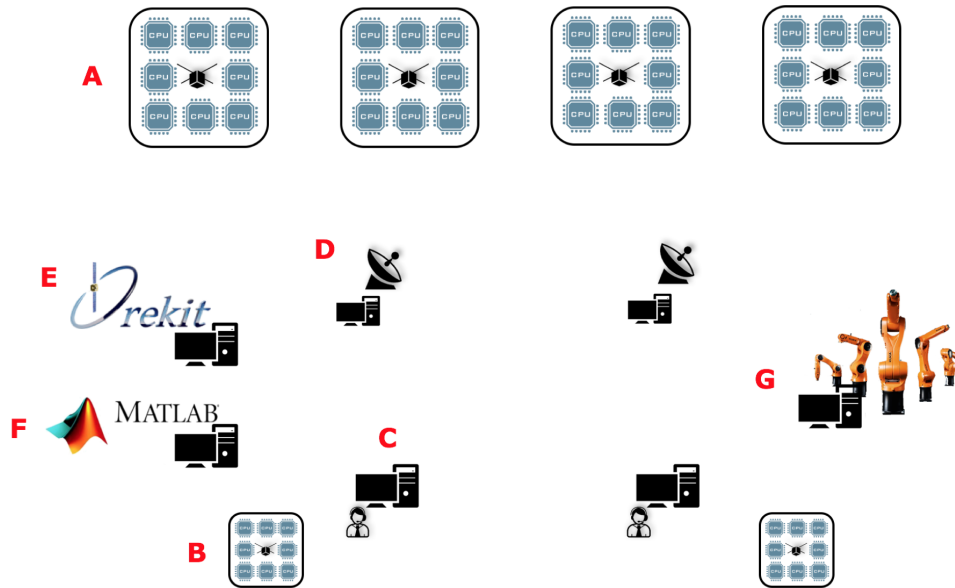


Figure 2.1: Simplified overview of the components involved in the NetSat mission. (A) NetSat flight models, (B) NetSat engineering models, (C) operator's workstations, (D) GS server, (E) orbit simulator, (F) algorithms simulator, (G) hardware test facility

In the scope of this work the latter three (*E*, *F*, *G*) are called *auxiliary* nodes, as they are mainly involved in the pre-orbit phase of the mission. Nonetheless, the integration of the auxiliary nodes during the in-orbit phase may strongly improve the quality and the outcome of the satellite operations. For instance, test facilities in combination with simulations may be used to partially test high-level formation commands before they are uplinked. All of the listed systems clearly have very different responsibilities and therefore different functionalities, e.g.:

- UHF GS: orbit propagation, hardware control (rotator, transmitter) and radio protocol selection during the overpass
- Mission server (MS): store relevant traffic in the database, act as a bridge between operators and multiple ground stations, accept packets received by the radio amateurs, etc.
- Operator's workstation (or MCC): monitor satellite housekeeping data, provide tools for satellite control, monitor the ground station(s) status, provide tools to control auto-operations and visualize received data
- OBC: act as a communication relay between subsystems and other ground and space (ISL) formation nodes, keep track of the satellite's healthiness, logging, beacon transmission

In comparable missions, the software of a particular system is typically realized either by using available software libraries or full-fledged software solutions for particular tasks, whereas mission-specific and satellite-specific functions (application code) require more in-house software development. This approach leads to an inhomogeneous software landscape

or to multiple enclosed *software and protocol islands* for particular task areas, which is manageable in missions where single systems are maintained and operated by several specialized companies (e.g. mission control, ground station, space segment software) – but introduces too much overhead for smaller missions, where smaller development teams must take care of all these system by their own. That is, to ensure the robust operation of the entire mission, all involved systems must be continuously monitored and controlled. For example, a wrong orientation of the UHF antenna can be the result of either too old Two Line Element set (TLE) data, time offset on the GS server or broken interface to the rotator. If different software solutions are used to fulfil these tasks, the operator is forced to check multiple logs or use different monitoring front-ends provided by the solution developer. Hence, *glue code* is required to unify the monitoring for fault-detection and to enable automatic recovery. The main goal of this work is to propose a middleware solution to homogenize software and protocols on all systems involved in a satellite mission. It on the one hand does not require to implement everything from scratch and on the other hand minimizes or even eliminates glue code.

## 2.1.2 Protocol

The desired protocol must appear in the protocol configuration of all space and ground systems (see figure 2.2) and possess inherent network capabilities. That is, irrespective of the – possibly inescapable – protocols on lower (OSI) layers of any particular system, all systems must be capable to communicate with each other using the same common protocol (requirement in *addressability*, *decentralized network* and built-in *routing capabilities*). This implies that the same protocol is used between ground nodes, for ground-space links, for ISL and for communication between satellite subsystems. The process communication between all systems must solely rely on standard services, i.e. the services must be designed to support any possible action and data exchange required in a satellite mission.

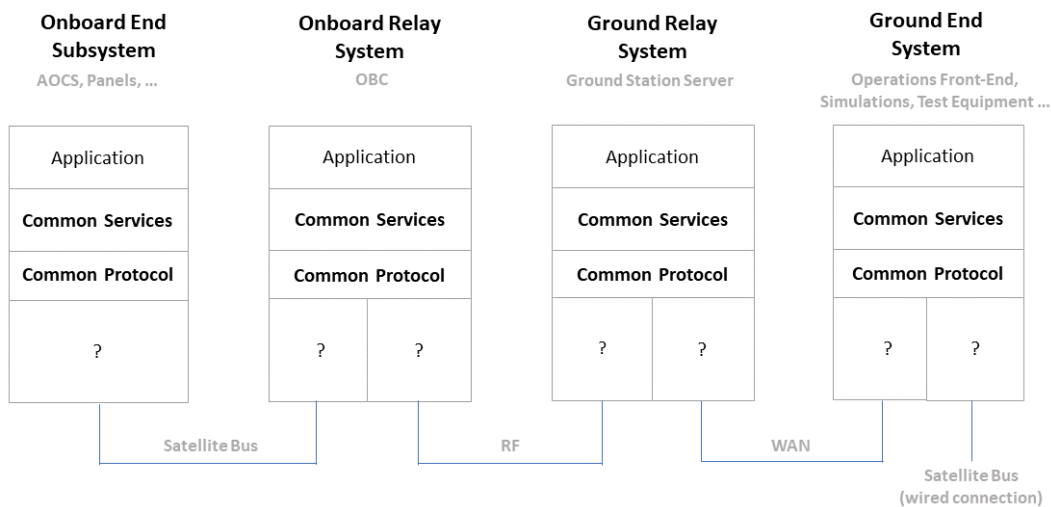


Figure 2.2: Desired common protocol configuration

All satellite subsystems must be individually accessible in the network – in contrast to the UWE-3 CubeSat, where the functionality of all subsystems is represented by the OBDH subsystem. To promote the logical cohesion of multiple nodes, e.g. multiple satellite subsystems being part of a satellite, the addressability must either support *subnetting* or some *grouping* concept. The routing within the (virtual) network should happen automatically, i.e. no manually created routing tables are mandatory to enter an existing network – requirement in *dynamic network*, *auto-routing capabilities* and *network awareness*. The network awareness must be able to detect neighbors as well as the *maximum speed* and *maximum packet size* a neighbor can handle, since both values can highly vary – depending on the node’s hardware (speed, available memory) and channel capabilities. To ensure the *scalability* of the network, the network awareness should only be exchanged between direct neighbors.

If more than one route exists between two nodes it may become important to be able to override default routes (auto-evolved shortest or fastest routes) for single packets (*per-packet route override*). This feature can, for example, be used for traffic *load-balancers* or when multiple formation nodes are currently directly accessible from the ground station, the operator can enforce ISL packet forwarding (for testing reasons) – without the need to manually change routing tables on all intermediate hops.

Due to the speed and packet size limitations of the satellite bus (I2C), the protocol overhead must be as low as possible, i.e. header fields should be in numeric format instead of text-based (*numeric address field*, *service designator*, *optional route entries*). To further decrease the overhead, the protocol should have *built-in header compression* functionality, i.e. protocol fields that become obsolete under particular circumstances, should not be transmitted, e.g. avoid transmitting subnet identifier between two systems within the same subnetwork or use *number compression techniques* if a value can be transmitted with less bytes, than defined in the header. The address range must on the one hand keep the protocol overhead minimal, and on the other hand enable unique addressability of all available systems in current and future satellite missions. More specifically, the protocol should provide at least 16 bit – better *32 bit global address range*, whereat it is advantageous when less address bytes are transmitted for small address values. Since the packets may also be transmitted via channels without integrated *error detection*, such as I2C and serial, the packet header should provide an optional/switchable *CRC field*. Integrated forward error correction mechanisms are *not required*, since for error-prone links the correction is usually performed by the transceiver hardware itself on lower protocol layers. From experience with UWE satellite hardware, other low-level channels, such as serial or I2C, have extremely low packet loss ratio – here a CRC check is enough to avoid processing of broken packets.

From experience with UWE-3 operations, a dedicated and optional/switchable *timestamp field* is required in the packet header. The availability of a timestamp field has several use-cases: detection of remote time deviation during the satellite operations, service-independent round-trip time calculation, and reconstruction of the recorded traffic. Furthermore, in UWE-3, UWE-4 and NetSat only the OBC subsystem has a real-time clock. Since most of the subsystems are only powered up on-demand, they wake up without any time knowledge. With timestamp in the header of an incoming packet (e.g. periodic network knowledge packet from OBC), a receiving subsystem can decide to synchronize its time knowledge up to some degree of accuracy.

Topic	Requirement	Ref.
Network	Addressability (32bit)	R-PRT-1
	Auto-Routing capabilities	R-PRT-2
	Subnetting	R-PRT-3
	Decentralized and dynamic network	R-PRT-4
	Scalability	R-PRT-5
	Per-packet route override	R-PRT-6
	Per-packet DTN capabilities	R-PRT-7
Network-awareness	Neighborhood detection	R-PRT-8
	Maximum packet size a neighbor can handle	R-PRT-9
	Maximum speed a neighbor can handle	R-PRT-10
	DTN-ability of a neighbor	R-PRT-11
Header	Compress-able or dynamic	R-PRT-12
	Low overhead	R-PRT-13
	Source and Target address fields (up to 32 bit)	R-PRT-14
	Optional intermediate addresses (route override)	R-PRT-15
	Service-id (or upper-protocol identifier)	R-PRT-16
	Timestamp field	R-PRT-17
	CRC field	R-PRT-18
Implementation	C version, runnable on 16 bit system, 2Kb RAM	R-PRT-19
	C version, same code-base for multiple platforms	R-PRT-20
	Java version for all ground systems	R-PRT-21

Table 2.1: Summary of protocol requirements



The protocol must offer Delay Tolerant Network (DTN) capabilities (*store-and-forward*), which can be switched on and off on packet-level. This functionality can obviously only be offered by systems with enough non-volatile memory, thus the network awareness should also automatically provide information about DTN-ability of every node within the network. The projected DTN-activated systems are: UHF ground stations (practically unlimited capabilities) and OBC subsystems in space (limited to N storable packets). The former is used to forward packets to satellites during the overpass, the latter to store packets and forward them later to currently inactive subsystems or another formation node via ISL.

Finally there are implementation requirements, i.e. the protocol stack and all relevant services must be runnable on embedded satellite subsystems (C) and on all ground systems (Java). The most limited subsystem is the UWE's PPU subsystem and NetSat's Thruster Control subsystem, both having only 4 kB of RAM and 64 kB of code space – from which at least 50% must be reserved for application code. Since several different microcontroller types are used (e.g. TI MSP430x and AtmelSAM families), the implementation must provide a proper hardware abstraction layer (HAL) to enable the reuse of the common code-base in all subsystem software implementations.

### 2.1.3 Services

The functionality of all space and ground nodes must be realized solely by using common services listed in table 2.2. In other words: the ground systems must be accessed and controlled in the same way as satellite systems, i.e. using the same services and front-end. The consequence is that all common services must be suitable for demanding space-ground links: low transmission rate (e.g. 9600 bauds UHF), possible high packet loss ratio, half duplex channels with comparable long up-down switching times, and highly asymmetric link properties (good downlink, bad uplink) – as has been frequently experienced in UWE and NetSat missions. Furthermore, all service protocols must be designed to support small packet sizes. Due to the memory and buffer size limitations of UWE and NetSat subsystems as well as limitations dictated by the satellite's radio, a maximum *packet size of 200 bytes* has been selected, at which the service protocols must be able to achieve their tasks. Services may use the network awareness information to select a suitable packet size for a particular remote network node.

In general a service should perform *without per-packet acknowledgements*, as it would drastically reduce the protocol performance via routes containing half-duplex links with long up-down switching times or highly asymmetric up-down packet loss ratio. Instead, where applicable, the service should request a *bulk-acknowledgement* for multiple packets.

To increase the performance of a service protocol without packet ACKs, the protocol should be designed without dedicated session start/end packets. A classic session based service interaction can be divided into three steps: send *start session*, perform actual service task, and send *stop session*. To reduce the service protocol overhead, the *start session* packet usually contains additional information that is indispensable for the actual task. In case the receiver misses the *start session* packet, all further incoming session packets become unusable. The solution is to design the service protocol in such a way that the receiver can always reconstruct the session from every service packet.

**Example: File upload without dedicated start/stop session packets**

System A is going to upload a file (`D:Test.xml`, 400 bytes) to system B. A transmits:

1. *start upload session D:Test.xml*
2. 200 bytes content (`[0 – 200]`)
3. 200 bytes content (`[200 – 400]`)
4. *stop session*

If B misses the *start session* packet, it cannot process incoming file content. This transmission scheme can only work properly, if every packet is acknowledged by the receiver.

Without dedicated start/stop session the transmission can be designed as follows:

1. write to `D:Test.xml` at 0: 200 bytes content (`[0 – 200]`)
2. write to `D:Test.xml` at 200: 200 bytes content (`[200 – 400]`)

Now, if B misses a packet, it can still process received ones – at the cost of higher protocol overhead. The receiver should also keep track of missing packets and use this information to send back a *bulk acknowledgement* – either on request or after some defined time period.

The goal is that any action on ground and space systems can be achieved with a combination of:

- remote command calls and either directly receive answers or pipe the answers to remote files (*Command service*)
- file exchange (*File service*, file operations can either be handled here or in the *Command service*).
- access remote variables, states or any entity that is representable as a key-value pair, and subscribe to value changes (*Model service*)
- remote execution of scripts, which in-turn can access other local and remote services (*Script service*)
- remote unit-test execution (*Unit-Test service*)
- receive (and store) incoming log messages (*Log service*)

Specific requirements for every service are listed in table 2.2.

The *Model service* should, for example, be used to access and/or modify satellite's state, modes, sensor values, actuator control values, calibration matrices, RTC time, beacon transmission rate, etc. Also actions, for which Command service initially comes to mind as a solution, can be achieved in more straightforward way with Model service. For instance, instead of sending `switch-on AOCs` command to the main OBC system, a Boolean AOCs variable can be set to `true`. Same service can also be used to monitor and control the ground station: current tracked satellite, loaded TLEs, camera image of the antenna, current frequency setting, etc. Another projected use-case of the Model service is to control other services.

The *File service* is used to exchange data amounts larger than the maximum possible packet size, e.g. download data or images recorded in orbit, download log files, upload scripts and software updates.

More complex operations must be enabled by the *Script service*: time-tagged and constraint-based actions. A script executed on a satellite subsystem must be able to access

Service	Requirement	Ref.
Overall	No per-packet ACKs	R-SRV-1
	No dedicated session start/stop packets	R-SRV-2
	200 bytes packet limit	R-SRV-3
	low overhead	R-SRV-4
Command service	Remote command (-batch) execution	R-SRV-5
	Answer piping to a local or remote file	R-SRV-6
	Answer piping as payload of user-defined service packet	R-SRV-7
File service	File upload and download	R-SRV-8
	Support for bulk-acknowledgements	R-SRV-9
	File system operations: format	R-SRV-10
	File operations: delete, write, read, size	R-SRV-11
Model service	Read and modify remote variables	R-SRV-12
	Variables are hierarchically structured	R-SRV-13
	Support for listener registration	R-SRV-14
	Supports value recording	R-SRV-15
	Supports groups, arrays, strings and binaries	R-SRV-16
Script service	Supports on-demand structure request	R-SRV-17
	Sand-box dynamic code execution	R-SRV-18
	Suitable for on-board task scheduler	R-SRV-19
Unit-Test service	Ability to access other local and remote services	R-SRV-20
	Remote execution of unit tests	R-SRV-21
Log service	Store local and incoming log messages	R-SRV-22

Table 2.2: Summary of protocol service requirements for space and ground systems

local and remote variables, perform file operations, generate log messages and create user-defined packets to access remote services. As soon as a script can access all available local and remote services, it is capable of performing any possible task of an operator. The aim of this thesis is to find a proper balance between ground-based and script-based operations.

The *Unit-Test service* should provide a list of available tests and return results of requested test executions. Unit tests can be used to verify the correct operation of the satellite software on different abstraction levels: sensor read-out, file access, output of an algorithm execution, satellite bus communication, etc. Dedicated Unit-Test service would allow the realization of a well-arranged operator's front-end, with hierarchically ordered available tests, progress bars, etc. Furthermore, the service would theoretically allow automatic test execution of newly inserted satellite subsystems during the development process.

### 2.1.4 Space Segment software

Space segment software represents a summary of all satellite subsystem software images: OBC, AOCS, active panels, auxiliary high-power computing board, and thruster control.

In the baseline mission, several different microcontroller types with different capabilities are used, such as Texas Instruments MSP430x and Atmel SAM families. To optimize the process of the satellite software development and to improve the software maintainability, all subsystems must share the same software baseline, i.e. the implementation of the protocol core functions and services (*middleware*) must be runnable on all subsystems. As a consequence of this, the middleware must be runnable on most limited subsystem microcontrollers of the baseline mission: *16 bit architecture, 4 kB RAM and 32 kB ROM*. On such limited systems the middleware may be configured to provide minimal support for particular functions, such as limited or deactivated DTN support or ability to hold only few routing entries (limited network awareness).

All protocol functions and services mentioned in table 2.1 and table 2.2 must be implemented in a common middleware code-base. Since the hardware of the baseline satellite subsystems is already given, further requirements towards the software implementation can be derived based on the protocol requirements – the top-level ones are shown in table 2.3. These requirements are *not bonded* to specific service implementations, i.e. specific protocol scheme, instead they are resulting either from the actual functionality the protocol or service must offer or indirectly from the functionality the service must have access to. This is important for the state-of-the-art analysis, as existing protocol implementations must be compatible with satellite hardware used in the baseline mission.

The limitations of the communication channels have already been mentioned in the previous section. The agreement is to support a packet size of at least 200 bytes, i.e. all internal and channel buffers must be designed in such a way, that *every* satellite subsystem is capable of forwarding a packet of that size. The ability of a satellite subsystem to be connected either directly to a workstation or be inserted into a flat-sat evolves from the core protocol capabilities (requirements *R-PRT-2*, *R-PRT-2* and *R-PRT-8*) – presupposed the subsystem offers the corresponding connector (serial, USB).

To allow the same code basis to be runnable on different platforms, all low-level system and hardware interfaces calls must be performed against a *hardware abstraction layer* (HAL): local time, interrupt service routines and hardware interfaces. In order to achieve its basic functionality, the implementation of the protocol core must have HAL-enabled access to local time and to all available hardware (communication) interfaces: I2C bus, serial interface and radio transceivers (via serial). Since the software of all UWE and most of the NetSat subsystems is implemented on *bare-metal* without multi-tasking support, the protocol and services must be runnable in a *single-threaded environment*. The projected middleware must also be *runnable on a Linux and Windows* operating systems to support software-in-the-loop tests.

Several services must have access to locally available file systems. The OBDH subsystem of the UWE-3 CubeSat had access to one SPI connected NAND flash chip. Due to RAM and ROM limitations, a dedicated UWE File System (UFS) file system was designed specifically for that flash chip. The updated version of the OBDH, is the OBC subsystem used in the UWE-4 and NetSat satellites. It has the same microcontroller but now has access to 6 FRAM and 2 NAND flash memory chips. In addition to that, every UWE-4 and NetSat panel is equipped with a microcontroller and a dedicated NAND chip. Hav-

Component	Derived Requirement	Ref.
Protocol core	Access to local time	R-SW-1
	Abstraction layer for available communication channels	R-SW-2
	Channel support for: I2C, serial, TCP/IP, UHF transmitter	R-SW-3
	Support for single-threaded software	R-SW-4
	Support Linux and Windows as host OS	R-SW-5
Command service	Concept for creating new commands	R-SW-6
	File system access to support answer-piping	R-SW-7
File service	Abstraction of persistent data storage (e.g. NAND, FRAM)	R-SW-8
	File system abstraction (drive and file concept)	R-SW-9
	File system implementation	R-SW-10
Model service	Concept for creating new model values	R-SW-11
	Model values must be hierarchically ordered	R-SW-12
	Concept for optional GUI hints (ranges, limits)	R-SW-13
Script service	Sand-box design with own stack	R-SW-14
	Support for multiple concurrent scripts	R-SW-15
	File system access to save/load persistent scripts	R-SW-16
	Ability to access existing C-functions	R-SW-17
Unit-Test service	Concept for creating new unit-tests	R-SW-18
	Unit tests must be hierarchically ordered	R-SW-19
Log service	File system access to store local and incoming logs	R-SW-20

Table 2.3: Embedded software requirements as derived from protocol requirements

ing different memory storage technologies (FRAM, NAND) and multiple available chips from different manufacturers, new abstraction layers for *data storage* and *drive/file access* become requirements.

The in-space script execution must be performed in a *sand-box* to ensure that the overall system stability is not harmed by faulty scripts. The interpreter must support *concurrent execution of multiple scripts*, i.e. execute a predefined amount of instructions in multiple scripts in round-robin manner. It must not rely on the multi-threading capabilities of the underlying operating system. The interpreter should also not be a closed system – the scripts must be able to *access existing C functions*, which could be registering at compile time.

Aside from some examples, the application code of particular satellite subsystems is not handled in this thesis. Instead, the projected middleware is intended to offer a basis for subsystem-specific application code that rely solely on the services defined in this chapter.

### 2.1.5 Ground Segment software

In the NetSat mission the ground segment consists of: two UHF ground stations, mission server, operator workstations, Matlab simulations, Orekit-based orbit simulation and two high-precision motion simulators (turntables). Satellite engineering models and flat-sats belong software-wise to the space-segment, as they are equipped with space software. The responsibility of every ground system is shown in the *Overview* section.

Some of the mentioned ground systems were already present (before this thesis) to support the UWE-3 CubeSat mission: one UHF ground station, several operator workstations and a Matlab workstation. The software on all systems was lacking a common basis, i.e. there existed no *common* way to access the functionality of the ground systems, nor was it possible to monitor their current state in a common way. During the UWE-3 LEOP phase the small UWE team (three co-workers) had great issues to point out the cause of the very poor communication. In addition to that, the mission network was consisting of only two nodes: one operator node and UWE-3's OBC subsystem. Neither was it possible to access further subsystems in the assembled satellite, nor was it possible for the ground and space systems to interact with each other. Nonetheless, the existing hardware and software infrastructure had to be used as a starting point of the transition towards a new set-up.

Aside from the Operations front-end, the actual application implementation for particular ground systems is not discussed in detail. Instead, the aim is to put the software of all ground systems on-top of a common middleware basis, such that all systems can be monitored, controlled and maintained with default services listed in *Protocol* and *Services* sections. Same services are also used by ground systems to access all other ground and space systems within the network, e.g. to perform cooperative processes. The middleware for the ground segment must fulfil all protocol and service requirements and must be made available as a library that can easily be integrated in the software of all ground nodes.

Component	Derived Requirement	Ref.
All ground systems	Share same middleware code-base (library)	R-GSW-1
	Can be monitored with default services	R-GSW-2
	Can be controlled with default services	R-GSW-3
	Persistent knowledge cache for remote systems	R-GSW-4
	Ability to connect to different mission networks	R-GSW-5
Operations front-end	Offer GUI for all available services	R-GSW-6
	Offer advanced GUI for specific model values	R-GSW-7
	Monitor current mission network state	R-GSW-8
	Simultaneous operation of multiple ground/space systems	R-GSW-9
	Auto-operations scheduler	R-GSW-10

Table 2.4: Ground middleware requirements

## 2.2 Formation Missions

Until present, several formation mission were flown or are currently in development. Some have a direct emphasis on a formation control (research formation control) – others are using formation control to realize the actual mission goals. A detailed survey of formation missions can be found in [D’E13].

### 2.2.1 CanX-4 & 5

CanX-4 and CanX-5 is a dual 8U CubeSat mission developed by the University of Toronto, Institute for Aerospace Studies/Space Flight Laboratory, and were launched in 2014 into Low Earth Orbit (LEO) orbit (635 km). The aim was to demonstrate autonomous formation control without required operations from ground. The projected maximum separation was defined at 1 km – whereas the S-band inter satellite link (10 kbps) was designed to cover up to 5 km separation.

Details on the satellite subsystems composition are described in [Orr+08]. An overview of the UHF and S-Band ground station can be found in [Kek06]. The operations are based on the Nanosatellite Protocol (NSP), which was developed at the University of Toronto specifically for CanX missions. In its usability it is rather comparable to the Cubesat Protocol from Gomspace (e.g. limited 8 bit address range). The NSP header consists of destination and source address (8 bit range), five *command* bits, *ACK* bit, *Package correlation* bit, *Reply* bit and up to 255 Bytes payload. Operations are performed with the *GNBControl* (Generic Nanosatellite Bus Ground Control Application) operations software, which entirely relies on the NSP protocol [Cho10].

### 2.2.2 Prisma

Prototype Research Instruments and Space Mission technology Advancement (Prisma) is a two satellite technology demonstrator mission led by the Swedish Space Corporation (SSC). Both satellites, *TARGET* (40 kg) and *MAIN* (140 kg), were launched in near-circular 720-780 km orbit on June 2010 and stayed coupled until the separation two months

later – more details can be found in [PDH10]. The primary aim was to demonstrate satellite formation flight, whereat the larger MAIN satellite is approaching the smaller one[D’E13].

Since TARGET has no dedicated ground link, only MAIN was controlled directly from ground. The communication is performed in S-band with 1 Mbit downlink rate and 4 Kbit uplink rate. The communication protocol is based on the CCSDS standard.

The ground segment consists of (more details can be found in [HPL09]):

- Operations Control Center (OCC) located in SSC Esrange (Kiruna/Sweden) and is responsible for routine operations and surveillance, and for TM/TC upload/download according to MCC instructions.
- MCC located in Solna (Sweden) and is responsible for the preparation of operations as well as for the simulation and validation of flight procedures.
- *ECC* (Experiment Control Centers) is not fixed to one location. ECCs prepare experiment scenarios for MCC and post-process experiment data.

Operations are conducted in RAMSES (Rocket and Multi Satellite EMCS Software), which has been developed by OHB Sweden. It supports CCSDS and ECSS standards and can, to some extent, be viewed as a middleware for ground systems. However, RAMSES protocol does not offer (virtual) network protocol capabilities and therefore relies entirely on the underlying UDP protocol [BS12]. RAMSES LAN is also used for the communication with the SATSIM satellite simulator, which was developed to validate the on-board software by generating code from Matlab/Simulink models[BNB12].

### 2.2.3 GRACE

Gravity Recovery and Climate Experiment (GRACE) joint-mission has been developed by National Aeronautics and Space Administration (NASA) and Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR) to measure Earth’s gravitational field. For that purpose two identically designed satellites (GRACE-1 and GRACE-2), each weighing 487 kg, were launched into 500 km orbit in 2002 and were separated by 220 km along their orbit track. The decommissioning started 2017, GRACE-2 and GRACE-1 performed atmospheric re-entry in 2017 and 2018 respectively. The successor GRACE-Follow-On (GRACE-FO) was launched in 2018 and was equipped with a laser ranging interferometer.

An overview of all ground systems can be found in [Pot21] and [AA02]. The TT&C is based on the CCSDS standard, whereat high-priority commands are handled directly by the telecommand decoder, thus surpassing entirely the OBC on-board software.

### 2.2.4 HawkEye 360 Pathfinder

HawkEye 360 Pathfinder Cluster Mission is developed by built by Deep Space Industries (DSI) and SFL for the HawkEye 360 company[36017]. The formation is composed of three 20x20x44 cm satellites, each weighing 13.4 kg. The satellites were launched into 580 km orbit in 2018 and eventually acquired an along track formation with 125 km satellite-to-satellite distance. The aim of the mission is to use RF technology for signal geolocation.

The satellites are based on the NEMO (Nanosatellite for Earth Monitoring) platform – an evolution of the GNB (Generic Nanosatellite Bus) used in CanX-4&5 mission.



The spacecrafts are operated from Multi-mission Spacecraft Control Center (MSCC) in Bangalore (India). The communication system, the protocol (NSP) and the operations software is tailored from the SFL standards (see CanX-4&5 mission).

### 2.2.5 MMS

The Magnetic Multiscale Mission (MMS) is developed by NASA with the aim to study the Earth's magnetosphere [Bur+16; Fus+16]. It consists of four 320x120 cm satellites (1360 kg each), which were launched into high elliptical orbit in 2015.

A detailed description of the ground segment is shown in [Bak+16]. The Deep Space Network (DSN) is the primary high-bandwidth communication link and is used to downlink scientific data to observatories. The Universal Space Network (USN) is used as a backup network in case DSN becomes unavailable. TT&C is performed via the low bandwidth Tracking and Data Relay Satellite System (TDRSS). MMS supports file communication based on the CCSDS CFDP protocol, which will be described in more detail in section 2.6.

### 2.2.6 NetSat

The NetSat mission consists of four 3U satellites, developed by ZfT, and are intended to be used for nanosatellite formation experiments. The satellites were successfully launched into LEO orbit in September 2020.

NetSat is the first nanosatellite *formation* mission (and the second nanosatellite mission after UWE-4) where all space and ground systems are based on the same uniform Compass middleware, thus promoting the IoT protocol approach for the entire mission landscape.

### 2.2.7 PROBA-3

The ESA Project for On-Board Autonomy-3 (PROBA-3) mission consists of CSC (Coronagraph Spacecraft, 320 kg) and OSC (Occulter Spacecraft, 180 kg) minisatellites. The primary objective is to demonstrate technologies for high-precision formation flying, required for space science, Earth observation and surveillance. The launch into an elliptical orbit is planned for 2022.

Details of the spacecraft design are shown in [Pen+20] and [Gal+19]. The inter satellite link will be enabled by TEKEVER's GAMALINK Software Defined Radio (SDR) transceivers (S-band). The ground segment is built upon heritage from previous PROBA missions (see [San+13] for more details), with Mission Operation Center being located in Redu (Belgium) and the ground station in Santa-Maria (Portugal).

### 2.2.8 CloudCT

The CloudCT mission is consisting of 10 3U nanosatellites and is developed by ZfT in cooperation with the Israel Institute of Technology. The aim is to perform tomography of clouds; the launch is planned in 2022 [SA20].

As in the previous NetSat and UWE missions, all satellite subsystems and ground segment systems will be based on the Compass system. The Compass protocol enables

a seamless combination of multiple existing mission networks into one *super-mission* network, thus allowing satellites from different missions to cooperate.

## 2.3 On-board Autonomy in Constellations

In this section operations strategies of five selected constellation missions are shortly discussed. Even though the solutions in “bigger” missions may be an overkill for smaller CubeSat satellite missions, they were nonetheless used as an inspiration for the design of the Compass middleware.

*Notice:* The content of this section was published in the *Background* chapter of [Nog+17].

### 2.3.1 SWARM

The Swarm three satellite constellation is operated from ESA’s operations centre ESOC. The Swarm Mission Control System (MCS) is based on SCOS-2000, deployed with one prime and one backup server per satellite. The MCS clients are connected to all prime/backup servers, and are configured in multi-domains, allowing an operator to quickly change between views to control the three satellites simultaneously from a single client. In routine operations only one satellite is commanded at a time [Die+16]. To minimise the chance of operator errors, each domain is clearly identified with a different colour.

Mission planning is performed weekly, and generates as output a set of time-tagged commands to be uploaded to the on-board queue for deferred execution, and another set to be executed directly from the MCS. The latter automates the activities that need to be performed at every ground-station contact. The current operations approach gives Swarm a nominal on-board autonomy of seven days.

On-board, the software is designed around the standard ECSS Packet Utilisation Standard (PUS) services for nominal operations and fault management, as for TSX-TDX. In case of a failure the satellites are placed in a safe configuration and the recovery is conducted by operators on the ground.

### 2.3.2 TerraSAR-X and TanDEM-X

TerraSAR-X (TSX) and TanDEM-X (TDX) are two satellites commissioned and operated by DLR. The satellites fly in close formation in low Earth orbit with inter-satellite distances varying between 150 and 500 meters [Mau+12]. The formation conducts radar measurements that are used to create an altitude map of the Earth [D’E13, p. 387ff.].

For TSX-TDC there is no real master/slave configuration, but instead an equal level of operations for each satellite. Each has a dedicated TT&C link and an MCS based on the generic SCOS-2000 software. Apart from one specific telemetry packet sent from TSX to TDX over the ISL, there is no Telecommand (TC) or Telemetry (TM) packet routing between the satellites. Mission planning is combined for both satellites, as the master timelines for TSX and TDX depend on each other. For example, bi-static acquisitions must be executed simultaneously on both satellites and sequential downlinks pose inter-satellite constraints that need to be respected. Modifications to the master timeline must therefore be synchronised between the two spacecrafts [Mau+12].

TDX carries an autonomous formation flying package (TAFF) able to perform autonomous formation control using TDX's cold gas propulsion system, and the navigation data transmitted by TSX over the ISL. To cope with this extra level of autonomy, the TDX Fault-Detection, Fault-Isolation and Recovery (FDIR) had to be extended to handle formation-level feared events, including indications of collision risk and unavailability of TSX navigation data over the ISL. While already successfully tested in closed-loop in-flight, the TAFF is not nominally used in operations [SGU12]. The on-board software is based on the ECSS PUS standard.

Both TSX and TDX use time-based on-board command queues for nominal operations (PUS service 11), complemented by the traditional set of services allowing to configure, monitor and execute on-board failure identification and recovery: PUS services 5, 12, 18 and 19 [SGU12].

### 2.3.3 Galileo

The Galileo Ground Control Segment (GCS) provides for overall monitoring and control, mission planning, flight dynamics and operations preparation for the 24+ GNSS constellation. The GCS is composed of several components, including the Spacecraft Constellation Control Facility (SCCF) and the Spacecraft Planning Facility (SCPF).

The SCCF, based on ESA's SCOS-2000 system, is responsible for the typical monitoring and control tasks, including: telemetry processing, command preparation, transmission and verification, on-board command queue management, limit checking and on-board software management. To ease day-to-day operations, the SCCF includes a component that automates many of the routine tasks. This component automatically processes telemetry, uplinks the command schedules to the satellites' on-board command queues and executes the ground-based schedules with the activities to setup the TM and TC links at each ground-station pass [Lor+08]. The automation logic is encoded in PLUTO procedures that interface with SCOS-2000 via SMF. The SCCF automation component interfaces with the SCPF to receive the latest constellation Short Term Plan (STP) and to provide the status of the schedules currently being executed. The STP is then used to generate all on-board and ground schedules.

Planning for the constellation is centralised in the SCPF that generates, among other products, the STP to be sent to the SCCF for execution. The STP covers typically a seven days window and contains all the operations that need to be performed on ground and on board for all the satellites in the constellation [HMF08]. This centralised planning system takes into account planning requests from different sources and with different priorities, from nominal operations (e.g. TT&C contacts) to manoeuvres and other special activities. The planner tries to maximise the contact time per satellite and has to guarantee a minimum number of contacts per satellite per day. In a typical week the SCPF plans more than 300 ground-station contacts and about 1500 tasks [HMF08].

The Operations Progress Monitoring (OPM) display is the entry point for constellation monitoring. The OPM provides an overview of the current status of the whole constellation, allowing as well to monitor the execution of the automated tasks on each of the satellites. The OPM introduces a Gantt view allowing to quickly browse the schedules on each of the satellites [Lor+08].

On the on-board software side, Galileo makes use of the standard set of ECSS PUS services for telecommand on-board queue, software management and FDIR [RSW12].

### 2.3.4 Planet

Planet’s Flock 3p constellation consists of 88 nanosatellites. Mission operations are highly automated with the ground and space operated over interfaces based on *microservices*. Software updates are reviewed weekly and tested on a limited number of satellites in orbit before being deployed to the rest of the constellation. The satellites run Ubuntu Linux on a x86 cpu, making it possible to easily share software between the space and the ground segments. Command and control is done using the so called *Ops Scripts*, which are typically deployed to the satellites one or more times per week [Zim+17].

In routine operations the satellites follow a simple pattern: (a) when over land point towards nadir and take images; (b) whenever a ground-station contact is scheduled switch on the X-band transmitter and downlink the images. When not in one of the previous modes, the satellites are either doing some periodic housekeeping activities (e.g. optical calibration) or are placed in a predefined attitude such as to make use of the differential drag to control the satellites’ relative orbits [Zim+17; CK16; FHM15]. From the available literature it is not clear how these operations are actually scheduled on board, and whether they make use for example of time-based or position-based scheduling. Given that the *Ops Scripts* are updated regularly, it is reasonable to assume that every update contains a new schedule generated by a centralised planning engine on the ground, and that is valid for the next few days to one week.

To make a satellite operational requires a series of commissioning activities that include, among other things, the calibration of the attitude determination and control system (ADCS). The speed with which new satellites are commissioned and made operational is important from the business perspective, but also to test as fast as possible new software and hardware in time so that any corrections can be injected in the next batch of satellites [Zim+17]. Independently of potential issues found during commissioning, the speed with which a new satellite can be commissioned is mostly constrained by the limited number of ground station passes and the limited number of operators. To speed up these activities the satellites in Flock 3p were launched with on-board software scripts that automate the initial configuration and checkout, reducing the number of required passes for commissioning under nominal conditions to two.

In case a failure is found during execution, the satellite is placed in a safe configuration and waits for ground [Zim+17]. From the available literature these scripts seem to be in all akin to the typical on-board control procedures (OBCP), that execute a pre-programmed set of instructions (commands) against a pre-programmed set of criteria (e.g. telemetry checks). These scripts are then run on-board as part of a periodic automated task [Zim+17].

### 2.3.5 OneWeb

The OneWeb 600+ broadband small satellite constellation [Azz16] is planned to be operated from two centres, one in the United Kingdom and one in the United States. The satellites are built by Airbus and make use as well of the pus standard [GMV16]. The operations centre is composed of the typical elements, including command and control, flight dynamics and mission planning [Mor+17].

The Command & Control element is developed by GMV, and its based on GMV’s *hifly* system with some key updates to efficiently manage the constellation. At an archi-

Network	Company	IoT Payload	S/Cs (planned)
Starlink	SpaceX, USA	Ku, Ka band at 100 Mbit per user	1081 (30000)
Myriota	Myriota, Australia	VHF/UHF, 20 Bytes per message	4 (50)
Astrocast	Astrocast, Switzerland	L-band, 1 kB per day	5 (80)
Kepler	Kepler Communications, Canada	Ku band	3 (140)
Swarm	Swarm Technology, USA	VHF, 196 Bytes per message	36 (150)
kineis	France	UHF, 30 Bytes per message	8

Table 2.5: Overview of IoT missions

tectural level functions have been split into two groups: fleet and single satellite. The fleet level functions, deployed as one redundant set of tools for the complete fleet, includes components for fleet-level scheduling and awareness, and fleet-level MMIs that allow an operator to monitor and control multiple satellites simultaneously. At the single satellite level is the classical telemetry, telecommand and out-of-limit functions, deployed as one redundant *hifly* core instance per satellite [Mor+17]. The main panel for quick fleet status monitoring is the *fleetDashboard*. This view provides in a glance the status of each of the satellites using coloured cards. The cards can be further stacked to save space, providing then summary information on all the satellites in the stack [Mor+17]. A scheduler panel provides a Gantt view of the executed and scheduled tasks for each satellite.

The mission planning element is provided by *SciSys*, and is based on their *Pleniter@* software [Sci16]. Planning is done daily and is mostly automated. For planning efficiency purposes the satellites are divided into different groups depending on the current activities being carried, e.g. LEOP, routine operations or orbit raising manoeuvres.

For the OneWeb system no sources could be found that detail the on-board software design and to what extend the standard PUS services are actually implemented. It is to expect that they make use of the standard set of services based on a timetagged command queue, complemented with configurable parameter and event-based monitoring that triggers pre-defined on-board control procedures for failure handling.

## 2.4 IoT/M2M Missions

In recent time nanosatellite IoT missions gained in popularity. In contrast to this work, these missions are *not aiming* at using IoT technologies to realize the actual mission. Instead, they are offering IoT-enabled communication services to other existing systems distributed on ground. Several selected missions are discussed here – a more comprehensive overview can be found in [Hof+19] and [BP19]. A detailed comparison between Starlink, OneWeb and Telesat is shown in [PCC18]. A detailed survey on recent advances and future challenges of CubeSat communication is described in [Sae+20].

Aside from Starlink network, which is mainly aimed at *Internet from Space*, most of the mentioned IoT satellite missions are designed for smaller packets (circa 30-200 Bytes) and are thus only suitable for rather simple TT&C tasks. Nonetheless, in-orbit IoT services may in future become a game-changer for nanosatellite missions, as they can drastically reduce the complexity of the required ground segment.

### 2.4.1 Starlink

SpaceX *Starlink* is a large constellation mission, which provides almost worldwide high-speed satellite Internet access. It is currently the largest constellation with 1081 satellites (March 2021) and is planned to be enlarged to up to 30000 units. All satellites were launched into LEO orbit (550 km). Each Starlink spacecraft is equipped with Ka and Ku band transceivers, weighs 260 kg and is orbiting in 550 km LEO [McD20].

The mission is not aimed at IoT devices per se – the required ground terminals are comparable large, consisting of an 59 cm phased-array dish antenna, a router and a power supply. Due to the size of the terminal and the very high power consumption (over 100 W), the system is currently not suitable for CubeSats. Apart from *reddit Ask Me Anything* sessions offered by the SpaceX software development team, it is hard to find any information about the Starlink’s on-board software design. Every spacecraft has multiple computers running realtime-optimized Linux operating system, which in parts is shared with Falcon and Dragon. Also the alerting system is shared between Starlink and Dragon. So, it seems that SpaceX has chosen a very unconventional way to implement the software of space and ground segment and reuses many existing solutions from Falcon and Dragon[tea21].

### 2.4.2 Myriota

Myriota is a planned constellation of 50 3U LEO CubeSats, each equipped with in-house SDR-based VHF/UHF payload transceivers. By now, the company has purchased four in-orbit satellites from *exactEarth* to start initial service operations and plans to expand to 25 spacecrafts by 2022. Customer’s ground IoT end nodes can be equipped with small form-factor (34x21 mm) transmitters (currently uplink-only) and can expect 4 Myriota passes per day and transmit 20 Bytes messages with up to 3 hours delivery time [Myr21].

The satellites are developed by Tyvak Nano-Satellites Systems (US) – and will probably be based on the *Trestles* platform [Sys21b]. The ground segment is currently consisting of 6 ground stations from exactEarth and are located in Canada, US, Norway, Singapore, Panama and Antarctica. Unfortunately, no software or protocol details could be found for the space and ground segment.

### 2.4.3 Astrocast

The Swiss Astrocast LEO (550 km) constellation is currently consisting of 5 3U nanosatellites and is planned to be enlarged to up to 80 satellites. The bi-directional communication to customer’s IoT ground nodes, which must be equipped with tiny *Astronode S* transceivers, is performed in L-band [Ast21].

The ground segment is developed by Leaf Space and is consisting of 12 antennas. The mission operations are controlled by the Elveti Mission Control System from Solenix [Sol21]. Elveti uses CCSDS and PUS protocols, an overview of the systems is shown in figure 2.3.

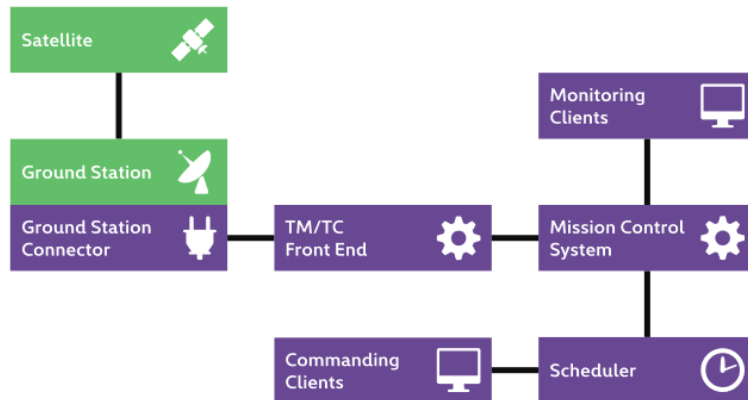


Figure 2.3: Solenix Elveti Mission Control System overview. Image source: [www.solenix.ch](http://www.solenix.ch)

#### 2.4.4 Kepler

The Canadian Kepler Communication company is planning to launch up to 120 3U and 6U nanosatellite into LEO (575 km), from which 5 spacecrafts are currently in space. The aim of Kepler is to provide communication links to other satellites. With ISL capabilities and store-and-forward approach the data is transmitted with much lower latency as, for example, compared to Myriota.

The satellites are developed by SFL (University of Toronto), based on their SPARTAN 6U-XL platform[Tor21], and are equipped with SDR-based Ku band transmitters. The ground stations are installed by the company itself, some being already set-up in Inuvik (Canada), Svalbard (Sweden) and Awarua (New Zealand). No further details on the software or protocol for the space and ground segment could be found.

#### 2.4.5 Swarm

Swarm is a constellation of currently 36 (150 are planned) 1/4U SpaceBEE spacecrafts, each weighing only 400 g. Ground nodes that are equipped with a small form-factor Swarm Tile, can transmit up to 196 Bytes in a single packet and up to 25 packets per day in the VHF band (bi-directional). Some minor information about the ground station set-up can be found in the obligatory FCC form under [Inc17]. The plans were to bring the total amount of ground stations to 30 by the end of 2020. Also here, no further details on the software or protocols used for the mission operations could be found.

#### 2.4.6 kineis

The kineis constellation will consist of 25 nanosatellites by 2023 and will be served by 20 ground stations around the globe. It will provide bi-directional communication to ground IoT nodes and offer location services down to 150 m accuracy.

The satellite development is performed by Nexeya (platform), Syrlinks (payload) and Thales Alenia Space[Kin18]. The demonstrator satellite ANGELS (Argos Neo on a Generic

Economical and Light Satellite) is currently in orbit and is used to verify the flight software. It is operated in S-Band with CCSDS protocol. Details on the ground segment are described in [Sal+18].

## 2.5 OPS-SAT and MO services

Mission Operations (MO) is a service-oriented approach for operation of spacecrafts and payloads[Sec10]. It includes:

- managing of the on-board software
- monitoring and control of satellite subsystems
- scheduling and execution of mission operations
- satellite performance analysis and reporting
- attitude and orbit determination, prediction and manoeuvre preparation

The OPS-SAT mission is funded by the ESA General Support Technology Programme. The satellite was successfully launched in December 2019 into 515 km LEO orbit and is controlled from the ESOC. The goal of the mission is to engage the new MO service-based software architecture in space[CEK15; Eva+16; Coe17]. For this purpose a new *NanoSat MO Framework* was developed by the Graz University of Technology in partnership with the European Space Agency. The framework is Java-based, i.e. it can only be used on high-power satellite subsystems and is therefore not suitable for subsystems based on ultra-low power microcontrollers. In OPS-SAT only the payload computer, which is powered by the MitySOM system-on-a-chip (performance comparable to Raspberry PI zero), can be operated with MO services. Most satellite bus components are from Gomspace (NanoCom, NanoMind, EPS) and are accessed with the Cubesat Protocol (CSP)[Alm14].

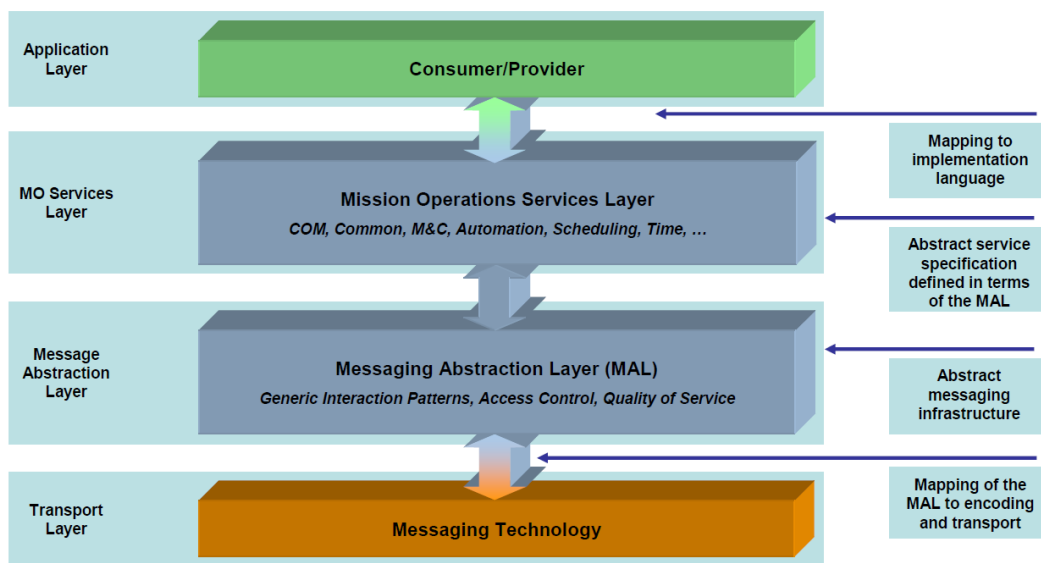


Figure 2.4: Overview of the MO Service Framework (image source: [Sec10], p.19)



As soon as the NanoSat framework has been extensively tested in space, it may become an option for advanced MO-based operations executed on a high-power computation subsystem, which is only activated on demand (cf. Computing Board in the NetSat mission).

Even though the OPS-SAT mission is not a CubeSat formation, it shows the “agency-way” to combine modern nanosatellite approaches with the existing agency’s software and ground infrastructure. A more detailed description of the spacecraft and the ground segment composition can be found in the [Coe17, p.128-129]. Compass framework was therefore designed with OPS-SAT approach in mind. OPS-SAT is focusing on MO-based operation of a single high-power satellite computing subsystem, whereas Compass – with its inherent IoT nature – is targeted at bringing all ground and space system to the same basis.

## 2.6 Protocols

Every satellite mission is composed systems distributed along the *ground segment* and *space segment*, thus resulting in different types of communication paths:

- *ground-ground*: link between an operation workstation (or MCC) and the actual ground relay (ground station)
- *ground-space*: connection between a ground relay and a satellite, plus an optional payload-specific link between the payload provider and the payload’s transmitter
- *space-space*: inter-satellites communication (ISL)

In addition, following *auxiliary ground-ground* paths may be in use:

- *radio amateurs*: receive packets from external ham radios
- *orbit simulation environment*: e.g. communication between a satellite engineering model and Orekit to perform in-the-loop simulations
- *Matlab simulation environment*: e.g. communication between Matlab and a satellite panel to perform sun-sensor calibration
- *test equipment*: e.g. communication between a Matlab calibration algorithm and motion simulators (turntables)

In the following, only protocols beginning with the OSI Layer 2 (Data link layer) are discussed as well as protocols that are constrained by the available COTS transceivers. For wireless communication paths only those protocols are considered that are also feasible for UHF transmission, as only UHF-based communication was of particular interest in all satellite missions (UWE-3, UWE-4 and NetSat) at ZfT during this thesis. This chapter is mainly focused on protocols that are usable on pico/nano satellites, i.e. realizable on very low-power components (cf. requirements in table 2.1). Also the space protocol recommendations by the *Consultative Committee for Space Data Systems* [AA14](figure 2.5) were taken into account during the elaboration of a new protocol.

### 2.6.1 CCSDS Recommendations

The CCSDS recommendations propose several stack configurations with following protocols as a common base for ground and space systems ([AA14], p.4-5,6,7):

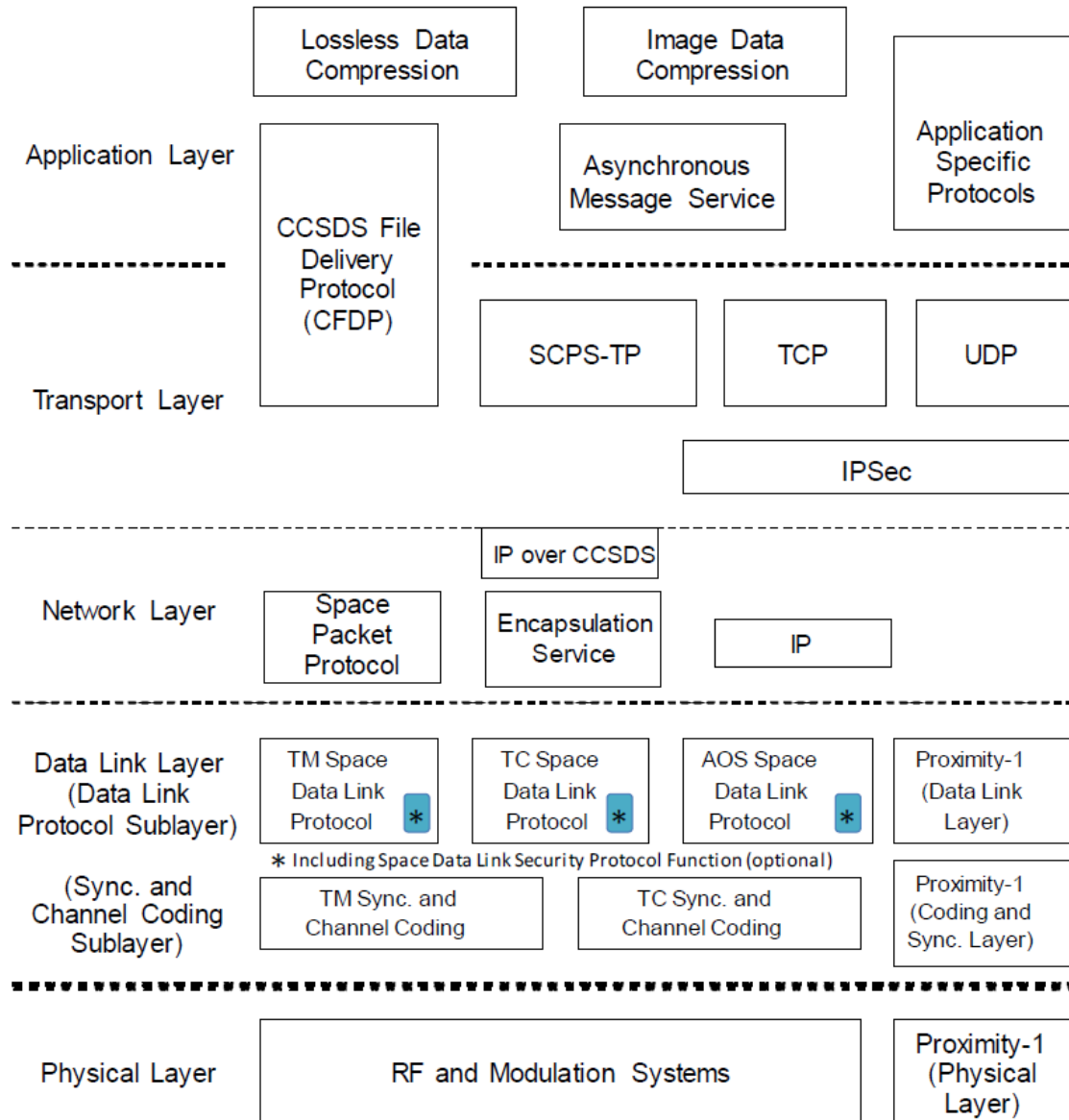


Figure 2.5: Overview of space protocols recommended by the CCSDS (image source: [AA14], p. 2-4)

1. Space Packet Protocol (figure 2.6)
2. IP protocol (figure 2.7)
3. CFDP protocol (figure 2.8)

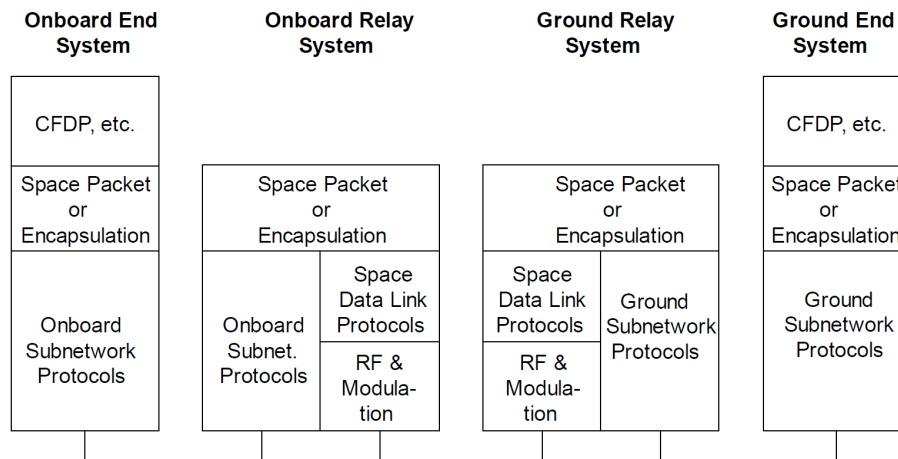


Figure 2.6: CCSDS recommendation with Space Packet Protocol as End-to-End forwarding (image source: [AA14], p. 4-3)

*Space Packet Protocol* (SPP) is designed to support all possible space applications – even to the point of fractionated spacecrafts[Sec20]. SPP relies on predefined data-layer protocols – either TM, TC, AOS, Proximity-1 or USLP[Sec15c; Sec15b; Sec15a; Sec13a; Sec; Sec13b]. However, the SPP lacks the definition of path, network or routing functionality – these functions must be provided by the underlying protocols.

The *IP* protocol is not an option for many low-power nanosatellite subsystems for several reasons. The implementation of the full IP specifications would consume too much program and main memory space. For example, the segmentation and routing capabilities would by far exceed the 4KB main memory limitations of some UWE and NetSat subsystems (Thruster control, Power Processing Unit and OBC). Using a very limited implementation of the IP protocol would ruin its main advantage – compatibility with available IP driven networks. Using the full implementation only on higher-powered systems would require another common protocol to allow low-powered systems being a part of a uniform common network. Nonetheless, using IP or TCP/IP protocol as a carrier between ground systems is from the *practical* point of view the only available option.

*CCSDS File Delivery Protocol* (CFDP) is similar to FTP protocol and is also suitable for *file-based* communication. Several ESA and NASA missions were equipped with CFDP, such as *Deep Impact*, *MESSENGER*, *James Webb Space Telescope*[Wil12]. CFDP introduces a concept of a *Virtual Filesystem*, which avoids requirement for traditional disks or file systems [Ray03]. File based communication may not be suitable best for any kind of ground-space communication – as every possible transaction would need to be reduced

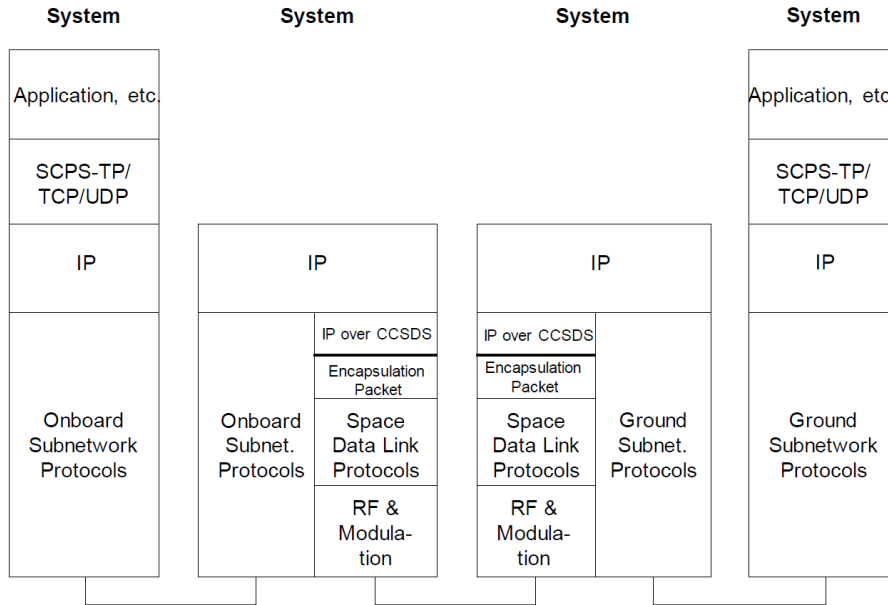


Figure 2.7: CCSDS recommendation with IP protocol as End-to-End forwarding (image source: [AA14], p. 4-5)

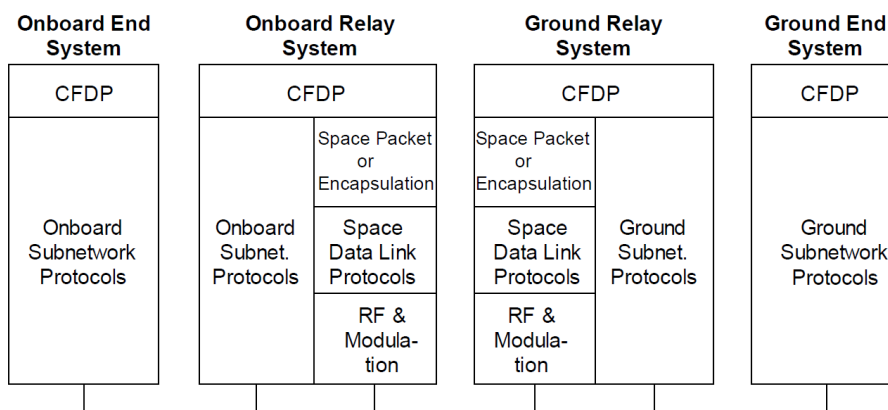


Figure 2.8: CCSDS recommendation with CFDP protocol as End-to-End forwarding (image source: [AA14], p. 4-5)

No.	Service name
1	Telecommand Verification Service
2	Device Command Distribution Service
3	Housekeeping and Diagnostic Data Reporting Service
4	Parameter Statistics Reporting Service
5	Event Reporting Service
6	Memory Management Service
8	Function Management Service
9	Time Management Service
11	On-board Operations Scheduling Service
12	On-board Monitoring Service
13	Large Data Transfer Service
14	Packet Forwarding Control Service
15	On-board Storage and Retrieval Service
17	Test Service
18	On-board Operations Procedure Service
19	Event-Action Service

Table 2.6: Standard PUS services

to file operations. That is, CFDP is rather not suitable as the *only* service for uniform service-based communication, but can still be in use for file-specific transactions. The protocol relies on the dedicated session start and stop messages, which contradicts the requirement *R-SRV-2* (see table 2.2). Despite this limitation, CFDP specification was used as an inspiration to define a File Service on top of the common protocol.

## 2.6.2 Packet Utilization Standard services

*ESA Packet Utilization Standard* (PUS) standard addresses the utilization of telecommand and telemetry packets to support monitoring and control of remote satellite subsystems and payloads[Sec03b]. The standard aims to satisfy fundamental operational requirements and is not intended to be used for video or audio data, nor is it designed to be used for monitoring and control of ground systems. All standard PUS services are shown in table 2.6. A service specification defines:

- requests that can be handled by the service
- success or failure notification as well as event reporting during the service execution
- internal service activities required to process a request or to detect and process service-related events

The company *12G Flight Systems*, founded in 2018 in Sweden, offers commercial implementation of PUS services usable for CubeSats [Sys21a]. Using a commercial version (provided by a newcomer company) is too risky, especially given the fact that the software will be used on all ground and space systems in all current and future mission of ZfT. New implementation of PUS services was considered as a not forward-looking option, since PUS is slowly superseded by *Mission Operations* (MO) services[Mer+10].

### 2.6.3 Ground Segment Protocols

In contrast to the space segment, ground systems are in general not suffering from communication problems. TCP/IP can be assumed as a standard *transport+network* protocol combination between distributed WAN-activated ground systems, i.e. all further mission specific protocols are practically always placed on top of TCP/IP. Since TCP/IP is not the best choice for the space-segment, it can not be used to realize the same network capabilities in space. That is, if common (virtual) network is desired that combines all ground and space systems, an additional virtual network (and transport) protocol is required.

The actual process-communication between network activated space and ground systems is performed via protocols that are placed above the network/transport protocol. There is no common higher-level ground segment protocol that supports all mentioned tasks, such as distributed ground station control, simulation access, etc. – and at the same time support space systems. Instead, several protocol stacks are used for different system types to enable particular functionalities. For distributed ground stations, the GENSOO [SK07] software module and the corresponding *GS Remote Operation Web Service* (GROWS) exist to support remote ground station control. The GENSO idea of GSN was proposed in 1998 by the University Space Systems Symposium (USSS). After the construction of the stand-alone ground stations with no common compatibility between 1998-2006, a working group started under the support of *UNISEC*. Eventually, software was released in 2007, consisting of *Ground Station Management Service* (GMS, figure 2.10) and *GS Remote Operation Web Service* (GROWS, figure 2.11). The GROWS service can be activated only on Windows-based systems and it lacks ground station scheduling capabilities, thus limiting its usability.

Another mentionable communication path is between the radio amateurs and the satellite providers (operators). Especially when satellite providers have no access to a worldwide ground station network, incoming packets from radio amateurs all around the world may drastically improve the outcome of a mission. At the beginning of this work there existed no common standard for automatic reception of ham radio packets – instead either mission specific applications were made available that had to be executed by the radio amateurs, or the packets were transmitted via e-mails. For example, the AMSAT-UK offered a *FunCube Dongle*, which could be purchased by any person to receive and decode data from the FunCube satellite [Sch13].

To improve the scientific outcome of the UWE missions, one of the results of this thesis was the *Simple Downlink Share Convention* (SiDS)[Dom15b], which is used by radio amateurs to automatically pass received raw satellite packets (UWE-3 and UWE-4) to the UWE Mission Server. Throughout this thesis the *SatNOGS* platform increased in popularity in the radio amateur community. It is used as a central storage platform for satellite packets received by external ground stations all around the world. SatNOGS also provides SiDS interface for packet reception.

The *SONATE 3U CubeSat*, which was developed under the supervision of Prof. Hakan Kayal at the University of Würzburg, Chair VIII, was successfully launched in 2019. One of the main goals of SONATE was to show extended autonomy with its ASAP payload. The development of the mission was supported by a client-server based simulation software, which used a custom protocol on top of the TCP/IP [Her17]. It is one further example

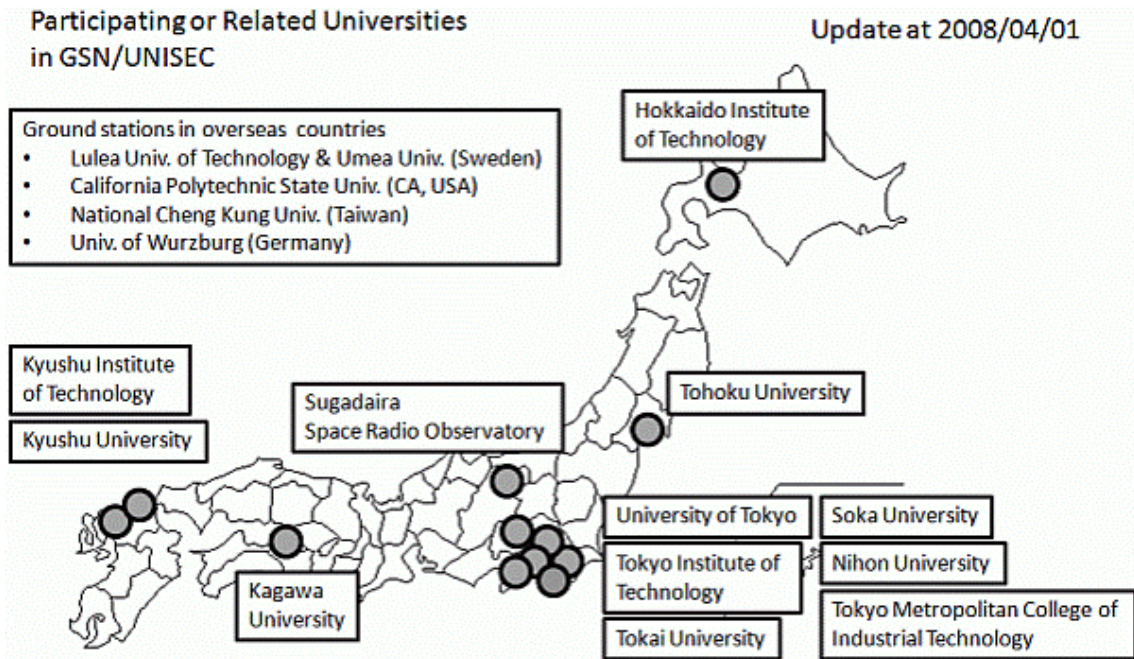


Figure 2.9: GENSOO: Participating and Related Universities (image source UNISEC global)

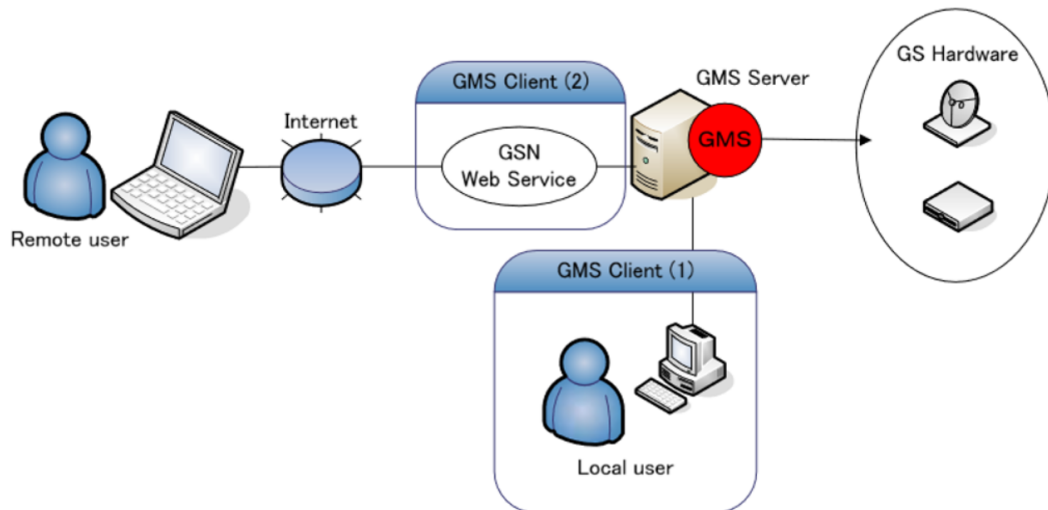


Figure 2.10: GENSOO: Ground Station Management Service (image source UNISEC global)



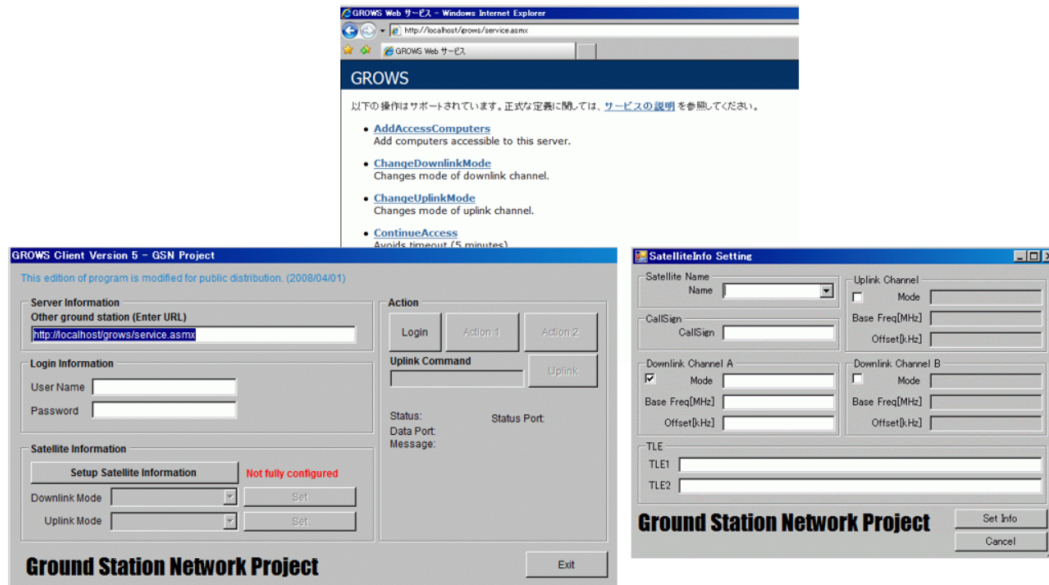


Figure 2.11: GENSOO: GS Remote Operation Web Service (right) (image source UNISEC global)

of the common concept in the satellite development community to elaborate different protocols for different tasks.

## 2.6.4 Ground-Space Protocols

The *ground-space* communication is usually to some degree constrained by the transceiver being used in the space segment. For example the *Lithium* UHF transceiver from *Astrodev* (used in UWE-3 and UWE-4) supports the transmission of any byte-array, therefore enabling the use of any Layer 2 protocol. In contrast to that, *NanoCom AX100* transceiver from *Gomspace*, which was selected for NetSat, is limited to Gomspace's in-house *CubeSat protocol* (CSP)[GOM11], thus a ground station *must* use CSP to contact the satellites. Since the CSP configuration of the AX100 radio only supports 5 routing entries and lacks broadcasting capabilities, it cannot be used to cover the entire NetSat mission.

On higher OSI layers, space protocols as recommended by the *CCSDS* can be established to enable ground-space communication[Sec03a]. So the promising *CCSDS Mission Operation services* approach has been shown in the *ESA's OPS-SAT* mission as a service protocol (MO-services) [CEK15]. However, the implementation is aiming at comparable high-power satellite subsystems and can therefore not be used for all satellite (bus) subsystems.

For the sake of simplicity and to meet specific mission requirements, many CubeSat providers (e.g. Universities) implement their own space protocols. The *MOVE-II* satellite from the Technical University of Munich (TUM) is equipped with their in-house *Nanolink* protocol, which has been designed for ground-space links with high asymmetry and weak signal quality [Lan+15]. The *Missionlink* protocol of the UWE-3 satellite has very dense



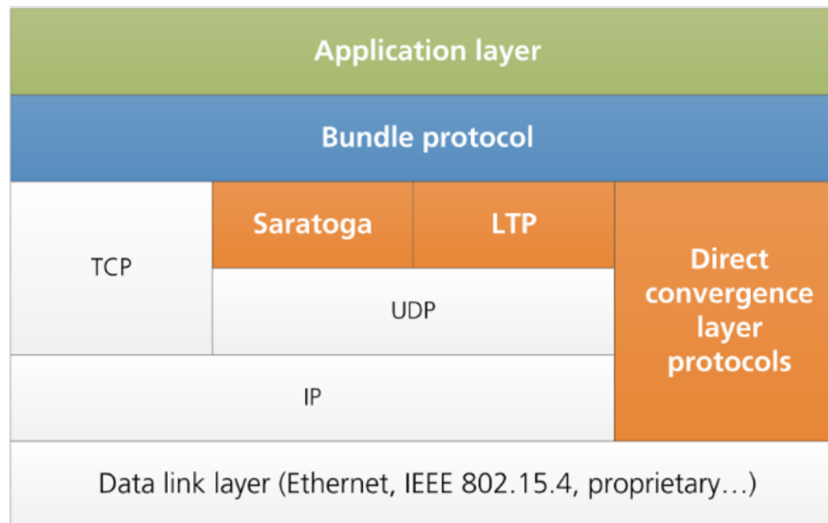


Figure 2.12: Available DTN protocols

overhead but lacks addressability (no network possible). The *UNISAT-6* satellite from GAUSS Srl [DR18] and the *SOMP2* from the Technical University of Dresden use the AX25 protocol to enable data reception by the radio amateur community. A counterexample is the *Flying Laptop* from the University of Stuttgart, which is equipped with CCSDS protocol and is operated from a SCOS-2000 enabled MCC [Kli16] – but being a bigger satellite ( $60 \times 70 \times 87 \text{ cm}^3$ ) it is not representative for most pico-/nanosatellite missions.

In general all ground-space links suffer from higher error-susceptibility and from the limited availability of space nodes during specific time periods. For this purpose several available *Delay-Tolerant-Networking* protocols can be utilized (figure 2.12)[AA14], for example:

- *Licklider Transport Protocol* [AA15b]
- *Saratoga Convergence Layer Protocol* (figure 2.13), e.g. used on-board the Disaster Monitoring Constellation (DMC) satellites for image transmission[Woo+07]
- *Bundle Protocol* [AA15a]
- *CCSDS File Delivery Protocol* (CFDP)[AA20], e.g. used partially (*CFDP-Lite*[Mes]) by NASA’s MESSENGER to transmit data to Earth

All these protocols were designed exclusively for interruptible paths and are not suitable as a common protocol for the entire mission network. Nonetheless, the idea of *store-and-forward mechanism* and *hop acknowledgement* was picked for further considerations regarding the design of a uniform network-wide Compass protocol.

To tackle the problem of bit errors, *error correcting code* (ECC) can be applied during the transmission, more specifically the *forward error correction*. For this purpose the *Viterbi decoder* or *Polar Code* can be applied. The limiting factor of these correction techniques is, that they are usually implemented in the radio hardware. Especially correction algorithms that rely on the probability of a current baud being a 0 or 1 (*soft bits*), are dependent on the information from Layer 1 (physical layer). The Nanocom AX100 from Gomspace, which has been selected for the NetSat satellites, supports Viterbi algorithm out-of-the-box.

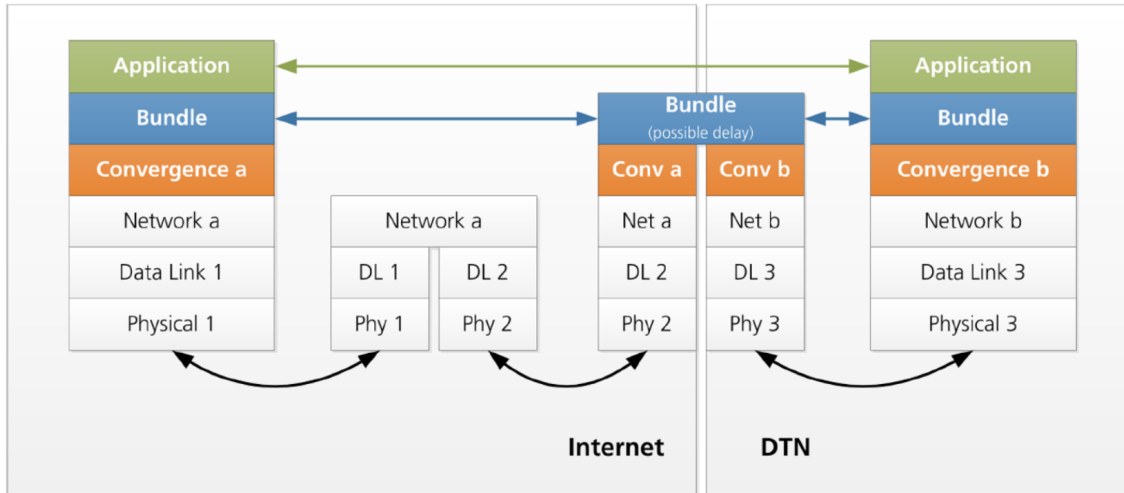


Figure 2.13: Convergence Protocol

## 2.6.5 Inter Satellite Link

During the development phase of NetSat mission, the decision was made to use the same communication protocols between the satellites (ISL) and between the satellites and ground relays. Thus, satellite transceivers that are used for ground communication, can also be used for ISL. In addition, the satellites can access ground systems and other satellites in the same way.

Having multiple satellites in orbit raises the problem of concurrent access of satellites to the same radio channel. The issue of concurrent access can be solved using several approaches (see figure 2.14):

- *Time Division Multiple Access (TDMA)*: different time slots for different data streams
- *Frequency Division Multiple Access (FDMA)*: different frequencies
- *Code Division Multiple Access (CDMA)*: simultaneous transmission over a single communication channel using special coding schemes for every channel/node
- *Space Division Multiple Access (SDMA)*: use beam forming or beam orientation change to spatially separate channels
- *Carrier Sense Multiple Access (CSMA)*: verifies the absence of other traffic before transmitting

All these techniques can only be applied in the OSI layer 1 (physical layer) and are therefore in general handled by the radio hardware itself, thus if one of the option is desired, appropriate transceiver hardware must be selected and integrated into the satellite. Additionally, a compatible solution for the ground segment must be elaborated if the same channel type is used for the ground-space communication.

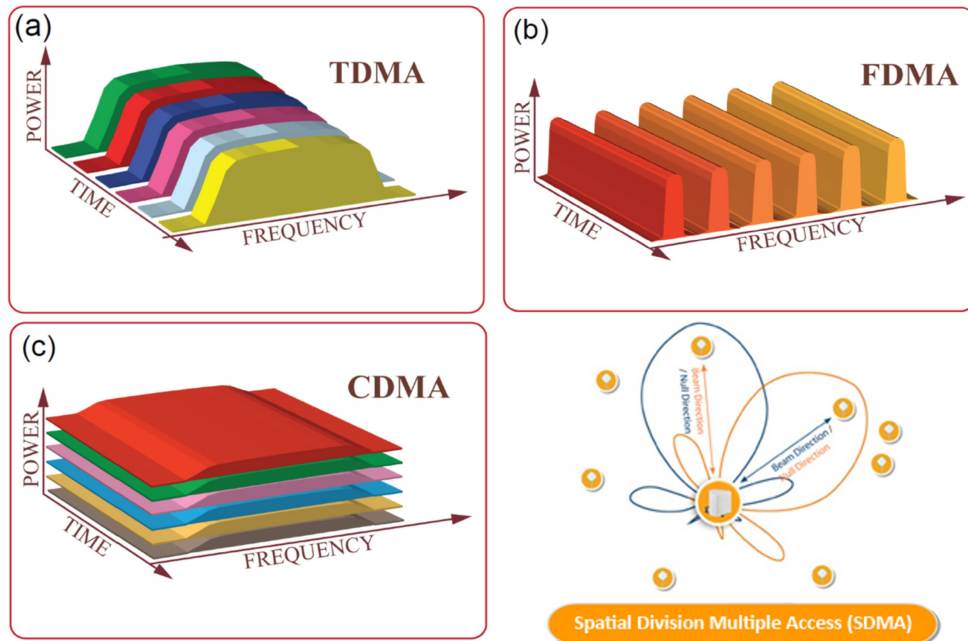


Figure 2.14: Multiple Access techniques (image sources: nature.com and max-iustech.com)

## 2.7 Considerations on Satellite Development and Operations

In this section several non-communication related challenges and possible solutions are discussed. In the scope of this work, the ground segment consists not only of essential nodes, such as ground stations, mission servers and operation workstations (or mission control center), but also of auxiliary nodes: simulations, test facilities and satellite engineering models.

### 2.7.1 Satellite Software Development

In general, several software developers are working on different satellite subsystems at the same time. In this development phase the subsystems are in different network environments (from the point of view of the subsystem) compared to the environment in the final flight state. That is, in practice during the development, all subsystems are located at different places and cannot interact with each other as they should do later in the final assembly. As a consequence, the developers need to periodically get together, insert all relevant satellite subsystems into one *flat-sat* or the engineering model and test the cooperative processes. It is highly beneficial to enable the communication between distantly located subsystems during the development stage, as it improves the overall quality of the development due to the constant testability of the cooperative processes. A solution is the proposed *distributed sat* approach as an extension of the flat-sat approach, i.e. a *dynamic mission network* that is established in the early development phase and remains active

throughout the whole mission.

## Dynamic Code Execution

To enable more sophisticated and customizable mission goals achievements, every nanosatellite in a formation must offer an interface for *dynamic code execution* as a basis for constraint-based on-board scheduler. Depending on the mission, the scheduler tasks can either be uplinked from ground or be generated in-orbit. Especially the latter use is important for autonomous in-orbit formation control, which dynamically calculates thrust maneuvers depending on the selected target orbit and currently measured orbit parameters. The execution must be therefore performed on a *continuously running* satellite subsystem. Since this thesis covers pico- and nanosatellite formation missions, the dynamic code execution capability must be available on very low-power subsystems, as those are usually the only *continuously* running components – such as the NetSat’s OBC, which is powered by a 16 bit microcontroller with only 16kb RAM.

## Common Code Base

The hardware of every satellite subsystem is designed in such a way, as to fulfil its particular tasks and at the same time to minimize space and power consumption requirements as well as development cost. This usually results in subsystems with different microcontrollers. Since all subsystems offer the same basic functionalities, such as common protocols and services, it is highly advantageous to reuse the implementation of those libraries in all subsystem software images. This way, a corrected software bug or new features in a library will be automatically applied to all other subsystem software implementations. To achieve this, the software must be designed with appropriate abstraction layers, i.e. to make the entire code hardware-independent (except the code in the hardware abstraction layer itself). Further advantage of the abstraction is the ability to compile and test subsystem code on workstations, which in turn enables straightforward design of SIL, PIL and HIL tests.

## Embedded Operating Systems

Depending on the real-time requirements, the projected software complexity and the available hardware resources, the satellite software can be implemented on top of an existing embedded operating system. Some of the prominent examples are:

- *VxWorks* used, for example, on board the NASA’s *Mars Pathfinder*, *Spirit*, *Opportunity* and *Curiosity* rovers[Mui96].
- *FreeRTOS* open-source operating system, whose core was for example used in StudSat-2 nanosatellite [Lam+15].
- *SmartOS* was developed for distributed (ground) applications by Marcel Baunach [Bau12].
- *Rodos* – successor of *BOSS* operating system, developed by DLR and University of Würzburg and was flown on TET-1, BIROS, BeeSat 1 and BeeSat 2.
- *μCLinux* is a stripped down Linux system, which was flown on UWE-1 and UWE-2 nanosatellites

Unfortunately most of the embedded OS either do not support 16 bit MCUs (VxWorks, SmartOS, Rodos) or have comparable high memory requirements ( $\mu$ CLinux) and are therefore not suitable for all subsystems in the UWE-3, UWE-4 and NetSat satellites. One exception to that is FreeRTOS, which will be considered for the follow-up missions at Zentrum für Telematik. The "downgrade" from the Linux-based UWE-2 to the non-Linux missions UWE-3, UWE-4 and NetSat is due to the decision to simplify the on-board data handling subsystem: minimize potential area for single event upsets (no external RAM) and to optimize the on-board software back to *bare-metal*. In retrospect this step back considerably improved the in-orbit performance of both non-Linux UWEs in comparison to their predecessors. Nonetheless, a full-fledged Linux operating system is now used as a base of the NetSat's Computing Board, which is activated on-demand.

Despite their advantages, real-time operating systems also introduce a new complexity layer: process synchronisation, prioritization, proper resource management and mutual exclusions. Furthermore, the available main memory must be split and assigned to all available processes, thus limiting available memory resources even more. Another option is to implement common middleware in OS-agnostic way, i.e. make the middleware runnable on bare-metal or on top of some operating system. This approach was selected for the implementation of the embedded Compass middleware.

## 2.7.2 Testing and Verification

Some of the satellite subsystems cannot be fully tested using just software (e.g. unit-tests). Instead, some of the components need to be tested in an environment that is comparable to the environment expected in-orbit: launch conditions, temperature and radiation. Furthermore, many sensors and actuators need to be tested and calibrated in the appropriate test facilities. For example, to calibrate a sun sensor of a satellite panel, the panel can be mounted in a motion simulator and oriented towards a light source at different angles. This task is accomplished with at least three components: satellite panel, motion simulator and a calibration conductor. The staff involved in the calibration procedure has two options:

1. Manually perform the calibration, i.e. periodically control the motion simulator, read-out the sun-sensor values and note them down along with the currently set orientation.
2. Write a calibration algorithm (e.g. in Matlab) and *glue-code* that enables the algorithm to access both the motion simulator and the satellite panel.

In the first case, the calibration conductor is a person and in the second one some developed algorithm. Due to the non-uniform access to the test facilities and the satellite subsystem, the glue code is mainly used to translate between the desired action and the specific protocol. Now, the same test facility can also be used to calibrate other sensors, such as inertial measurement units. But it would either require at least two specialists for manual execution or another glue-code designed to perform the test automatically. The same issue occurs in any other testing or verification scenario with non-compatible test equipment.

For a single satellite mission, the manual approach may be an adequate solution. For the development of many satellites in a comparable short time period, the automatic testing solution is the preferable scenario, as it reduces the complexity, the required time and the amount of required personnel.

Before the eventual launch, the satellites (flight models or equivalent engineering models) must undergo verification tests at external test facilities, such as shaker tests or temperature-controlled vacuum chambers. These procedures are in general very cost intensive and limited in time, e.g. contract over 24 hours. The ability to remotely access the satellites "from home" may considerably improve the outcome of the test procedures, as it allows non-attendant co-workers to offer remote support – especially in case of undesired anomalies. This has been shown multiple times during the vacuum chamber tests of UWE-4 and NetSat in the iABG test facilities in Munich. The satellites were connected to the ZfT's Compass mission network via mobile connections during the test procedures.

### 2.7.3 Distributed Dynamic Mission Network

All communication paths that are present in a satellite mission, can be combined to a *decentralized virtual mission network*. That is, some of the systems are only accessible via other systems, which is for example the case when the ground station can only access satellite B via the satellite A. Ideally, all systems in a mission network should be able to communicate with each other in a uniform way. In this context a system is, for example, a ground station (ground relay), an operator's workstation, a motion simulator, a Matlab simulation and all particular satellite subsystems. This requirement implies a uniform and unique addressability of all systems within the network, i.e. *every* system must have a common protocol baseline that allows them to communicate with each other. The desired protocol must be optimally usable in all mission segments – ground-ground, ground-space, space-space and between the satellite's subsystems (intra-satellite communication). The protocol must offer enough capabilities to handle inhomogeneous paths (routing, DTN etc.) and at the same time minimize the overhead on paths where these functionalities are not required.

The dynamic requirement implies that every system can access the network at any time and gain knowledge about the entire network without a priori knowledge – comparable to the functionality of the Internet. This is an alternative concept to statically structured networks, where every node is equipped with a static routing map that needs to be modified manually every time the structure of the mission network is changed. In addition, the dynamic nature of the network allows nodes to enter the network at different places (e.g. satellites passing over different ground relays) or yet unknown nodes to enter the network (e.g. new operator's workstation).

### 2.7.4 Operations

The operations phase begins after one or multiple satellites were launched into orbit, so the *Launch and Early Orbit phase* (LEOP) is the starting point. The operations can be divided into two parts that can theoretically be performed by two independent teams (satellite bus and satellite payload provider):

- Basic satellite operations (carrier): TT&C (Telemetry, Tracking, and Command) – keep track of the entire system healthiness
- Payload or mission operations: perform mission specific tasks being enabled by the payload (e.g. Earth imaging)

The operations software can either be some in-house solution or can be based on existing frameworks, such as ESA’s *Spacecraft Control and Operating System* (SCOS 2000) or more generic ESA’s *Ground Operation Software Systems* (EGOS) [Pec05], whereat both are limited to the CCSDS frame standard, thus creating a constraint on the protocol used between the operations workstation (or mission control center) and the in-orbit satellites. There also exist solutions from the industry, such as the *Pleniter Modular Control Center Software Suite* from SCISYS, which, for example, will be used to control the satellites of the *OneWeb* mission. However, the Pleniter suite is expensive, is aimed at larger missions and is not publicly available. In addition, the frameworks can be an overkill for smaller nanosatellite teams.

The screenshot shows a window titled "TC History: RTE" with a "TC History Report" tab. The window displays a table of TC records. The table has the following columns: Name, Description, Sequence, Domain, Release Time, Execution Time, S, D, C, G, B, I, L, S, T, Source, FC, TC, R, G, O, A, SS, 1122, CC. The records are filtered by Domain: RTE. The table shows a list of TC records with their respective release and execution times and status flags. Below the table, there are filter controls for Domain (RTE), Type, Sub-Type, APID, Mnemonic, Sequence, Ack, Workstation (valmcs3), Local WS, and Manual Stacks. There are also "Apply" and "Default" buttons.

Figure 2.15: SCOS 2000 TC History Display (image sources: esa.int)

## 2.8 Roundup

All discussed solutions to the common satellite mission challenges were used as the input and inspiration to fulfil the main goal of this thesis: unify the functional and communication interfaces of all ground and space systems. Since the modus operandi of this thesis was to proof the feasibility of all proposed approaches in a real-life CubeSat mission, the



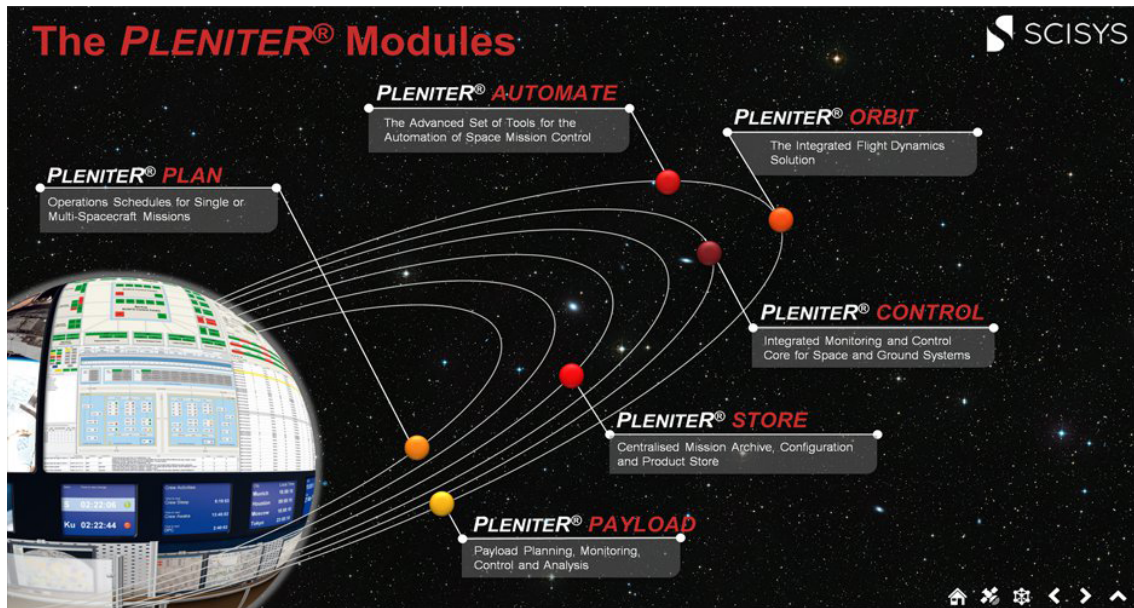


Figure 2.16: Pleniter Software Suite (image sources: pleniter.com)

realization had to be performed during the active satellite development at the University of Würzburg, Chair VII, and ZfT. Therefore, additional practical limitations were present:

1. very limited manpower, e.g. 2-3 full-time employees for the whole UWE-4 mission
2. since the hardware of the UWE-3 subsystems was proven in-orbit, most of it had to be reused in the UWE-4 and NetSat missions
3. *gradually* transformation of the UWE-4 mission software baseline (initially cloned from UWE-3) towards the new system

The *first* limitation is the most crucial one, as the small team had to perform all mission-relevant tasks: develop satellite hard- and software, maintain the UHF ground station, implement software for all ground systems (Operations front-end, ground station, mission server, protocols and services), scientific work, tests and verification, launch organization and eventually satellite operations.

The *second* limitation dictated the *software-follows-hardware* paradigm, i.e. the common software and protocol solution for all satellite subsystems had to be suitable for the smallest microcontroller (16 bit, 4 kB RAM, 32 kB ROM).

Due to the *third* limitation the new software framework for ground and space systems could not first be fully implemented and then used to replace the existing infrastructure at one go. Instead, the realization of the proposed approach had to be orchestrated in such a way that:

- the recently developed component is capable of running in the currently active software and/or protocol environment
- the time range is maximized, during which the most crucial parts of the new development are in-use (real-life tests)
- parts of the embedded software are finished on a par with the corresponding hardware development (software driven hardware testing)



## 3 | Approach

The perimeter of this thesis has been deliberately chosen to embrace as many relevant areas as possible: ground and space segment, communication, operations, software interfaces, testing, experiment execution and cooperative machine behaviour. Exclusive focus on a particular area, without taking all other relevant areas into account, will lead to results that very likely may become unsuitable at some point during the mission.

This thesis was started during the early LEOP phase of the UWE-3 mission in January 2014. This work was initially motivated by the very poor outcome of the UWE-3 operations: extremely low communication throughput, no auto-operations capabilities, inflexible maintainability of the ground stations, and the inability of the satellite to execute dynamic software (new experiments required hazardous in-orbit software updates). The realization of the thesis' approaches was performed in such a way that not only the follow-up missions will be improved but also as many new implemented components as possible are used to augment the currently active UWE-3 mission.

The protocol configuration of the UWE-3 mission, which was technically the starting point of this work, is depicted in figure 3.1. It shows four different system types:

- *UWE-X Operations* front-end
- *Ground Station Server* (ground relay) based on several (open-source) software components
- *OBC* as the only satellite subsystem with Missionlink support
- *ADCS, Panels* subsystems that could only be accessed using specific protocols during the development phase

Due to the nature of the UWE-3's Missionlink protocol (1-to-1 network only), the protocol configuration was only allowing two peers: *space node* (represented by the OBDH subsystem) and *ground node* (all operation front-end instances and the ground server acted as one node). Further auxiliary systems, such as Matlab instances, for data post-processing or software components used for conducting calibrations of magnetorquers, sun-sensors and acceleration sensors were not part of this scheme. In the UWE-3 mission, no common protocol was used during both the development and in-orbit phase. As the functionality of the ADCS and the panels could not be accessed directly (e.g. using forwarding on the OBC), the access had to be implemented in OBC for every single function (*facade methods*). That is, if one of these subsystems had to offer some new routines to the operator, the OBC had to be extended.

During the development and the AIT/AIV phase of the UWE-3 mission the team had access to several non-inter-compatible systems. That is, different software artifacts and protocols were used to fulfil specific tasks: testing, communication with engineering models, operation of flight models etc. With the raising requirements in the following missions,

such as the *UWE-4* (cooperative implementation with TU Dresden and yet multiple active satellites in space), *NetSat* (cooperating formation of four in-house nanosatellites), *QUBE* (cooperative development with partners from *LMU*, *DLR* and *Max-Planck Institute*) and *TOM* (cooperative satellites from different partners) missions and with the limitations in mind, such as the available manpower, time-frame and financial resources, it became evident that a new approach for the entire mission design and development must be elaborated.

Due to the limitations described in 2.8, the realization order of this thesis was:

1. Design unification approach for space or ground system functionalities (MTBA). Motivation: theoretical basis for all new software module implementations.
2. Re-implement *UHF Ground station software*: propagator, hardware low-level control (rotor, transmitter, modem). Motivation: improve the flexibility and prepare for auto-operations component.
3. Implement first version of the *auto-operator* (not mentioned in this thesis, as it has been superseded by a new version later). Motivation: improve the outcome of the UWE-3 mission and enable new experiments.
4. Design *Tiny scripting* language, implement it as a service and perform in-orbit UWE-3 software update. Motivation: dynamic attitude determination and control experiments, in-orbit frequency noise measurements to find better UHF base frequency
5. Design *Compass protocol* and *Compass services*. Motivation: specification of the communication unification between all systems and current UWE-3 and future (UWE-4, NetSat) tasks.
6. Implement Compass middleware for ground nodes (*CompassNode*). Create *Compass-Missionlink bridge* to support the UWE-3 satellite. Motivation: equip with new software Matlab nodes, Compass Operation front-end, ground station, mission server and Compass gateways
7. Implement Compass middleware for embedded nodes (*Compass OS*) as a final transformation step. Motivation: unify ground and space protocols, unify access to all satellite subsystems (UWE-4, NetSat) and enable ISL.
8. Include Compass middleware in further auxiliary systems: Matlab instances, Orekit simulations, test equipment (motion simulators, sun simulators) and satellite development kits. Motivation: integrate auxiliary systems in the Compass ecosystem.
9. Use the completed Compass ecosystem for the UWE-4 and NetSat development, verification, software testing and simulation of LEOP operations.
10. Combine all space (UWE-3, UWE-4, NetSat) and all ground systems to *one* supermission Compass network. Motivation: (auto-)operate all satellites at the same time.
11. Implementation of the final *auto-operations* technology with recording capabilities.
12. Throughout the thesis: In-orbit experience with the UWE-3, UWE-4 and NetSat (LEOP) missions.

The *Compass Operations* front-end was continuously developed during all these steps.

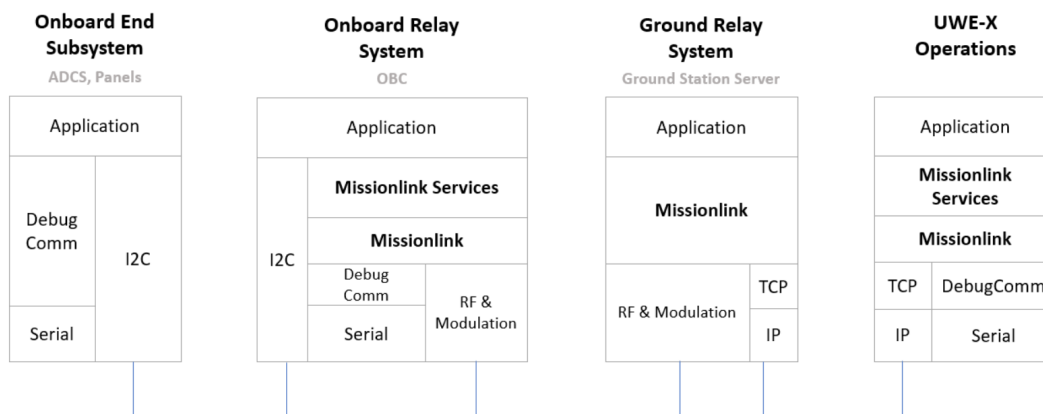


Figure 3.1: Protocol configuration of the UWE-3 mission (before this thesis)

### 3.1 Uniform Model Interface

This section proposes an approach for the unification of the functionality of *distributed* hard- and software components. First, the theoretical background without implementation details is addressed, followed by particular implementations shown in the following chapters.

What is functionality? The definition in the Oxford dictionary is: “The range of operations that can be run on a computer or other electronic system”. This work is focused *only on software-based functionalities*, i.e. any capability that can be accessed or enabled by a microcontroller or a computer. The *high-level* functionality (e.g. satellite tracking) is mostly based on *low-level* functions (e.g. propagation, antenna movement etc.), whereat both levels should be individually accessible. A discrete access to different levels is highly beneficial, as it allows to implement new high-level capabilities based on already available low-level ones.

For the sake of convenience, an instance that offers a range of functional capabilities is called a *Node* and the totality of *all* nodes relevant to a mission is called a *Mission Network*. In most cases a node is a microcontroller or CPU based hardware component. For example, a satellite *subsystem* with computational capabilities is considered a node. Furthermore, multiple nodes can be logically grouped to a *System*, e.g. a satellite is a system based on multiple subsystem nodes. A node must be discretely addressable within the mission network – similar to a workstation being accessible in the intranet or Internet via its IP address.

#### Example:

A simplified example of the NetSat formation mission network with:

- four in-orbit nanosatellites, each based on multiple different subsystem nodes: OBC, AOCS, Panels and Thruster control.
- one satellite engineering model for test purposes
- multiple development models, which are used during the development phase
- developer’s and test engineer’s workstations

- operator's workstations
- in-house servers: ground station server, mission server
- external ground stations
- test facilities, e.g. turntables, sun simulator
- simulation environments: Matlab, Orekit

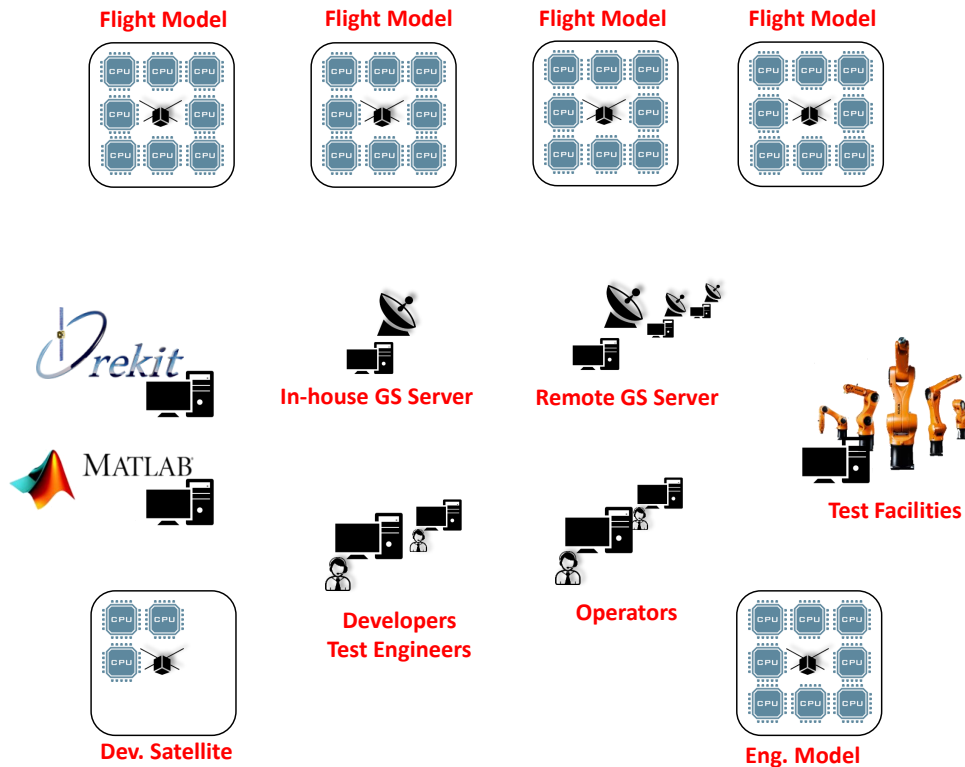


Figure 3.2: Simplified NetSat formation Mission Network

Consider the example above. All of the nodes have obviously very exclusive duties, which are commonly accessed via very different interfaces, thus forming adjacent *interface islands*. As a consequence, many different – mostly *non inter-compatible* tools – must be used to access different nodes, which in turn leads to either unnecessary high man-power demand for simple tasks (many experts are required to enable a cooperative task) or unnecessary high training overhead for one-time procedures – see example below. Usually the *glue code* dominates the development process, i.e. software development that is necessary to combine different interfaces and protocols to fulfil some specific task.

**Example:**

A test engineer needs to calibrate the sun sensor of a satellite flight model. During the test procedure following components must be operated:

- satellite panel subsystem
- sensor model in the simulation environment (Matlab)
- test facilities: sun simulator and motion simulator

Three to four experts must be involved in the process: subsystem developer, simulation expert, test engineer and communication expert. For this specific test procedure specific software must be prepared by multiple responsible person in order to provide access to different interfaces and to ensure a proper test *life-cycle* (timing, error recognition etc.).

This concept is obviously not feasible for smaller teams (cf. 2-4 full-time co-workers in the UWE and NetSat missions) and is not suitable for *agile* development. Instead every team-member should be able to access as much functionality as possible in the entire mission network with as less as possible effort. To face this challenge, a *Model Tree Based Architecture* (MTBA) approach has been developed, which considerably decreased the overall complexity of the mission development and improved the development quality and *richness* by allowing all team members to focus on their particular tasks but still be able to uniformly access the functionality implemented by other members.

### 3.1.1 Model Tree Based Architecture

The *Model Tree Based Architecture* (MTBA) is inspired by similar, but more limited approaches, such as the *Model-based design* (MBD) or the *E4 Application Model* of the *Eclipse Development Platform*. Model-based design is widely used to develop complex controls or visually design signal processing chains. The E4 Application Model is used to describe the junction of multiple plugins or visual components to form a RCP-based E4 application. However, all of the studied approaches are designed to fulfil some very specific task, such as signal processing or RCP application. In addition, all of these techniques are bundled with further components – such as Eclipse or Simulink – that are not usable on all present nodes. Therefore, the design of a new software unification approach was decided that can cover all nodes with as less as possible prerequisites.

The basic idea of the MTBA is that any software component can be represented by its *Model Tree*. The model tree consists of hierarchically ordered branches, with each branch containing functions, states, variables, and other sub-branches. Multiple software components or libraries can now be combined to form a larger model tree, such that the entire *software landscape* of a satellite mission can be viewed as a *distributed mission model tree* [DS14]. Instead of using specific software interfaces (e.g. via file interface or sockets) or protocols, any component can be accessed or monitored via its model. For example, to set some specific UHF antenna orientation, the angle values are set in the corresponding model location (see Figure 3.4 under GS/Server/Rotator path). This way the vast majority of the inter-system communication and system monitoring can be reduced to the task of a modification of a (remote) model and propagation of model changes – therefore dramatically abating the amount of the required *glue code*.

In order to make an existing library or a software solution MTBA-able, the developer needs to implement a *Model Tree Bridge*. This portion of code is required to reflect the software states, parameters and functions in the model tree and vice versa. It is up to the developer to decide the level of the abstraction, i.e. which states and functions should remain invisible to the network and how the model tree is organized.

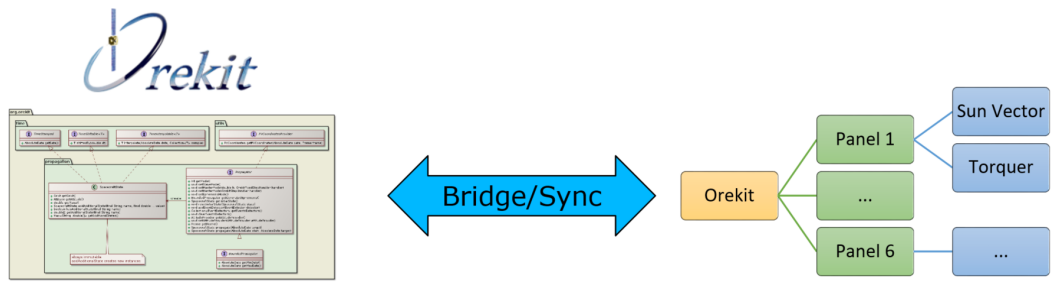


Figure 3.3: MTBA Bridge

**Example:**

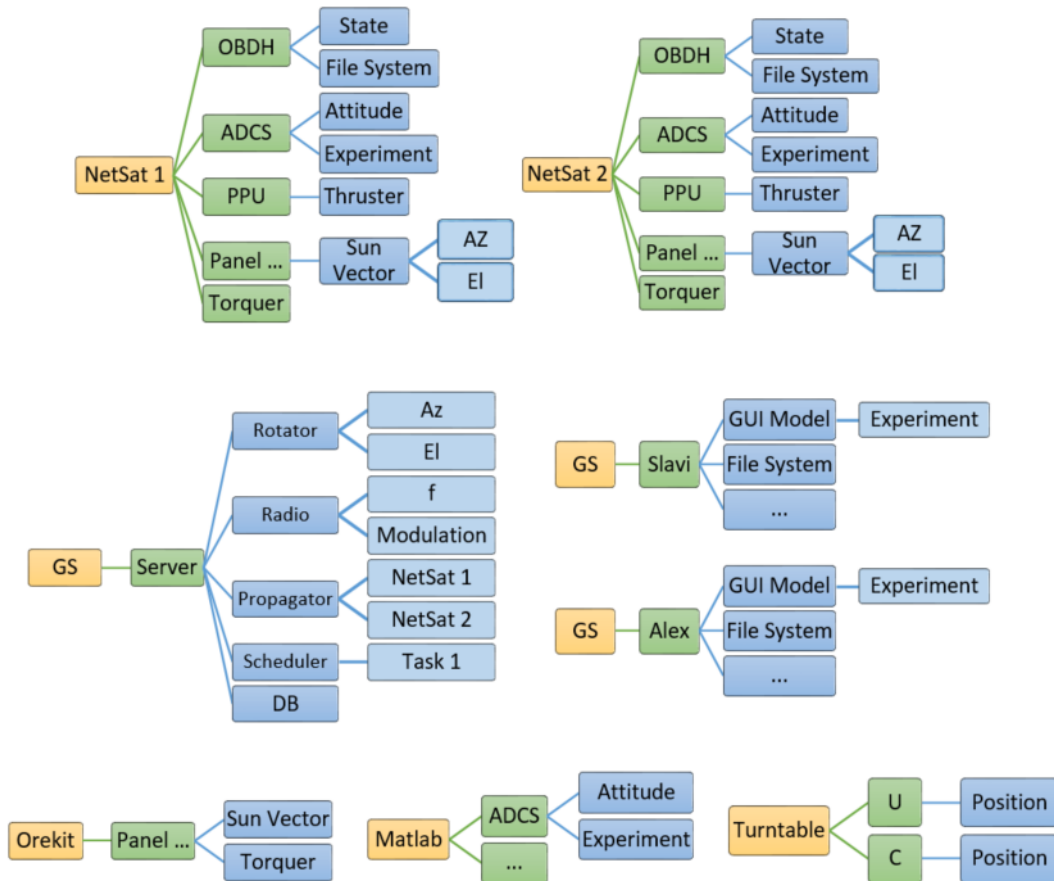


Figure 3.4: Simplified view of the NetSat mission MTBA model

A simplified distributed model tree of the NetSat nodes is shown in figure 3.4. Two satellite models are depicted in the upper part of the figure. Each of the satellite branches contains the model of all available subsystem nodes, where every node exposes

its subsystem-specific parameters, states and functions. The Orekit simulation system provides 6 virtual panel nodes, which can be used to substitute the actual panel hardware and thus enabling Software-in-the-Loop testing.

### 3.1.2 Model Tree shadowing and Model-based communication

In the previous section a concept of a *distributed mission model tree* has been shown. From the logical point of view, this tree is always present and describes the composition of the entire software landscape. However, from the technical point of view the presence or the availability of single branches may vary with time, e.g. when a satellite is out of range. Therefore it is necessary to introduce the concept of *model knowledge*. In practice a single node does not require the knowledge of the entire mission network – instead only a subset of the remote model tree is required to realize cooperative processes.

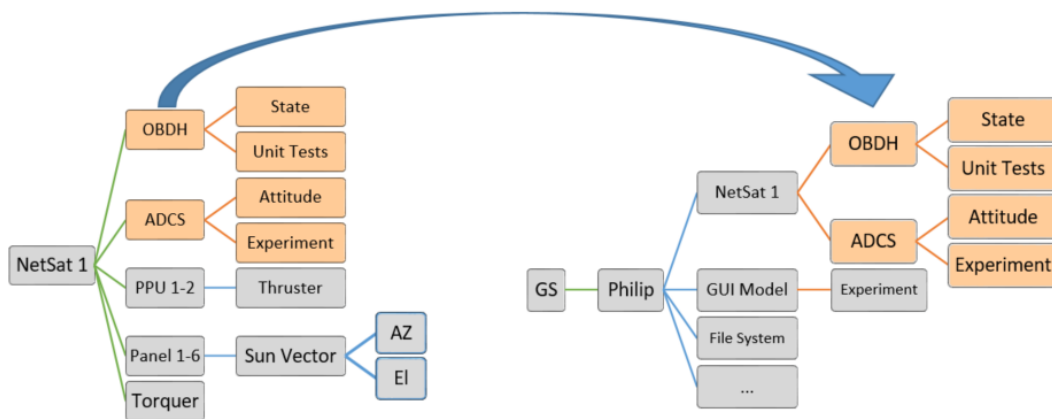


Figure 3.5: Model Shadowing

For example, an operator’s workstation node can hold its last knowledge about the model of a *NetSat* satellite (see figure 3.5). Any change in the local *model copy* leads to its synchronization with the *real* model. With this *Model-based communication* technique and supposed that the mission model is properly designed, most of the cooperative tasks and tasks that rely on multiple nodes can be implemented in a communication agnostic way, i.e. without taking care of the communication process.

### 3.1.3 Model buffering

Another advantage of the Model shadowing is the ability to *buffer model knowledge* on high-performance nodes, such as mission server nodes, to minimize communication throughput to the model’s origin. Since every model change event is implicitly transferred as a communication packet between the model’s origin and its listener, it most probably passes some high-performance intermediate node, such as a mission or ground station

server, where it can be cached for later use. Every model entry holds not only the value itself but also a timestamp at which the value was created in its origin. If some model *read request* contains a *maximum age* constraint and on its way to the target the packet passes a node that possesses a recent enough value of the desired model entry, the request can be satisfied without reaching the target. This dramatically reduces the required communication throughput induced by multiple operator's workstations, as the request of one operator can indirectly fulfil the future requests of other operators.

### 3.1.4 Model swapping

In the context of the mission model tree, every node-specific model branch must contain some sort of information about its origin, such as address, name or any other addressing information supported by the used communication interface. Every time a local model knowledge is synchronized with its origin, a request or update packets are created with the origin's or listener's address. Now every physical (i.e. "real") node can be represented in the mission network by some hardware or software simulated one. This is of particular interest for in-the-loop tests, where real hardware nodes can be substituted with hardware and software simulations. This is described in more detail in the section 3.1.6.

### 3.1.5 Implementation

A MTBA-able software component can be included at any time in the entire mission software model without additional software or glue code. During this thesis *Compass middleware* was developed, which provides Model interfaces, and Compass Model service for model requests, modification and propagation on the upper Compass service layers. It is available for Java (*CompassNode*) and C/C++ (*Compass OS*) and can be deployed both to high-end workstations as well as to ultra-low power microcontrollers (min. requirement 16 bit MCU, 500B RAM, 10+ KB ROM).

By the end of this thesis, all relevant nodes for current and planned missions at the University Würzburg (Chair VII, Robotics and Telematics) and ZfT were made accessible via the Model service:

- Compass Operations front-end
- Embedded software for all satellite subsystems
- Mission Server
- GS Server
- Matlab environment
- Distributed simulation environment (Orekit-driven)
- Test facilities

The proposed Model approach does not dictate *how* to implement a functionality – it rather describes how the functionality should be seen and accessed from the outside. The Model representation can either be achieved by creating a bridge (see section 3.1.1) using the now available middleware or by adding the support at the protocol level (see section 4.3.10). Further implementation details are described in more detail in the Chapter 5 and Chapter 6.



### 3.1.6 Testability

As mentioned in the section 3.1.4, model-swapping forms a base for *in-the-loop* testing and is therefore a suitable tool for *Model-driven engineering* (MDE). Model-driven development is extensively used in the automotive industry [Wae16]. The basic idea of the Model-driven engineering is to thoroughly test a newly designed component in several development stages (Model  $\rightarrow$  Software  $\rightarrow$  Processor  $\rightarrow$  Hardware) against a *plant simulation*, which represents all related dynamic systems of the counterpart. During the test, the behaviour and interaction of the implementation is tested against the simulation of the future environment. On success, the development stage is shifted to the next level. There exist four different stages (see figure 3.6):

- *Model-in-the-Loop*. Test the model (e.g. Simulink) of the component.
- *Software-in-the-Loop*. Test the software of the component running on a development Workstation.
- *Processor-in-the-Loop*. Test the software running on a platform that is compatible to the final one (e.g. on a Development Kit).
- *Hardware-in-the-Loop*. Test the final component against the plant.

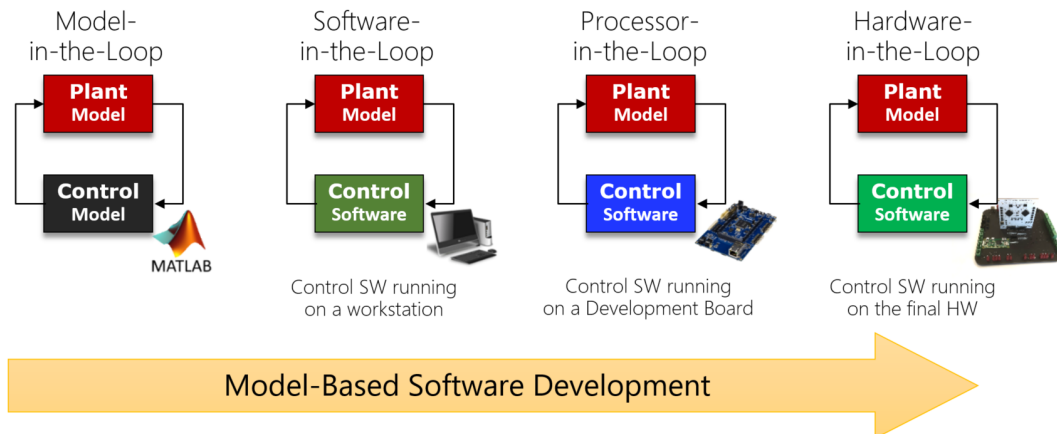


Figure 3.6: Model Based Development

In the MTBA context, the plant is a network being populated with simulated nodes. The functionality of a new node is first implemented as a Model (e.g. in Matlab), then implemented in a software framework compatible with the final node hardware (e.g. GCC), followed by the software being executed on the compatible and later on final hardware platform.

However, the Model-driven engineering concept is limited to a plant *simulation* being the counterpart of the test procedure. Thus, a more generalized *Mixed-Loop-Testing* concept is proposed as described in the following section.

## Mixed-Loop-Testing

As stated before, the MDE testing requires a simulation of the entire plant. In the nanosatellite context, all satellite subsystems, all environmental inputs and dynamics must be modelled and simulated in a proper environment. This implies *tremendous* efforts for small teams, as not only the functionality but also the response behaviour of all relevant nodes must be simulated, which in-turn requires to some extent the simulation of the microcontroller behaviour (frequency) and the behaviour of (error-prone) links, such as I2C, serial, radio etc. Moreover, many satellite subsystems or major hardware components (transmitters, propulsion system, etc.) are purchased from other companies.

Usually, the teams create models for very specific areas that rather belong in the Model-based design category. The MDE testing concept is a great opportunity to improve the development organization and has the potential to significantly improve the quality of the developed components. With some modifications it becomes a useful tool in realistic development conditions.

Mixed-Loop-Testing (MLT) proposes a generalization of the MDE testing concept, by introducing a mode of the plant itself. Now also the plant nodes (network) can be in the model, software, processor or hardware stage. The abbreviations MIL, SIL, PIL and HIL are now extended by an additional plant stage letter: AIBL with A is the stage of the test component and B is the stage of the plant. Using the proposed terminology, a test can be described more precisely – see examples below. The MIL, SIL, PIL and HIL tests become special cases of the Mixed-Loop-tests.

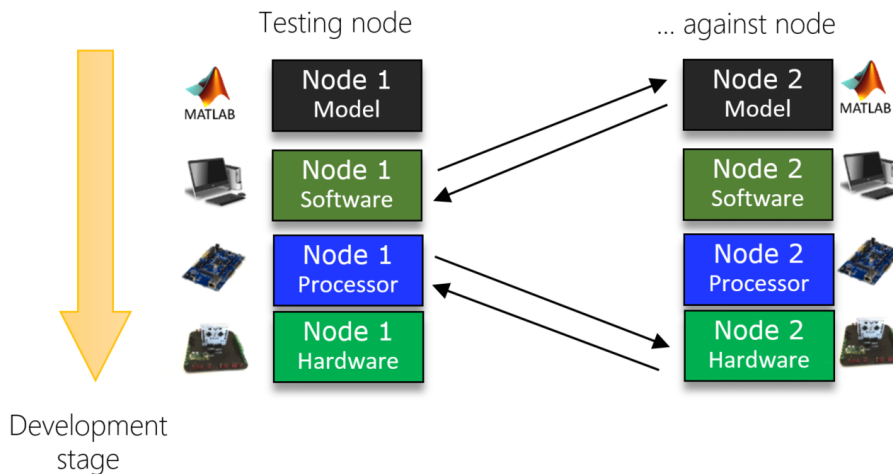


Figure 3.7: Development stages

### Example: Mixed-Loop-Tests

- Software-in-the-Software-Loop (*SISL*): run ADCS and OBC (plant) software on separate workstations
- Software-in-the-Processor-Loop (*SIPL*): same but with OBC software executed on a hardware development kit

- Hardware-in-the-Model-Loop (*HIML*): feed ADCS subsystem with values from an Orekit instance
- Model-in-the-Hardware-Loop (*MIHL*): run sun sensor calibration algorithm in Matlab with plant consisting of a flight model in the turntable.

From experience with the in-house and external (TU Munich, TU Dresden, DLR, LMU) nanosatellite projects, a subsystem implementation begins mostly in a software development environment that is provided by the microcontroller’s manufacturer – such as *Code Composer Studio* for Texas Instruments MCUs or *Atmel Studio* for Atmel MCUs. One exception to that is the implementation of Linux-based subsystems – here usually a standard *GCC compiler* or *higher* (script) languages are involved. The testing procedures of the cooperative behaviour are mostly executed using either hardware development kits or the final satellite hardware (e.g. *PIPL* or *HIHL*).

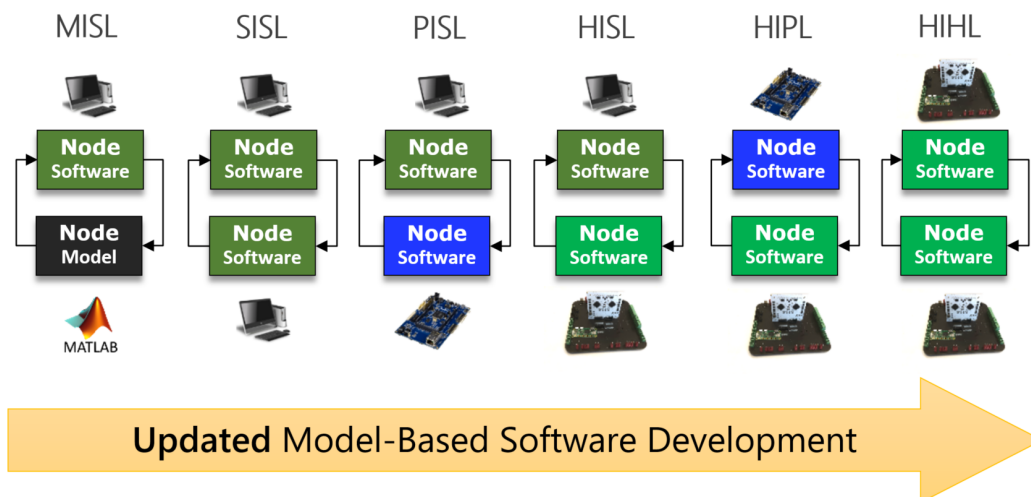


Figure 3.8: Mixed-Loop-Testing

In order to enable the software stage of the plant, the software of all relevant plant nodes must be runnable on a workstation (see figure 3.7). The hereby achieved testing environment is not identical to the real hardware with respect to the timing behaviour – but considering the comparable low effort, such a system gives a great opportunity to perform cooperative functional tests without additional hardware. One of the tasks during the thesis was therefore to convert the software implementation of all satellite subsystems – see Chapter 5. In the consequence, many formation-relevant in-the-loop tests could be performed, which is described in more detail in Chapter 7.

## 3.2 Uniform Communication

In the previous section a method was proposed that was applied in the current Cube-Sat missions at the ZfT and University of Würzburg to uniformly describe and release mission-relevant functionality of all components involved in the mission network. After the *functionality interface* is settled down, a problem arises of *how* to remotely access

the desired functions. In real-world missions, many hardware and software components are connected to the mission network via *heterogeneous* communication links with respect to the utilized protocols (e.g. TCP/IP via LAN or WAN, local Sockets, Serial interface, SPI, Two-line Interface (I2C) or Radio) and physical links (wire, wireless LAN or UHF communication). A simplified example of the network composition of the NetSat mission can be seen in the figure 3.9. As can be seen in the example, numerous *protocol islands* are existent in such a network. Thus special protocols/software must be utilized to access some specific component. The communication gets even more complex if it must be established along multiple islands – for example if a Matlab script needs to access satellite’s values to perform a calibration task.

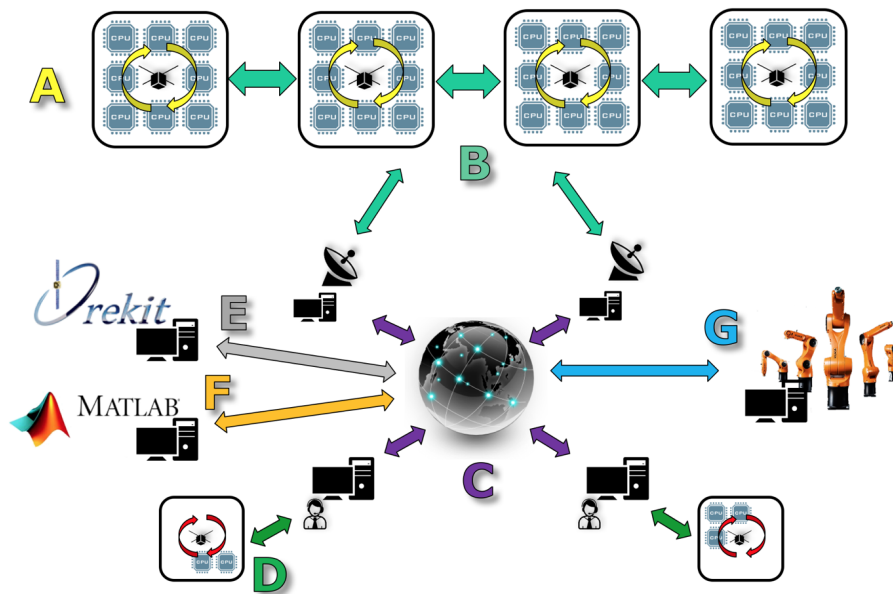


Figure 3.9: Protocols in a simplified NetSat mission network. (A) embedded intra-system protocol, (B) wireless inter satellite protocol, (C) protocol between workstations, (D) engineering link protocol, (E) orbit simulation handling protocol, (F) Matlab simulation protocol, (G) test facility control

A common way to face the problem of the communication inhomogeneity is to implement a protocol tunnel, which implies that every component between two end-points must be able to detect *some* incoming packets as tunnel communication and forward them as specified by the configuration. Another option is to add a *common protocol* on top of the protocol stack of all involved components. Such protocol must *at least* fulfil the requirements.

1. Runability on low-performance components (satellite subsystems), max. hardware requirement in the scope of this thesis is: 16 bit architecture with 32 KB flash and 4 kB RAM.

2. Provide global addressing and built-in routing capabilities on all components.
3. Usability on *asymmetric* lines, i.e. where the transmission quality (drastically) differs from the receive quality. From in orbit experience with UWE-3 and UWE-4, the packet loss during the uplink can be *much* higher than the downlink (*acknowledgment problem*).
4. Dynamic network spanning: new components must be added with zero configuration. E.g. if a satellite is in range, it should appear in the network.
5. Advanced functions: encryption, DTN, extendability and compression.

A more detailed requirement overview is shown in the Requirements section. With experience in mind from UWE-1 [SPS07] and UWE-3 [Bus+15] new *Compass protocol* was elaborated that meets all set requirements and can either be placed on-top of existing protocol stacks or replace already existing protocols, thus reducing the overhead and the complexity of systems. If required, Compass can be used on some network segments as a transport protocol for other specialized protocols, such as ESA's *CCSDS Mission Operation Services (MO Services)* [Reg+16].

Following sections show summarized approaches for specific requirements. A detailed description of protocol functions are shown in Chapter 4.

### 3.2.1 Addressing

The required protocol must span a dynamic mission network and offer globally unique addresses. In the section 3.1 a concept of *Node* and *System* was introduced. Compass is designed to support three hierarchical domains (also see figure 3.10):

- *Node* domain: communication between different services within a node. In the satellite context, this corresponds to a communication *within* some subsystem.
- *System* domain: communication between nodes, which are grouped to a *System*.
- *Global* domain: communication between any nodes within the network.

Therefore every node has an address of this form:

$$\textit{SystemID} : \textit{SubsystemID}$$

whereat software on each node is organized in standard services. A service has a unique ID and can therefore be accessed from any node via:

$$\textit{SystemID} : \textit{SubsystemID} : \textit{ServiceID}$$

Technically every node is a subsystem, i.e. there is no *separate* hardware component, which represents the system as a whole. Thus a system is always an *emergent entity* formed by selected subsystems. Nonetheless, in real world there is usually an *entry* subsystem (*Entry node*) in every selected system that is usually conditioned by the available communication links (see example below). It is also possible to have different system entry nodes for different physical communication interfaces.

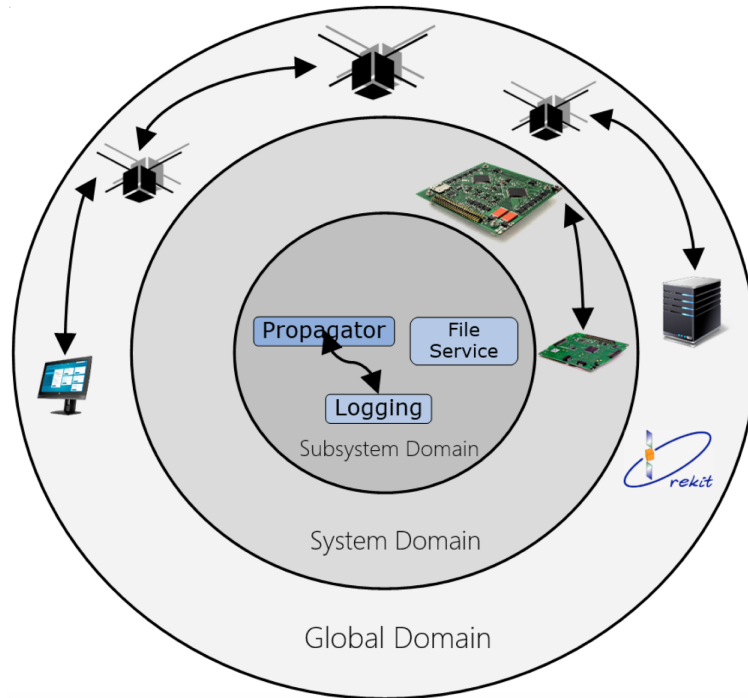


Figure 3.10: Compass Domains Approach

### Example:

The NetSat-1 satellite is built up of OBC, AOCS, Panels and a Thruster Control subsystem. Every subsystem has a globally unique address: `NetSat-1:OBC`, `NetSat-1:AOCS` and so forth. The radio transceiver (off-the-shelf component) is connected via serial interface with the OBC. Therefore any radio-based communication from the mission network to the *NetSat-1* subsystems always passes OBC. So, for *radio* communication OBC is the entry node of the *NetSat-1* system.

A Compass node address consists of 4-bytes SystemID and 1-Byte SubsystemID, thus allowing  $2^{40}$  unique nodes within one mission network. Depending on the domain of the communication, more or less address parts are transmitted within the Compass packet header (intrinsic compression). With special System or Subsystem IDs a packet can be transmitted as *flat* or *deep* broadcast. Depending on the network shape, a broadcast can be utilized as *multicast* or *anycast* (see section 4.2.5).

## 3.2.2 Channels

A node can have multiple physical communication channels. For example the OBC of the NetSat mission has following practically used physical interfaces: 2x serial and 2x I2C. A common workstation has a (W)LAN connection (TCP) and multiple serial connections that can be used during the engineering phase to access the satellite's hardware.

A Compass enabled node can handle an arbitrary number of channels, which are used to send, receive and forward Compass packets and also *automatically* detect nodes in the

neighborhood by analysing the traffic. In the context of Compass, a channel is not necessarily a low-level physical interface (such as serial), i.e. depending on the requirements the Compass protocol can be placed on different OSI-layers. On the software level new channels can easily be implemented using the provided interfaces – for both Java and C Compass implementations. During the thesis channel support for the following interfaces was implemented: TCP/IP sockets, Serial, I2C, AX25, KISS and for several transceiver modules used in ZfT’s missions – see the description of different link types for further details.

### Ground-Ground Links

Ground links are connections between hardware and software nodes on ground, such as: operations workstations, development workstations, ground station servers, mission servers, simulation instances (Matlab, Orekit), test facilities or radio amateurs. Since there exist lesser limitations on ground with respect to power, space or computation speed, these nodes are in most cases based on more powerful machines and can therefore offer more advanced Compass services. They are called *high-level nodes* in contrast to *low-level* satellite nodes. Under realistic conditions ground nodes are inter-connected via local (LAN/WLAN) or wide area network (Internet) – both using TCP/IP sockets. One exception to that is the development/engineering satellite model, as it is usually connected using the serial interface during the development and later is tested using the same radio interface, as used after the launch.

Due to the dynamic nature of the Compass network, a satellite can be seamlessly switched from the serial connection to the radio based – instead of being connected via the serial channel to the developer’s workstation it is then available via the GS server. Similar to that, all high-level nodes can be seamlessly switched from one server connection to another. From experience with ZfT’s satellite projects, the dynamic feature significantly speeds up the cooperative development, as zero configuration is required to perform different types of tests or other cooperative processes.

Furthermore, the Compass protocol provides intrinsic basic GSN capabilities. Multiple Compass activated GS servers automatically provide the information about the availability of a satellite (overpass) by receiving satellite beacons. A Compass-based GSN network will be a basis for the ZfT’s TOM/TIM mission and combine the partner’s ground stations to one Compass mission network.

### Ground-Space Links

Ground-space links are communication channels between the ground relays and the in-orbit satellite systems. Beginning with the UWE-1 mission, all UWE missions rely on the UHF-communication. For several reasons the team at ZfT decided to keep the UHF-link as a main communication channel:

- *Never change a running system:* keep the UHF channel – even if an additional faster channel will be implemented for the first time, such as S-Band (TOM), Laser communication (QUBE).
- *Simplicity:* the ground station hardware, required for the UHF communication, is *comparable* simple and inexpensive.

- *External help*: many radio amateurs with UHF ground stations may provide help – especially during the LEOP phase.

During the UWE-3 project, the Simple Downlink Sharing Convention (SIDS) was introduced in the scope of this thesis. It is used to date by the radio amateur community to forward received satellite packets towards the operator. The protocol was also implemented by Mike Ruprecht (callsign DK3WN) in his decoder software, which is widely used in the radio amateur community. The standard was also used by other Universities (e.g. *UNISAT-6* from GAUSS Srl) and became the main interface of the *SatNOGS* platform, which is also well established in the radio amateur and CubeSat community. Since this convention introduces a *downlink-only* GSN, many ham radios participate, who would otherwise opt-out if the proposed interface would open-up their hardware to external operators for uplink. Nonetheless, a new Compass-based *full GSN* convention would be a helpful follow-up standard.

To enable ground-space communication, it was necessary to separately face the task from both directions, as both sides are composed of entirely different hardware components. For the satellite a *radio Compass channel* was implemented in the embedded Compass OS, as well as a software interface was defined to allow straightforward implementation of future radio channels (e.g. S-Band). On the ground side the Compass-enabled ground station server accesses the TNC via a *serial channel* to send and receive radio packets.

The Compass based ground-space communication has been tested extensively since December 2018 with UWE-4 and since September 2020 with all four NetSat satellites. Therefore, along with ground-ground Links, this type of communication is now considered space proven.

### Space-Space Links (ISL)

In the NetSat mission, ISL is performed via the same UHF-channel *and* frequency as used for the ground-space communication. The disadvantage of the frequency sharing is lower communication throughput of the formation and higher probability of packet collisions. The advantages are:

- A satellite can broadcast its states to multiple neighbor satellites with a single packet.
- Using multiple frequencies *simultaneously* (*Frequency Division*) would require much more sophisticated radio hardware, which is currently not available off-the-shelf for nanosatellites and would also not scale with higher amount of formation satellites that are planned for future missions.
- Using multiple frequencies *successively* (*Time Division*) may introduce additional delays as the result of the frequency switching.

To decrease the probability of collisions, the radio channel implementation of the Compass software utilizes *Carrier Sense Multiple Access* (CSMA) and *Collision Avoidance* (CA).

- *Carrier Sense*: do not send if somebody is currently sending
- *Collision Avoidance*: use random break times before sending after sensed carrier or before resending



These techniques may not work for all available off-the-shelf satellite radios, as many of them do not offer an interface to detect the carrier. In fact, CSMA/CD methods are usually offered by the radio hardware itself – such as the radio subsystem used in ZfT’s missions (*Gomspace AX100*).

The Compass channels also support *constrained-availability*, i.e. a channel or a distant node can only become reachable under certain conditions. For example, the ISL can be realized using comparable high-gain antennas, which first need to be pointed towards the desired target satellite. This can be done both using a priori knowledge of the relative position of the target satellite or using a scanning-scheme if the desired orientation is not known (yet). Using high-gain antennas would also introduce the ability to use *space division* as a multiple access technique. The disadvantage of this type of communication is that it relies on many components, such as determination or scanning algorithms, attitude detection and attitude control, and the required communication constraints may collide with other tasks (e.g. Earth observation). Furthermore, since it takes comparable long time to establish a (high speed) link between two satellites, this technique is not appropriate for tasks where one satellite needs to exchange data at higher frequency with multiple formation siblings.

### Intra-Satellite Links

Two bus technologies are common for the communication between nanosatellite subsystems: Controller Area Network (CAN) bus used for example in the BeeSat-2 to BeeSat-4 from TU Berlin [Kap+16] and I2C-bus used for example on UWE, NetSat, TOM and QUBE missions. For one-to-one communication, e.g. to connect the OBC with the radio subsystem, serial or SPI connections are usually an option – but those are design-wise not multiple access buses (even though they can be extended to star-shaped topology using slave-selection lines). In the scope of this thesis, I2C and serial channels were implemented to support the communication between Compass-enabled subsystems. The intra-satellite interfaces are described in more detail in section 4.4.

### 3.2.3 Routing

One of the major aims of this work was to enable decentralized and dynamic Compass networks, i.e. the network can be extended without configuration, such that newly appeared systems are accessible by all other existing systems without a priori knowledge. To reach this goal, every node in such a network needs to broadcast its own identity (*network beacons*) via its connection channels. A node that *receives* a network beacon, gains knowledge about its neighborhood and can therefore detect new nodes or detect the inactivity of an already known node if no beacon was received from that node for some period of time (e.g. one minute).

A network beacon is only sent to direct channel neighbors and is *not* forwarded to nodes further away (*hop number* > 1) for several reasons:

- *Scalability*: a newly inserted node would only increase traffic in its direct neighborhood. With *deep broadcasting*, a node would increase the global traffic proportional to the global amount of nodes.
- Protection of *low-throughput channels*, e.g. ground-space channels are used as less as possible for the communication of meta data (beacons).

- Reduction of the *network load*

A node gains *deeper* knowledge of the entire network from beacons, which contain the network knowledge (routing map) of the beacon’s sender. This way the knowledge about a new node is indirectly transported beacon-by-beacon into the deeper network. If a new node accesses the network, its network knowledge will be increased with the first received beacon. Please approach the example below. It is evident that the speed of the *knowledge propagation* depends on the beacon rates (e.g. every 10s) and their timings with respect to each other.

### Example:

A satellite has several interconnected (I2C bus) subsystem nodes (see blue boxes in the figure 3.11), whereat the network knowledge of every subsystem is only limited to its siblings, thus resulting in a satellite-only subnetwork. The ground segment consists of a radio-activated ground server, company’s mission server and several workstations (orange boxes). During the overpass, the OBDH subsystem receives a network beacon from the ground segment and instantly gains knowledge about the entire ground segment subnetwork. This knowledge will be later transferred to other subsystems as soon as the `NetSat:OBDH` broadcasts its beacon via the I2C bus. On the ground side the situation is similar: first the `GS:Server` is aware of new space nodes, then its knowledge is transported as beacons toward `ZfT:Server` and so forth.

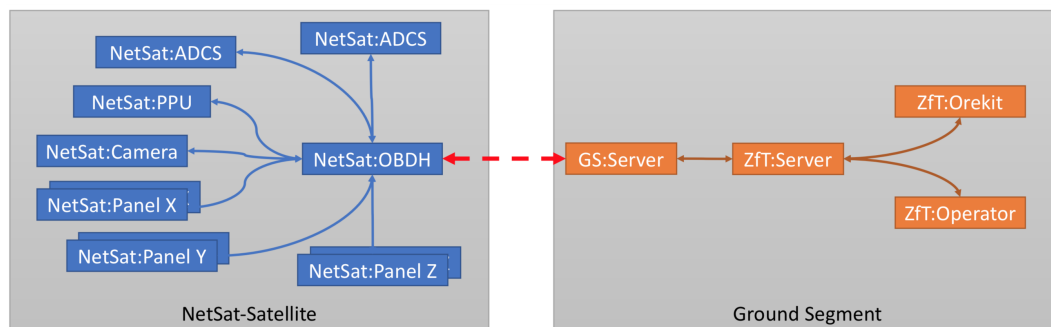


Figure 3.11: Routing Example, connection of two subnetworks

Due to the dynamic nature of the network knowledge, the routing map must be stored in the random access memory – which can become a problem for nodes with very limited resources (e.g. only 4kB of RAM). As can be seen in the example, a real-life mission network contains both high-dynamic parts (e.g. workstations, test facilities) and static parts (satellite’s subsystems). Compass allows to predefine a priori network knowledge for a node, e.g. every subsystem of the satellite’s flight model “knows” about its direct neighborhood without receiving any beacon. This static knowledge can be stored in the program memory, thus reducing the consumption of the dynamic main memory.

### 3.2.4 Advanced Protocol Functions

In addition to the mentioned basic protocol capabilities used for the unification of the communication, more advanced functions are necessary depending on the specific channel

properties. Here only the major functions are mentioned – a full list of functions can be found in section 4.2.

### Delay Tolerant Networking (DTN)

DTN describes a protocol architecture that is used to overcome technical problems with sparsely connected heterogeneous networks. One prominent example is the *Interplanetary Internet* (IPN) developed by NASA [ABH18]. In the scope of Compass, DTN is the ability to *store-and-forward*, i.e. a packet can be sent to a currently not available node along the *last known* path, whereat the packet is stored by the last node with DTN capabilities and remains there until the next hop becomes available again. Every Compass packet can either be configured as *DTN-activated* or as *burst* packet. The former will enforce the so called *hop-acknowledgements*, i.e. the packet reception is acknowledged by every node along its way to the target and, if possible, stored on a DTN-activated node if the target is not currently accessible. The latter non-acknowledged mode is used for broadcasting or enable communication with very asymmetric links with respect to the packet loss. From experience with UWE-3, UWE-4 and NetSat there are usually time periods where the downlink is *much* better than the uplink. In this case relying on the acknowledgement from ground would massively decrease the downlink throughput. Thus, for data transmission tasks *bulk-acknowledgements* are utilized by the File service – see section 4.3.4 for more details.

### Encryption

The Compass protocol supports the built-in XTEA symmetric encryption. XTEA has been chosen, as it can be implemented on very low-level hardware and it does not require handshaking or key exchange, which would become unusable in some realistic (i.e. experienced) scenarios. It supports the encryption using either the sender's or the receiver's key. The former provides also an intrinsic packet signing functionality.

### Tunneling

With the *tunneling* unknown protocol packets can be forwarded to any node within the Compass mission network. The tunneling is provided by the *Compass Tunnel* service.

#### Example: Tunneling

A Compass-enabled ground station has been configured to track a *non-Compass* UNISAT-6 satellite. For this task the satellite receives a *logical* Compass address UNIS:1 and an address of the node to which all packets must be forwarded (GS:Operator). From now on, all incoming packets from the UNISAT-6 satellite will be enveloped by Compass packets and forwarded to GS:Operator. The operator's workstation can extract the original data from the payload and visualize the data using satellite-specific GUI components.

Tunnel service is also used to tunnel (*not* forward) local Compass traffic to some distant node. During the satellite operations at ZfT the traffic between the satellites and the current (auto-)operator is additionally tunnelled to all other operators in the network. The

Compass Operations front-end handles "real" incoming packets and tunnelled-in Compass packets differently (it remains passive).

During the NetSat operations two Compass-enabled *Software Defined Radio* (SDR) receivers were used as redundant downlink channels. Both radios were running on separate workstations and used the Tunnel service to pass received *raw-bytes* to the ground station server that in-turn decodes the packets depending on the protocol configuration of the currently tracked satellites.

### 3.3 High-level Functionality

After a standard for functional and communication interfacing is established, the next step is to enrich the nodes with higher-level functionality. In context of Compass, some specific functionality is offered as Compass service. In the following sections the basic approach is shown, of how single tasks are realized with different services – a more detailed description can be found in the Chapter 4.

#### 3.3.1 Services

The Compass protocol supports up to  $2^{16}$  different Service IDs, whereat the first 200 IDs are reserved for current and future standard services and are therefore globally unique along all satellite missions. In the scope of this work only globally unique services are described, thus the Chapter 4 is at the same time the documentation of the current Compass protocol definition. IDs above 200 are freely defined by the satellite developers and are therefore unique only within a specific mission.

Regarding the OSI model, a Compass service is located on the *Layer 5 (Session Layer)*. One exception to that is the default *Network Service (ServiceID=0)*, which is used for *network features* and therefore affects Layers 2-4: *Data Link, Network* and *Transport*.

#### 3.3.2 Network Features

All available Compass network features are implemented in the default *Network service*:

- network beacon propagation
- DTN acknowledgements
- DTN packet buffering and flushing
- beacon consumption

A more detailed description can be found in the section 4.3.1. Depending on the available memory and computational resources of a (low-powered) node, the Network service functionality may be limited – e.g. lacking DTN capabilities or limited memory for network knowledge. Nonetheless, the information about those limitations are indirectly propagated to the entire network via the network beacons – see section 4.3.1.

Due to the in-build Compass header compression, the ID (0) of the Network service is not transmitted in the protocol header, thus reducing the overhead.

### 3.3.3 Telemetry, Tracking and Control (TT&C)

All *TT&C* tasks are performed based on the uniform *Model* interface, i.e.:

- the telemetry is received as a *model update* packet
- the control is achieved by changing values of the remote *model tree* and by calling remote commands
- the tracking is conducted by the ground station node, where all values (current satellite, overpass, distance, Doppler etc.) are also represented in the ground station's model tree

All model changes are synchronized – either automatically or manually – via the *Model service*. Model update packets from the satellites are automatically updating the model knowledge on the receiving node. The operator can either subscribe to all or to some specific model changes on the remote node.

The console-like commanding is enabled by the *Command service*, which is used as a simple text-based interface to execute commands remotely. In UWE-3 commanding interface was used much often than in UWE-4 and NetSat missions, e.g. to run experiments, activate processes or prepare file transmission. In the latter two, Model service has replaced most of the former commands. The Command service is nonetheless useful for TC tasks that cannot be made (meaningfully) controllable via Model service.

The remote data acquisition is offered by the *Automatic Record and Report service* (*ARR service*). Being an extension of the Model service, it is capable of storing all model changes to a file and can later be (automatically) downlinked via the *File service* to the (auto-)operator's workstation. Satellite subsystems without local file storage can store their recordings in *remote files*, located on a different subsystem – provided that they offer *Compass Network drives*.

More advanced tele-control activities are enabled by utilizing the *Tiny script executor service*. Compiled scripts can either be sent directly as a packet to the remote Tiny service, or the remote service is pointed to some previously uploaded file via File Link service.

### 3.3.4 Testing and Fault Diagnostics

It is highly advantageous to have testing tools during *all* mission stages:

- *Logging*: receive information about the current state of one or multiple processes
- *Unit-testing*: tree-ordered list of automatic self-testing routines
- *Network testing*: capability to test a specific network route

The logging is provided by the *Log service*, which is deactivated by default to reduce the network load. A log message can be created at any point in the embedded code and is automatically transmitted towards the current operator's address via the Log service. If a node has local file storage, this service offers file storage capabilities for local and incoming log messages. During the NetSat operations, the OBC was storing messages from all subsystems to a local file, which was periodically downloaded at night by the auto-operator. On the receiver's side (operations node), the log message must be visualized in a clearly arranged view.

A *unit test* is a set of routines that are used to determine whether the current implementation is faultless and suitable with respect to its task. Usually, a unit test has multiple

subordinated tests, which all need to become “green” in order to proof the correctness of the parent entry. A common way is to create unit tests on different complexity levels, i.e. first to test the low-level functionality and then move one abstraction layer higher to test software portions that rely on the already tested ones. With the *Unit Test service* the operator can enquire a tree of all available tests of a remote system and selectively execute them. Even though unit tests are extensively used by the ZfT’s team, in the scope of this thesis it will not be described *how to implement* a unit test (as it is very implementation specific), instead a way is proposed of how to *provide access* to existing unit test implementations.

At times it may be necessary to test the quality of some specific connection route, i.e. to derive the round-trip time and packet loss ratio. With *Echo service* all incoming messages are bounced back as-is to the sender. An echo packet can either be created manually in a console-like window or multiple packets can be generated automatically using a network testing GUI (*Echo View*). The *Echo View* can visualize the packet loss and mean round-trip time along with some additionally selected model variables, such as current antenna orientation, distance to the satellite, frequencies etc. Using the CSV export function all gathered values can be used to derive the correlation between the link quality and the selected conditions. Please consider section 6.6 for more details on the front-end GUI implementation.

### 3.3.5 File Link

Beginning with the UWE-3 mission, many operation tasks are file-based. In-orbit recordings on different subsystems are stored in the corresponding local files, e.g. magnetic field measurements on AOCS or S-values of the UHF transmitter on the OBDH subsystem – both actively performed in the UWE-3 mission [Bus+15]. The files are then downlinked using the *Downlink service* to ground during one or multiple overpasses.

Numerous scripts were uplinked as files using the *Uplink service* and were for example used to perform several *attitude control experiments* with UWE-3 satellite and to execute *Orbit control* experiments with NanoFEEP thrusters in the UWE-4 mission. In the NetSat mission, several Tiny scripts were uploaded to fix some unexpected misbehavior and to perform experiments over several orbits, e.g. capture values from the GPS, magnetic field and other sensors. The files were automatically downloaded by the auto-operator during the night overpasses.

#### File storage

The OBC and the AOCS subsystems in the UWE-4 and NetSat satellites have only 16 kB RAM – too limited to hold memory files with a size of 200 to 1200 Kb (commonly experienced sizes). Memory files are also non persistent and are lost after a reset. Due to file storage requirements, the OBC was equipped with 2 NAND plus 8 FRAM chips and the AOCS with a single NAND flash chip. Unfortunately no open-source file system could be found that fits in the OBC’s program memory and supports more than one chip. In the scope of this work a file operating system was implemented, which both can be used locally on very low-powered devices to access SPI-connected NAND and FRAM chips (*Uwe File System – UFS*) and can also be accessed via the *Network File System (NFS)* Compass service. It is the re-implementation of the UWE-3’s UFS file system, which has

been developed by Artur Gasparian [Gas12]. The original version had only support for one drive and only one specific flash memory type.

### Weak Asymmetric Links

The UHF communication in both directions with UWE and NetSat satellites has several demanding problems. The down- and uplink cannot be performed simultaneously, as both modes use the same frequency and the same communication path. In the University's UHF ground station configuration the switch between the downlink and the uplink lasts 300-500 ms (transceiver, amplifier and TNC switching times). From experience with the UWE-3 and the UWE-4, it is highly advantageous to use radio protocols that can be handled by external amateur radio stations: *FSK 9k6* for the physical layer (9600 bauds per second) and either *Compass* or *AX.25* for the layer above. If possible, a single UHF radio packet should not exceed 200-250 bytes (*roughly* 1600 bauds). Now, if every packet would need to be acknowledged, 300-500 ms delay (depending on the UHF hardware) will be added after *every* packet – thus drastically reducing the throughput.

In addition to that, the UWE team usually experienced large differences with respect to the uplink and downlink performance. Thus, resulting in usually non-received acknowledgements – leading to repeated transmission, additional transmitter switching times and so forth. This is comparable minor issue for the standard human operations (such as commands), but becomes unbearable for the file transmission.

For the file transmission the so called *bulk acknowledgement* was introduced. That is, before the transmission, the file is logically separated into *chunks*, whereat each chunk has a (selectable) size of 50-200 bytes. After the downlink or uplink has been initiated, *all* chunks are transmitted without acknowledgements. The opposite party detects the gaps during the reception and enquires missing chunks by sending a *chunk bit-map* – which can be interpreted as a bulk acknowledgement. The entire communication process must be handled automatically by the *File service*, so that the operator only needs to specify which files needs to be uplinked or downlinked either directly or in the operations-scheduler.

### 3.3.6 Dynamic Code Execution

In the context of this work, the Dynamic Code Execution (DCE) is the ability of a node to execute code/scripts/algorithms – without the need to re-flash the entire subsystem software image. Even though the on-board software may be re-flashed in space (which has been done multiple times with UWE-3 and UWE-4), this process is highly hazardous and requires time-consuming software image testing.

The ability to run dynamic code was a large step forward with UWE-3, as it allowed to test many *attitude control algorithms* with comparable low efforts and without affecting the robustness of the satellite. In satellite formations, the DCE becomes a requirement for many reasons:

- *Formation control* can be realized by distributing (generated) orbit control scripts – either from ground or from a master satellite (*autonomous formation control*).
- *Task scheduling* to perform dynamic mission tasks, e.g. capture images depending on constraints (orbit, time).
- *Goal-based operations*: a promising operations approach, profoundly researched by *Tiago Nogueira* in his on-going dissertation [NFS17].

- *Experiment execution*: new experiments can be realized without flashing numerous affected subsystems on several satellites.

There exist multiple common ways to enable DCE on high-level machines, i.e. by utilizing available script languages, such as *python* or *java-script*. However, these languages require *comparable* powerful systems and are not runnable on ultra-low power microcontrollers. There also exist more humble languages, such as the *Lua* [IFC10], *Pike*, *Pawn* or *Jim Tcl* [Ben10]. Unfortunately, they are also not suitable for very low-power subsystems, as they all require at least 50 kB (Lua in the *most basic* configuration) ROM space and also do not fulfil requirements that will be stated later.

A second option is to design an additional satellite subsystem that is capable of executing higher languages. All four NetSat satellites carry a Raspberry based *Payload Computer Subsystem*, which can be activated by the OBC on demand to pre-compute some algorithms, values or schedules (which would otherwise be computed on ground). This Linux-based Computing subsystem is included as a test platform for advanced in-orbit autonomy experiments in NetSat but it does not fulfil the requirement of DCE being executed on any specific subsystem (at any time).

It became evident that there were only two options to enable the DCE on all subsystems: upgrade the hardware of all subsystems or to design a new scripting language. Since the first option was out of the question (due to the fixed hardware design), a new scripting language called *Tiny* was introduced as a part of this thesis' realization. Based on the experience during this work, numerous features were implemented:

- *sandbox interpreter* with own stack – no stack overflow of the subsystem possible
- program and data share the same space with very dense *byte code* and extremely low stack consumption
- runs also on 16 bit microcontrollers and consumes *10 kB ROM* and *0.1* to 1 kB RAM (depending on needs)
- automatic *pause and resume* – does not affects the reactivity of a single-threaded system
- introduces *Tiny multi-threading* on a *single-threaded* system
- external C functions with *variable number of arguments*
- integrated *exception handling*
- support for *asynchronous* external functions
- a standalone *Tiny IDE*, which is used to *compose*, *compile*, *decompile* and *debug* Tiny scripts
- *Matlab* and *Java-Script* support in the Tiny IDE to mimic functions for debugging purposes that are not available on the local machine
- a set of Compass-related external functions to access local/remote models, execute local/remote commands and create custom packets

A simple Tiny script can have a size of under 10 bytes, whereat in more realistic scenarios the script size varies between 100 and 800 bytes (*attitude control algorithm*). All satellite subsystems developed by the ZfT and the University's UWE team support Tiny execution, which is offered by the *Tiny service*.



## Database Support

It is common practice to store all relevant outgoing and incoming traffic during the ground-space communication in a mission database. A mission database is used for many purposes, e.g. to:

- perform post-processing of incoming satellite data
- reconstruct the communication during a defined time period, e.g. to tackle occurred problems
- load traffic for some time span (e.g. last 24 hours) into a newly started Compass Operations front-end

The structure of a mission database cannot be easily standardized, as it is dictated by mission aims. At ZfT separate databases are installed for every satellite mission. Every database contains tables for Compass packets, model updates and raw data received from radio amateurs.

Since the Compass network can be used for multiple satellite missions, the operator's node would need to have several separate database connections in order to get access to the historical data of a specific mission. With the *Database service* the database communication is also handled by Compass, thus avoiding further side channels and provide historical data in SQL-agnostic way. That is, any Compass node can check if some other node (e.g. mission server node) offers database functionality – and if so, request stored packets with desired properties, such as time, addresses etc.

## 3.4 Summary

In this chapter approaches were shown that have been used to realize the thesis' goals. The elaborated protocol configuration for the UWE-4 and NetSat missions is shown in figure 3.12 (see figure 3.1 for comparison). The software of the ground station server (ground relay) was designed in such a way that all active missions (UWE-3, UWE-4 and NetSat) are manageable in the same (super) mission network – it chooses different Layer 2 protocols depending on the currently passing satellite. All satellite subsystems became independent from the OBC and can be accessed directly from the mission network. For example a panel subsystem can be unplugged from the EM satellite, connected via Serial interface to a developer's Compass Operations front-end node and gain access to the entire mission network again – with zero configuration.

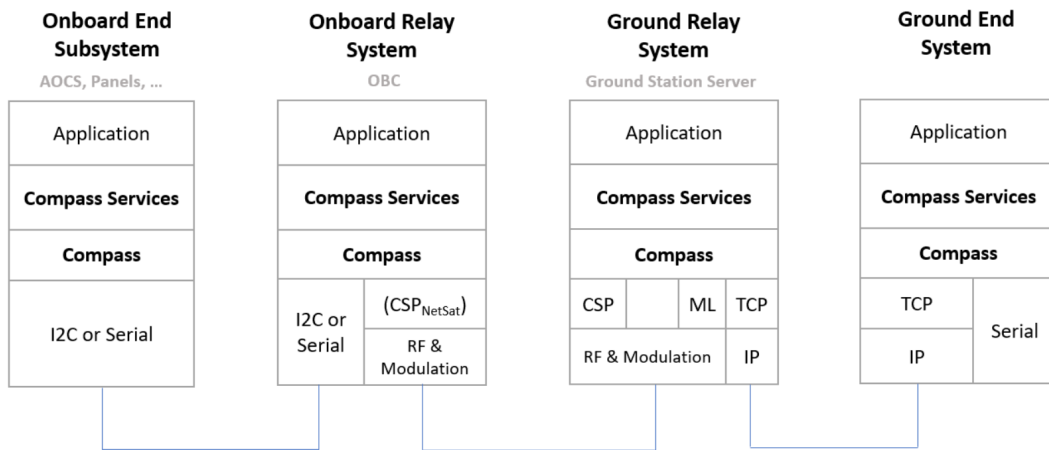


Figure 3.12: Protocol configuration of the UWE-4 and NetSat missions.

## 4 | Compass Protocol

The Compass protocol was designed to sort out multiple problems that were experienced with previous UWE missions and to face additional challenges in multi-satellite missions. An overview of all defined requirements is shown in section 2.1.

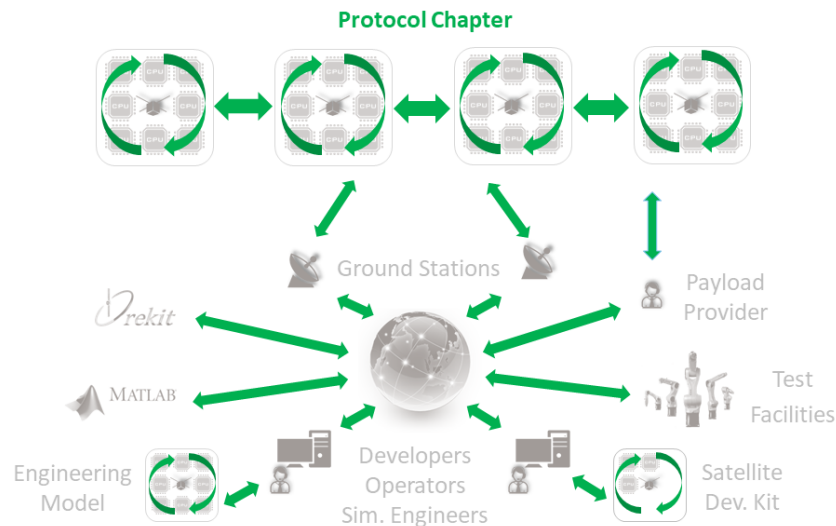


Figure 4.1: Coverage of the protocol chapter

In a nutshell, the main aims of the protocol are:

- Dynamic self-configuring and decentral *Compass network*, which bonds all space and ground mission nodes.
- Offer standard services for all common communication tasks: commanding, monitoring, file transfer, etc.
- *Minimize overhead* but at the same time offer numerous core functions for different scenarios.
- Solutions for *error prone links*: re-transmission, CRC, DTN.
- Operability on *asymmetric links* (up vs. down) with respect to the speed and the packet loss rate
- Network *scalability* similar to the Internet
- The *middleware* must be runnable both on high-end workstations and on 16 bit microcontrollers (min 2 kb RAM, 10 kb ROM).

Channel	Description	CompassNode	Compass OS	Compass OS (guest)
TCP/IP client	Connects to remote servers	x		x
TCP/IP server	Accepts incoming connections	x		x
Serial	Serial interface	x	x	x
I2C	Any-master I2C	ongoing	x	ongoing
Lithium-1 radio	Channel to Lithium-1 UHF		x	x
AX100 radio	Channel to Gomspace AX100		x	x

Table 4.1: Supported channels on different platforms

The *Compass Middleware* is a collective term for the implementation of the protocol core functionality and services. In order to unfold its full capabilities, the middleware is dependent on platform specific functionalities, such as available file system or access to hardware interfaces.

For high-level systems (Windows, Linux, OSX) Java was selected for the implementation of *CompassNode*. The entire implementation is available as a single file and can be included in any Java-based application to gain access to existing Compass networks. Beside the application specific code, the *CompassNode* covers all layers shown in figure 3.12 (ground nodes). At ZfT it is used as a base for: the *Compass Operations* front-end, two *UHF ground stations*, several *Mission servers*, *Matlab* simulations, the *Orekit* orbit simulation framework, *SDR* nodes and custom nodes used for in-the-loop testing.

The implementation for satellite subsystems (*Compass OS*) is a more demanding task, as the minimum hardware requirements are dictated by the smallest available microcontroller (UWE-4's power processing unit and NetSat's Thruster control): 16 bit, 4 kB RAM, 64 kB ROM. Even though it was projected to implement the embedded middleware with compile switches and thereby control its hardware requirements (e.g. to activate File service only on systems with connected flash chips), the *full-fledged* configuration must be made runnable on the OBC subsystem: 16 bit, 16 kB RAM, 256 kB ROM – from which at least 50% are reserved for application code. Embedded *Compass OS* was implemented during this thesis to meet the requirements and is described in detail in Chapter 5. It can either be executed on bare-metal or run as guest OS on another operating systems (e.g. FreeRTOS, Windows, Linux, OSX). Currently it supports:

- TI MSP430 and MSP430x family
- Atmel SAM
- Linux, Windows, OSX

## 4.1 Overview

Inspired by different studied protocols, shown in section 2.6, the Compass protocol was designed to harmonize the communication between all involved mission space and ground systems. It fulfills the requirements of a large formation mission and provides an abundant set of functions:

- *DTN* support: buffer and forward, retransmission, etc.
- *Burst mode*, i.e. non acknowledged communication, e.g. for broadcasting data – similar to UDP.

- *Hierarchical addressing*, covers the global domain (4B system address – similar to IPv4), the system domain (1B sub-system address) and the subsystem domain (2B Service ID – similar to TCP port).
- Payload encryption based on *XTEA* algorithm with variable number of rounds.
- Switchable *GZIP payload compression* for large packets between nodes with high computation power.
- All functions, except the payload compression, are available on very low-power 16 bit nodes.
- Automatic and manual per-packet *routing* with the ability to partially pre-define routing segments.
- *Network Awareness* based on passive and active neighborhood discovery and failure detection.
- Automatic header compression.

Compass is currently running on all UWE-4 and NetSat subsystems as well as on all ground segment nodes, resulting in a homogeneous mission network. Since the NetSat launch in September 2020, the number of distinctly accessible Compass-enabled subsystems in-orbit was raised to 58. All accessible nodes can communicate with each other using the same common protocol, thus offering new beneficial concepts. For example, in the pre-launch phase the cameras of the NetSat flight-models were calibrated autonomously by three cooperating nodes: motion simulator node, panel subsystem node and Compass Operations node (visualization). The Compass protocol was designed in such a way that the entire network structure is derived automatically using passive (listen to packets on hardware channels) and active methods (broadcast and request network beacons), thus forming local network knowledge. This knowledge is used during the transmission to automatically find the best active route to the target node by taking in account the number of hops and the maximum speed along the route.

## 4.2 Packet Definition

This section describes in detail the composition of a Compass packet header. The overview of the packet structure is shown in table 4.2. Every header field is described in more detail in the corresponding section.

The basic idea is to use a dynamic header, i.e. first one to four *header description bytes* are used to switch on/off or configure the optional protocol fields. Besides the comparable complex parsing, this approach has several advantages: compactness, feature richness and expandability. Every header description byte has a *HALT bit*, which denotes if another header description byte follows ( $HALT = 1 \implies$  no more bytes). In the current Compass version (September 2020) only the first three bytes are described.

The header description bytes were designed in such a way that in most common scenarios only the first byte is transmitted. Further bytes are transmitted for more advanced packet configurations, i.e. included timestamps, signature, larger payloads etc. Therefore the most used header fields are activated in the first byte, the lesser used in the second and the most seldom fields in the third. The absence of a header description byte implies,

Flags	PID	From		...	To		PID	API	Time	Size	Payload	SGN	CRC
		Sys	Sub		Sys	Sub							

Field	Description	Size [B]	Mandatory
Flags	header definition bytes	1-4	x
PID	ID of the Packet	2	x
From ... To	Source, up to 6 intermediate nodes and target	2-40	
API	Target Service ID	1-2	
Time	Absolute time (UNIX, ms) or duration	2-6	
Size	Payload size in bytes	1-4	
Payload	Payload bytes		
SGN	Signature	2	
CRC	Fletcher pseudo-checksum	2	

Table 4.2: Compass packet description

Byte1	HALT	Route Fmt. set	Size Len1	ABuf	Buf	API set	REF set	PID set
Byte2	HALT	16bit API set	Size Len2	Time	RSV		CRC	SGN
Byte3	HALT	-	-	-	Is Error	Is Urgent	Is Zipped	Is Encr.

Table 4.3: Header bytes, with only first byte being mandatory

Bit	Description
HALT	True if no more header bytes will follow
Route Fmt. set	If set, the routing definition byte is available
Size Len1	First bit of the Size-field configuration
Use ADTN	True if the answer packet must be delivered using DTN
Use DTN	True if the packet must be delivered using DTN
API set	True if the service ID is set
REF set	True if the packet is referenced to another Packet ID
PID set	True if the Packet ID value is set

Table 4.4: First header byte (mandatory)

Bit	Description
HALT	True if no more header bytes will follow
16bit API set	If set, the API field is 2 bytes long instead of 1
Size Len2	Optional second bit of the Size-field configuration
Time	Time included
RSV	Reserved
CRC Set	True if CRC is set (see CRC section)
SGN Set	True if the signature is set (see SGN section)

Table 4.5: Second header byte (optional)

Bit	Description
HALT	True if no more header bytes will follow
...	-
Is Error	True if the payload is a service error message
Is Urgent	True if the packet is <i>urgent</i>
Is Encrypted	True if the payload is encrypted using XTEA
Is ZIPped	True if the payload is compressed using gzip

Table 4.6: Third header byte (optional, advanced functions)

that all functions/bits described there are considered 0.

In the most compact configuration the packet consists only of two (header) bytes:

- First header byte: only *address format* field is activated.
- Second header byte: address format byte with `Hops#=0` (no address entries).

However such a packet does not have any data or addresses, so it can only be used for some meta-communication between two hops – or to handle multiple link access.

If a field has been deactivated, it is not transmitted and the corresponding value is considered 0. Conversely, header fields with 0 values are deactivated during the encoding, i.e. 0 values are not transmitted. For example, deactivated PID and API implies `PID=0`, `API=0`. The route with size 0 implies two equal addresses (from, to) with `SystemID=0` and `SubsystemID=0` – and is usable for inter-process communication *within* one subsystem.

### 4.2.1 Addressing

A Compass Address consists always of two fields: *SystemID* (1-4 bytes) and *SubsystemID* (1 byte). To improve the readability of IDs, it is a good practice to have number-to-text mappings in the front-end software. The Compass Operations front-end provides the ability to create such mappings and synchronizes them with the mission network via *Registry service*. It is entirely up to the mission designed to decide the logical meaning of system and subsystem. For instance:

- A satellite as a whole is a system with its subsystems being subsystem nodes in the context of Compass.

- The mission operations division can be a system, whereat single workstations are subsystem nodes.

The system ID can also be viewed as a logical group of multiple subsystems, whereat the subsystems can be distributed freely across the Compass network.

In contrast to the classic from-to addressing approach, a Compass packet can hold up to 8 addresses, where the first denotes the sender, the last the receiver and the remaining optional (up to 6) addresses are the desired intermediate nodes. In nominal mode, a newly created Compass packet contains only the local and desired receiver's address.

Even though only 8 addresses can be set in a packet, the real route length is unlimited. For further nodes (hops number > 8) the sender only knows the first 6 intermediate hops of the route, whereas the remaining nodes are discovered during the transmission as the target node gets closer and closer. Please read section Route for more information.

### 4.2.2 Payload size

Compass supports both small and very large packets up to theoretical 4 GB. To reduce the overhead for small packets, the size of the *payload size* field is set by two bits **Size Len1** and **Size Len2** in the header byte.

Size Len1	Size Len2	Size [B]	Description
0	0	0	No payload included
0	1	4	Large payload (>65535 bytes)
1	0	1	Small payload (<256 bytes)
1	1	2	Mid-sized payload (<65536 bytes)

Table 4.7: Payload size field setting

In UWE and NetSat missions all ground-space and space-space packets are below 256 bytes, thus only first bit is used. However, between ground nodes the packet size can reach several megabytes – for example when a Compass Operations front-end node polls via Model Service the current antenna camera image from the ground station node.

### 4.2.3 Time field

The **Time** is used to activate the **time** field, which is used to carry an absolute timestamp (UNIX format in milliseconds, beginning with January 1, 1970 00:00:00.000 GMT). The absolute time is used by many Compass services do denote: last change of the requested model value (Model Service), creation time of the log message (Log Service), and creation time of the beacon (Network Service).

### 4.2.4 Route Format

If the routing bit is disabled, the **route**-field consists only of two addresses, whereat each system's address is 1 byte long. The unchecked *route*-Bit is suited best for packets



without manually defined intermediate nodes and system addresses below 256 – and should therefore be the default mode for small missions. If the routing bit is checked, a *routing format byte* is defined right after the PID:

	SYS size		Hop counter			# of Hops		
Bit	7	6	5	4	3	2	1	0

From		To	
System	Subsystem	System	Subsystem
1B	1B	1B	1B

Table 4.8: Routing Entry if Routing bit is disabled

The amount of hops describes the number of contained addresses (system-subsystem duplets), and the hop counter shows the number of already passed hops. Using the counter:

- a receiver can identify if it is intended to be the next hop for the received packet (the counter is used as a 0-based pointer to one of the stored hops)
- which hop is intended to be the next receiver
- each hop increases the hop counter
- a hop that is not in the routing list, does not increase the counter

During the answer creation, the stored hops are reversed and the counter begins with 0. Now the answer packet will take the same route in the opposite order back to the origin of the referenced packet.

The SYS size describes the amount of bytes used to encode the system address (0, 1, 2 or 4 bytes). If set to 0, only subsystem IDs are set for transmission (inter-subsystem communication). If no format byte is available, the default values are:

- System and subsystem address size: 1 byte each
- Amount of addresses: 2 (sender, receiver)

### Example:

A packet from A:1 is designed to reach D:3 via B:0 and C:0:

$$A : 1 \rightarrow B : 0 \rightarrow C : 0 \rightarrow D : 3$$

So the routing byte is:

0x03 (0b00000011, Hop counter=0, Hops=3) during the creation  
 0x13 (0b00010011, Hop counter=1, Hops=3) when received by B  
 0x23 (0b00100011, Hop counter=2, Hops=3) when received by C  
 0x33 (0b00110011, Hop counter=3, Hops=3) at its destination

## 4.2.5 Route

A route can contain:

- 0 addresses. The system assumes that sender and receiver addresses are both {SYS=0, SUB=0}.
- 1 address. Only sender is defined, the receiver is {SYS=0, SUB=0}.
- 2 addresses. Sender and receiver are defined (default).
- >2 addresses. Some intermediate nodes are manually defined.

### SystemID=0 (flat system broadcast)

The SystemID=0 is a special undefined system name, which matches any SystemID. If a node receives a packet with zero SystemID, it considers itself as a target as long as there is a subsystem ID match.

### SubsystemID=0 (flat subsystem broadcast)

The SubsystemID=0 is also a special subsystem address and has a similar meaning to a SystemID=0. If a node receives a packet with zero Subsystem, it considers itself as a target as long as there is a system ID match.

## 4.2.6 SGN

If the SGN-flag is set in the header, a 2-byte *signature of the payload* must be placed after the payload area. If the given senders signature of the payload does not match the calculated signature of the receiver, the packet is immediately refused.

Listing 4.1: Signature Calculation

```

1  const char STR_SECRET_KEY [] = "SecretCode123";
2  const uint8_t STR_SECRET_KEY_LEN = (sizeof(STR_SECRET_KEY) / sizeof(char) - 1);
3  void calcSGN(unsigned const char *data, uint16_t size, uint8_t *sgn, uint8_t clear)
4  {
5      if(clear)
6          sgn[0] = sgn[1] = 0;
7      uint16_t i;
8      char c;
9      for(i = 0; i < size; i++) {
10         c = data[i] ^ (STR_SECRET_KEY[i%STR_SECRET_KEY_LEN]);
11         sgn[0] = sgn[0] + c + sgn[1];
12         sgn[1] = sgn[1] + (255 - c) + sgn[0];
13     }

```

The signature calculation is shown in listing 4.1. Obviously, the secret key must be same on both nodes. This functionality can for example be used to avoid accidental execution of specific commands, i.e. execution is only possible if the command packet is properly signed and not signed packets are omitted. So, signing is rather used for *safety* and not for *security* reasons. For critical tasks the more sophisticated signing can be achieved with XTEA encryption.

### 4.2.7 CRC

If the CRC-flag is set in the header, a Fletcher's checksum of the entire packet (incl. signature) must be placed at the very end of the packet. If the given checksum does not match the calculated checksum, the packet is immediately refused.

Listing 4.2: Pseudo CRC Fletcher algorithm

---

```

1 void Buffer_calcCRC(const uint8_t *data, uint16_t size, uint8_t *crc, uint8_t clear
2 ) {
3     if(clear)
4         crc[0] = crc[1] = 0;
5     uint16_t i;
6     for(i = 0; i < size; i++) {
7         crc[0] += data[i];
8         crc[1] += crc[0];
9     }
}
```

---

### 4.2.8 Error

The error flag can be optionally used by services to denote the answer as error. In case of an exceptional handling, a service usually uses its payload section to mark packets error packets – e.g. by using special service packet types. With the error flag the answer receiver can detect in a standardized way, whether the answer contains enquired data or some error message/code. The error flag can also be used as a service-to-service NACK.

### 4.2.9 Urgent

This flag is used by the receiver to engage faster (immediate) consumption. Depending on the Compass implementation, the packet handler may have a queue of packets from numerous hardware channels and sequentially handle the packets depending on their priority. To ensure the stability, the process of reading packets from channels and the process of packet handling is done separately, e.g. in different threads.

In some situations the packet handling should be processed as fast as possible to achieve real-time like execution. It is entirely up to the receiver's implementation to decide how to handle urgent packets. So, depending on the implementation the urgent packet handling can begin directly in a task that is only responsible for the channel read-out. However, for some channels – such as I2C – this would imply that the packet handling must be performed inside the I2C interrupt/transaction, which may lead to long bus occupancy/stall.

Urgent packets are not a replacement for packet priorities. In Compass the service ID denotes packet priority, therefore NetworkAPI packets have the highest priority.

For example in the UWE-4 mission, urgent packets were used by the OBC to disable magnetorquers on all panels shortly before magnetic field measurements.

### 4.2.10 Encryption

The Compass protocol supports symmetrical XTEA payload encryption with variable number of rounds (in the default configuration 64 encryption rounds are used). XTEA has been chosen to also support encryption on extremely hardware limited devices. The

7	Key	0=receiver, 1=senders key
6	-	Reserved
5	-	
4	Rounds	Number of rounds: 0 = 32, 1 =
3		64, 2 = 128, 3 = 256
2	Padded	Number of padded bytes: [0-7]
1		
0		

Table 4.9: Encryption info byte

payload can either be encrypted with the sender's or the receiver's 128 bit key. There are several security options for how to set-up an encrypted communication link:

- *Low* (group key): only one critical node A (e.g. a satellite subsystem) becomes a key. All other nodes use that key to communicate with A.
- *Middle* (node key): every node becomes one Key and uses that key for the transmission. The receiver must possess the key from the sender in order to be able to decrypt. In this scenario, the encryption is also an authentication of sender.
- *High* (link key): every node-node pair becomes a key, i.e. a node holds up to  $(n - 1)$  unique keys for every other node within the network. This can lead to  $n \times (n - 1)$  keys in a network of  $n$  nodes.

Encryption process is conducted as follows:

- The payload is first padded to achieve its length being a multiple of 8 bytes.
- The payload is encrypted using the appropriate key.
- Encryption-info byte (table 4.9) is added at the beginning of the payload.

### 4.2.11 Zip Compression

The ZIP compression is advantageous for large Compass packets and is automatically activated between high-power nodes if the payload size exceeds 50 kB – presupposed that all nodes along the route can handle the forwarding of large packets. If a packet is flagged with ZIP, the payload block is compressed using the *gZIP* algorithm (see Java example in listing 4.3).

Listing 4.3: Signature Calculation

```

1 public static byte[] decompress(byte[] data) throws IOException {
2     GZIPInputStream gzip = new GZIPInputStream(new ByteArrayInputStream(data));
3     data = Util.toBytes(gzip); // Convert InputStream to bytes
4     gzip.close();
5     return data;
6 }
7
8 public static byte[] compress(byte[] data) throws IOException {
9     ByteArrayOutputStream baos = new ByteArrayOutputStream();
10    GZIPOutputStream zop = null;

```

```

11     try {
12         zop = new GZIPOutputStream(baos);
13         zop.write(data);
14         zop.flush();
15     } catch(Exception ex) {
16     } finally {
17         if(zop != null) try {
18             zop.close();
19         } catch(Exception ex) {}
20     }
21     return baos.toByteArray();
22 }

```

### 4.3 Services

In the previous section core capabilities of the Compass protocol were described that are sufficient for an application to enter an existing network and handle all upper layer traffic in the application code. In addition to the basic functionality, a set of Compass services were defined and implemented during this thesis. An overview of all available standard services is shown in table 4.10. All three missions (UWE-3, UWE-4 and NetSat), including auxiliary nodes, were successfully realized using only these services. That is, no further side-channels, services or higher protocols are used between any mission-relevant systems. This is due to the fact, that the services were designed in such a way that any desired complex high-level functionality can be traced back to the usage of one or more standard services. At ZfT one common mission super-network is used for UWE-3, UWE-4 and NetSat missions. An overview of all services used between ground and space systems is shown in figure 4.2. Services between ground systems are shown in figure 4.3.

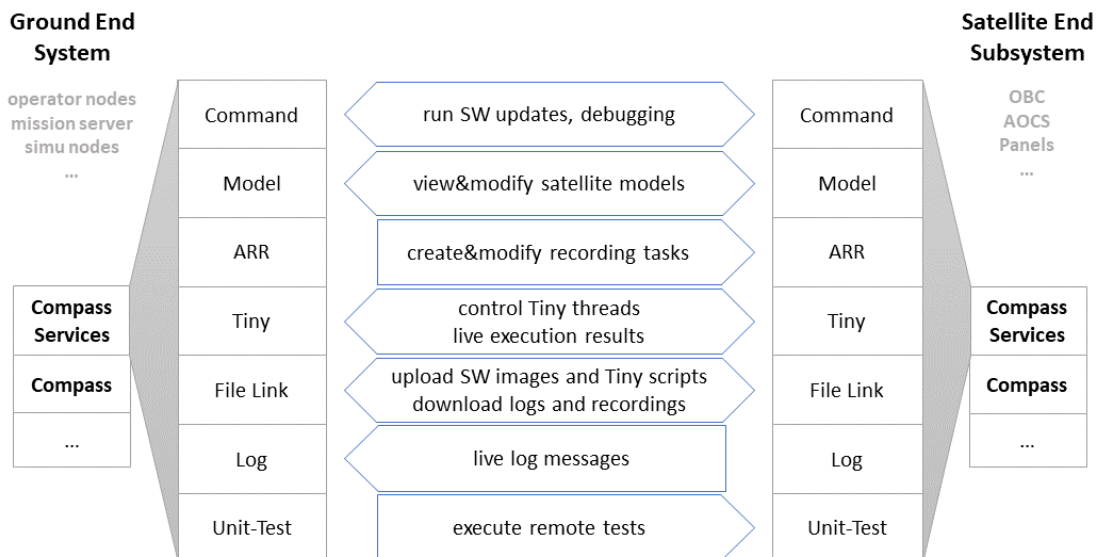


Figure 4.2: Overview of all services used between ground and space systems in the common UWE-3, UWE-4 and NetSat mission network

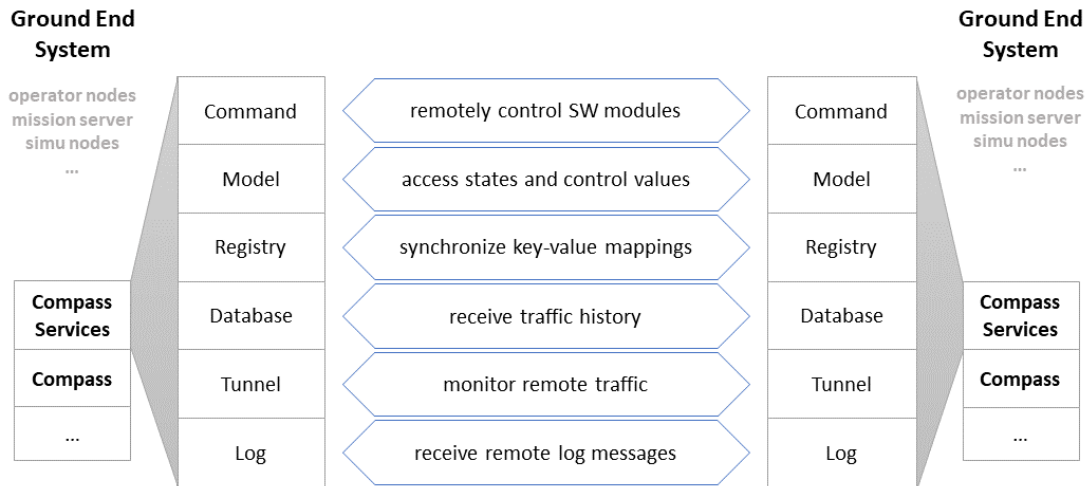


Figure 4.3: Overview of all services used between ground systems in the common UWE-3, UWE-4 and NetSat mission network

Most of the monitor and control tasks of the space and ground systems is performed with *Model service* – an implementation of the MTBA approach described in chapter 3.1. A model is a hierarchically ordered set of name-value pairs: parameters, control values, sensor values, configurations and any other entity that can be represented as such pairs. The model service can deliver the entire model structure on request, offers full access to model values and offers subscriptions to model changes.

*Command service* offers human-readable interface for remote command execution. Technically many commands can also be represented as model values, e.g. instead of sending `Thruster activate` command to the OBC system an equivalent OBC's control model value can be changed to 1 ( $Thruster/Activity=1$ ). In NetSat commands are used mostly for debugging purposes, e.g. scan the I2C bus and output the report or output the current network knowledge.

All file-related tasks are performed with delay-tolerant *File service* (a combination of Downlink and Uplink services). From the point of view of a user it is similar to FTP: list remote drives and files, upload or download file, delete files or format drives. On the implementation level it is inspired by the CCSDS File Delivery Protocol (CFDP) and provides robust file transfer over several orbits. In UWE-3 and UWE-4 the protocol was mainly used to upload Tiny-script based experiments and download experiment results (model value recordings). On NetSat this service is used to download log files, upload Tiny-based schedules and Tiny-based experiment execution.

The *Tiny service* offers on-demand execution of incoming scripts, which are transmitted as compiled Tiny byte code in payload of Tiny service packets. If local file storage is available, it additionally supports *Tiny-Threads* – scripts that are automatically loaded on start-up from the file system and executed concurrently.

Service	Name	Description
0	Network	Receive and consume network beacons, build-up network awareness
1	Echo	Bounce back received payload to the sender with updated timestamp
2	Command	Execute human-readable commands (i.e. remote console)
3	Uplink	Auto transmission of files. Handles retransmission of file chunks
4	Downlink	Auto receipt of files. Handles re-request of file chunks
5	Log	Processes log messages from other nodes
8	Unit-Test	Execute remote tests or offer local tests
11	NFS	Network File Interface. Write to remote file systems. No auto re-transmission
13	Tiny script	Execute Tiny script and control Tiny threads
14	Model	Access and modify remote node models
16	ARR	Automatic model Recording and Reporting service
18	Database	Used to load or store Compass packets from/to a database
21	Tunnel	Used to establish user-defined protocol connection between two nodes
22	File	Delay-tolerant file transmission

Table 4.10: Currently Implemented services

In the following sections only the service packet payload is described in detail, i.e. for the sake of readability the Compass protocol header is omitted.

### 4.3.1 Network

Many advanced core functions of the protocol are enabled by the *Network service*. It gains knowledge of the current network state (available nodes etc.) and promotes this knowledge to the neighbors via network beacons.

The embedded Compass OS middleware implementation contains basic and advanced Network service functions, which can be activated if the host system has enough storage and computation capabilities. In the high-level CompassNode implementation all available functions are activated by default. The Network service offers:

- *ACK handling*: generate and process acknowledge packets for packets marked with DTN bit. That is, send back an ACK if the packet has reached its destination or the to-forward packet has been stored locally and will be transmitted later (store-and-forward).
- *Network Beacon creation*. Periodically transmit own network knowledge via all local channels.
- *Auto-Routing-Table*: build up a routing table using the processed traffic and incoming network beacons.
- *DTN (advanced mode)*: store packets marked with DTN bit and take over the responsibility for further delivery. Requires local file system.

#### Routing table

An example shown in the figure 4.4 shows principles of the routing table generation. Initially, all routing tables contain only the local address (grey). For the sake of simplicity, all nodes in the example transmit their beacons *simultaneously*. The colors denote the source of the information: blue = from first beacon, green = second beacon and so on. A network beacon contains the sender's routing table and is transmitted only to direct neighbors. Every received network beacon is used to update the local routing map, thus leading to subsequent propagation of new information along the network. This scheme enables very good network scalability, i.e. the amount of transmitted beacons between direct neighbors stays same independent of the node number. For devices with very limited memory resources a single-entry routing map can be used to pass all packets to a more capable neighbor (cf. Gateway).

#### Network packets

The first byte of a Network packet payload denotes the service packet type. One exception to that is the ACK packet, which does not have any payload. An overview of all available Network service packet types is shown in table 4.11.



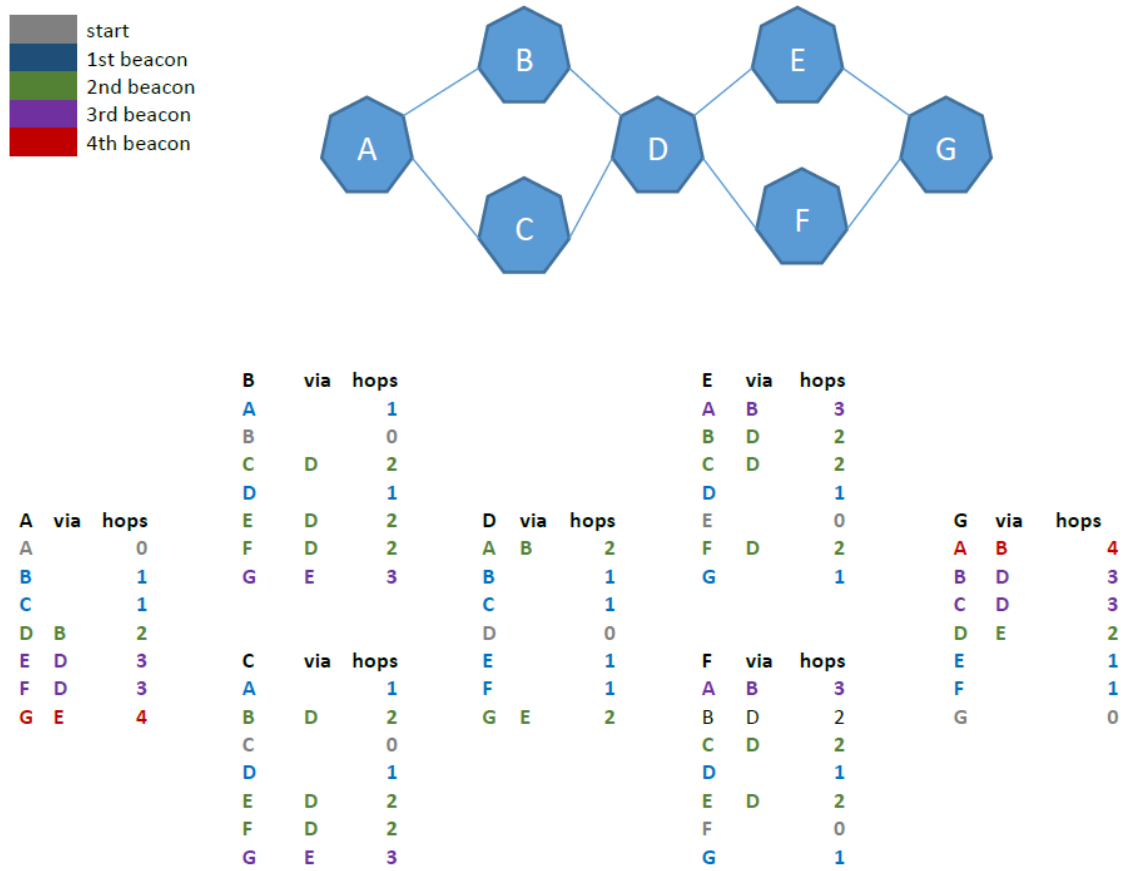


Figure 4.4: Routing table example

Type	Description
-	ACK message (no payload). Is received if a buffered Packet was successfully received by a hop.
0	Beacon. Contains the routing map of the sender. Is sent periodically (default 10s) to all direct neighbors
1	Request beacon. Can be used to manually request a beacon from a node
5	Extended Beacon with additional information for every routing entry: max. speed and max. packet size

Table 4.11: Network service packets

Payload
1

Table 4.12: Network service: beacon request

## Network Beacons

A node periodically transmits beacons to all direct neighbors. It contains densely packed local routing map with additional known information about the nodes (see table 4.13). The payload starts with a type byte (`value = 0`), followed by a list of routing entries. The first entry denotes the local node itself and is the only entry with `Hops=0`. Both service implementations (CompassNode and Compass OS) support simple and extended beacons with additional routing *entry info* bytes (max. speed and max. packet size).

The order of the entries appearance is important for the reproduction of the network's tree structure – see example below.

### Example:

A beacon contains 5 entries in that order:

Received Entries	Visually indented
[GS:Server, hops=0, ...]	[GS:Server, hops=0, ...]
[NetSat:OBDH, hops=1, ...]	[NetSat:OBDH, hops=1, ...]
[NetSat:PPU1, hops=2, ...]	[NetSat:PPU1, hops=2, ...]
[NetSat:PPU2, hops=2, ...]	[NetSat:PPU2, hops=2, ...]
[GS:Operator, hops=1, ...]	[GS:Operator, hops=1, ...]

The beacon is apparently from `GS:Server`, as it is the first entry, followed by 4 entries that are reachable via `GS:Server`. For clarity, the hop number of each entry is used for tabular indentation in the right column. The indent now results in a tree structure, which represents the entire route to every node. In this example both PPU's are reachable via the OBDH subsystem, which in turn is reachable via the `GS:Server`. An example of the routing table visualization is shown in figure 4.5.

## Static and dynamic routing entries

The Compass protocol definition does not force nodes to transmit or handle incoming network beacons. For example in UWE or NetSat only the OBC has a direct channel to the radio connection and is therefore considered a *space relay*. Hence only the OBC needs to keep track of the dynamic network neighborhood, whereas the remaining subsystems only have static routing entries pointing to the sibling subsystems and use OBC as a gateway to transmit packets to unknown targets.

Depending on the particular system software implementation, the local routing map may consist either only of statically defined entries or also contain space for dynamic entries. For dynamic maps, the incoming beacons are used either to update existing knowledge or to expand the knowledge with new nodes. The local routing map is also automatically updated without incoming beacons, i.e. on packet reception or forwarding the packet's sender and all previous nodes in the packet's route list are considered active and are placed into the routing map.

In both Compass implementations (Compass OS and CompassNode) a new derived routing entry  $E'$  replaces an existing one  $E$  if one of this is true:

- $E'$  has a lesser hop-number (shorter route)
- $E'$  has the same hop-number but higher speed

Type	Node Entry 1											...	Entry N
0	DTN	Active	Size mode	Addr size		Hops			Age	Addr	Info		...
1B	1bit	1bit	1bit	1bit	1bit	1bit	1bit	1bit	1B	1-5B	1B		...
	1B												

Info byte							
HALT	DL only	Speed			Packet Size		
1	1bit	1bit	1bit	1bit	1bit	1bit	1bit

Field	Description																
DTN	If 1, the node supports packet buffering																
Active	1 if the node is currently reachable. Depending on the implementation a node can be considered reachable if for example: <ul style="list-style-type: none"> <li>a packet was received from that node in the last 60 s</li> <li>the routing entry is flagged as <i>always active</i></li> </ul>																
Size mode	0=normal mode, 1=micro mode. In <i>micro mode</i> , only one byte is required to encode system and subsystem ID																
Addr size	<p>Normal mode: address stored in 2-5 bytes</p> <table border="1"> <tbody> <tr> <td>00</td> <td>1B System 1B Subsystem. For SystemID &lt; 256</td> </tr> <tr> <td>01</td> <td>2B System 1B Subsystem. For SystemID &lt; 2<sup>16</sup></td> </tr> <tr> <td>10</td> <td>3B System 1B Subsystem. For SystemID &lt; 2<sup>24</sup></td> </tr> <tr> <td>11</td> <td>4B System 1B Subsystem. For SystemID &lt; 2<sup>32</sup></td> </tr> </tbody> </table> <p>Micro mode: address stored in a single byte</p> <table border="1"> <tbody> <tr> <td>00</td> <td>xxxxxxx  8 bit System (0-255), Subsystem=0</td> </tr> <tr> <td>01</td> <td>xxxxxx xx 6 bit System (0-63), Subsystem (0-3)</td> </tr> <tr> <td>10</td> <td>xxxx xxxx 4 bit System (0-15), Subsystem (0-15)</td> </tr> <tr> <td>11</td> <td>xx xxxxxx 2 bit System (0-3), Subsystem (0-63)</td> </tr> </tbody> </table>	00	1B System 1B Subsystem. For SystemID < 256	01	2B System 1B Subsystem. For SystemID < 2 <sup>16</sup>	10	3B System 1B Subsystem. For SystemID < 2 <sup>24</sup>	11	4B System 1B Subsystem. For SystemID < 2 <sup>32</sup>	00	xxxxxxx  8 bit System (0-255), Subsystem=0	01	xxxxxx xx 6 bit System (0-63), Subsystem (0-3)	10	xxxx xxxx 4 bit System (0-15), Subsystem (0-15)	11	xx xxxxxx 2 bit System (0-3), Subsystem (0-63)
00	1B System 1B Subsystem. For SystemID < 256																
01	2B System 1B Subsystem. For SystemID < 2 <sup>16</sup>																
10	3B System 1B Subsystem. For SystemID < 2 <sup>24</sup>																
11	4B System 1B Subsystem. For SystemID < 2 <sup>32</sup>																
00	xxxxxxx  8 bit System (0-255), Subsystem=0																
01	xxxxxx xx 6 bit System (0-63), Subsystem (0-3)																
10	xxxx xxxx 4 bit System (0-15), Subsystem (0-15)																
11	xx xxxxxx 2 bit System (0-3), Subsystem (0-63)																
Hops	Distance in hops. Hops=0 stands for the local node (first entry). Hops=7 means $\geq 7$ (or unknown) distance.																
Age	Time elapsed since the last packet was received from the node. Connection age as a multiple of 5 s, i.e. Age=2 means 10 s. Depending on the implementation, some of entries may have always Age=0 (e.g. those flagged as <i>always active</i> )																
Address	Address bytes as defined by the <b>size mode</b> and <b>name size</b>																
Info	Node info byte (only in extended beacons)																
DL only	Downlink only entry, e.g. one-way S-Band channel																
Speed	Speed in bauds: 0=1200, 1=9600, 2=38400, 3=57600, 4=EDGE (59200), 5=1 Mbit, 6=10 Mbit, 7=100+ Mbit																
Packet size	Maximum allowed packet size: 0=50B, 1=100B, 2=150B, 3=200B, 4=500B, 5=1Kb, 6=100Kb, 7=1+Mb																

Table 4.13: Network service: (extended) beacon

- $E$  is inactive (entry too old), i.e. no packets were received from the node for longer time

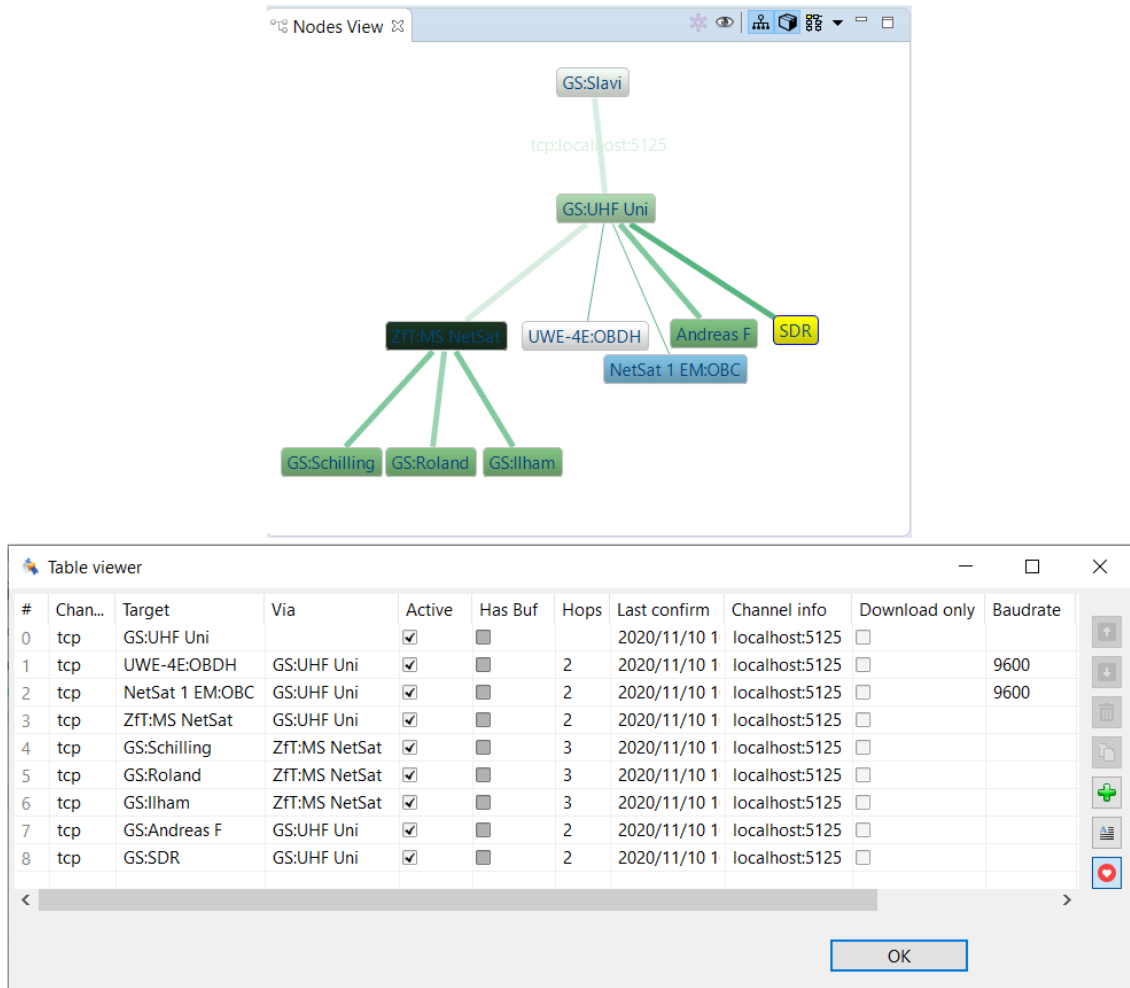


Figure 4.5: Visualization of the local routing map in the Compass Operations frontend. Green and red lines denote active and inactive routes respectively. The width of the lines is denoting the corresponding maximum speed. The age of the knowledge is visualized with fading line color.

### 4.3.2 Echo

The Echo service has a very simple functionality: the payload of every received packet is bounced back to the sender. The header of the answer packet contains the local absolute time and is referenced to the ID of the received packet. This service is mainly used to qualify the communication towards some specific node. The Compass Operations frontend provides a view (*Echo View*, described in section 6.6.10), which periodically transmits

Payload	
"CommandName Param1 Param2 ..."	

Answer	Case
'BAD CMD' + error flag	if command not found
[readable return value]	if answer not empty
'OK'	if answer is empty

Table 4.14: Command service: command request and possible answers

echo packets to a selected node and visualizes mean packet round-trip time and packet loss ratio.

### 4.3.3 Command

The Command service offers a text-based interface to the node's commanding system and is used for remote consoles. The Compass Operations front-end provides the *Command View* to access remote command service (figure 4.6). On UWE and NetSat this service is mainly used for debugging, conduction of in-orbit software updates and as a redundant access to essential functions of other services, such as commands to perform file operations (`cd`, `delete`, `dir`, etc.) that are normally handled by the File service, or to read or write model values with `get` and `set`, instead of using the Model service. In order to receive a list of available commands, an empty command packet is sent to the remote system (table 4.15). The format of a command packet and possible answers are shown in table 4.14. The command service is not intended to be used for *time-tagged* commands. For this purpose the *Tiny service* is utilized, as it supports the execution of scripts with complex constraints and time checks.

Payload	Payload
[EMPTY]	'CMD: Command1, Command2 ...'

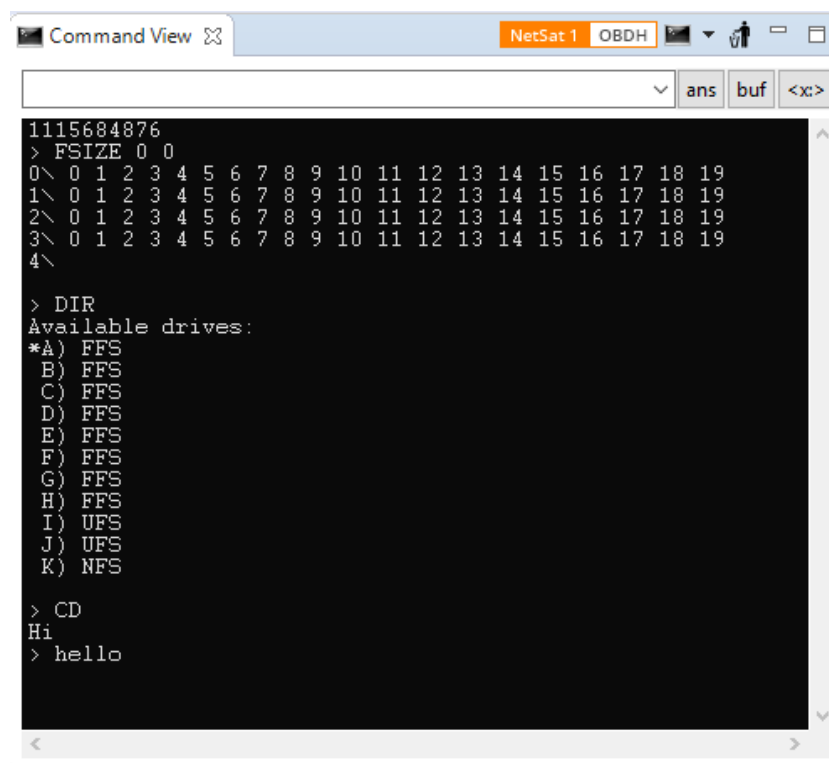
Table 4.15: Command service: command list request and the corresponding answer

#### Command list execution

The Command service supports the execution of multiple semicolon-separated commands, i.e. multiple commands can be invoked with one single packet. For example with

```
delete A:\1\1;write A:\1\1 hello
```

the remote node will first delete the `A:\1\1` file and afterwards create it again with `hello` content. For every command in the list separate answer packet is created by the execution node.



```
Command View NetSat 1 OBDH
1115684876
> FSIZE 0 0
0\ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1\ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
2\ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
3\ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
4\
> DIR
Available drives:
*A) FFS
B) FFS
C) FFS
D) FFS
E) FFS
F) FFS
G) FFS
H) FFS
I) UFS
J) UFS
K) NFS
> CD
Hi
> hello
```

Figure 4.6: Command service front-end in the Compass Operations front-end – here showing a commanding session with the NetSat 1:OBC.

## Answer piping

With piping the command answer can either be written to a file, transmitted to some other node or be omitted. Piping is activated with > character at the end of the command, followed by:

- *NUL*, to omit answer creation. Example: `hello>NUL`
- *SystemID:SubsystemID:ServiceID* to send the answer to a service of the defined node. Example: `hello>1:9:5` would result in `Hi` being sent to the Log service (ID=5) of the node 1:9.
- *Drive:\Dir \File* writes the answer to the defined file. Example: `hello>J:\4\5`

### 4.3.4 Downlink

The Downlink service was specifically designed for low-power devices with limited resources. The file communication is actively *controlled by the file receiver* (ground node). It is feasible for channels with small possible packet sizes (e.g. 200 bytes in UWE and NetSat missions) and long *downlink* ↔ *uplink* switching times. It has been successfully tested in space on board the UWE-3, UWE-4 and NetSat satellites.

This service is responsible for handling file downlink requests. It relies on the file system of the underlying operating system – in the implemented UFS file system of the Compass OS a file is denoted by two numbers: directory number (1-255) and file number (1-255). To overcome long downlink/uplink switching times of the ground relay, the downlink is conducted in *burst mode*. That is, the service splits the requested file into chunks of given size and transmits them all *without* checking the receipt acknowledgements. At the same time, the receiver detects all missing chunks and requests them later using a *chunk bitmap*, whose bits represent missing chunks. This technique allows short single-packet requests of multiple chunks.

A downlink request contains: target file name and desired chunk size (0-255 B). The chunk size is selected in such a way, that it can be transmitted in a single packet through the entire node chain. Due to the limitation of the OBC hardware, maximum packet size for UWE and NetSat missions has been set to 250 bytes.

Dir	File	Chunk size	offset	map
1B	1B	1B	2B	NB

Table 4.16: Downlink API request

Every received chunks contains the file name (directory+file number), the corresponding 0-based chunk index, number of all chunks and the data itself.

Dir	File	Chunk#	Chunks	Data
1B	1B	2B	2B	NB

Table 4.17: Downlink API answer

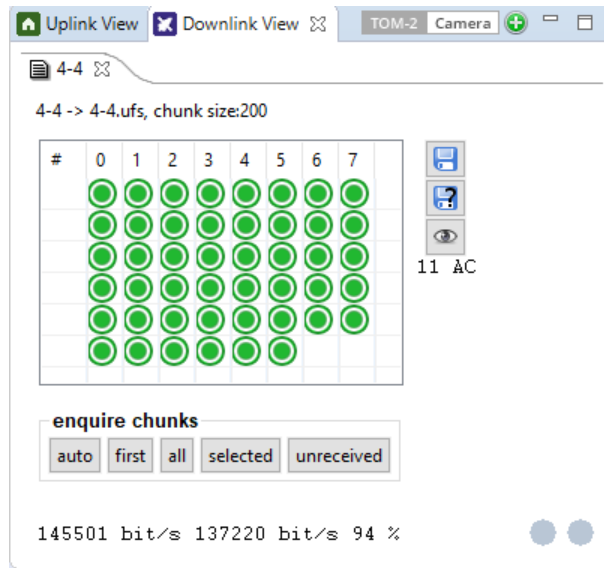


Figure 4.7: Downlink API front-end

## Downlink Process

In UWE-3, UWE-4 and NetSat the downlink process was automatically conducted by the *Downlink View* of the Compass Operations front-end. The operator has to select target node and desired file name (directory+file numbers) and start the download. The automatic process is then performed as follows:

1. send *get all chunks* request
2. wait until the sender stops sending chunks (5s timeout)
3. if some chunks are missing, request them with generated chunk-map and go to 2)

### Example:

The file system of UWE-4 contains the file 7 in the directory 4. The file size is 10000 bytes. The downlink receiving process requests the file with chunk size 100:

```
[COMPASS-HEADER 04dec 07dec 100dec COMPASS-FOOTER]
```

After no further packets were received, the process discovers that chunks 65, 68 and 87 are missing and prepares an additional chunk-mapped request. Since the first lost packet is 65, the offset is set to 65. The range between the last and first lost chunk is  $87 - 65 = 22$ , therefore  $\text{ceil}(22/8) = 3$  bytes are required to build up the request chunk bitmap: `[10010000 00000000 00000010]`. The new request packet is therefore:

```
[HEADER 04dec 07dec 100dec 65dec 10010000bin 00000000bin 00000010bin FOOTER]
```



### 4.3.5 Uplink

This service provides reversed functionality of the Downlink service. The file communication is actively *controlled by the file sender* (ground node). The uplink service is responsible for receiving remote files. Similar to the Downlink service, it abstains from packet acknowledgements. All incoming chunks are first saved in the incoming order in a temporary file and combined later after all file chunks have been received. The *chunks bit map* is used to hold the information about the received chunk numbers. This map is used for three operations:

1. To decide if an incoming chunk has already been saved in the temporary file.
2. The sending process request the map to discover which chunks need to be retransmitted.
3. To detect if the file transmission is finished, i.e. all desired chunks have been received.

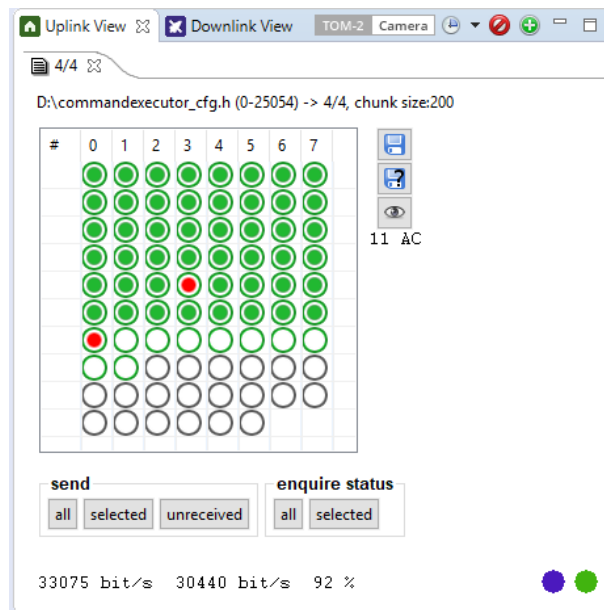


Figure 4.8: Uplink API front-end

### Uplink process

An uplink packet contains six fields: *link id*, target file (directory+file numbers), contained chunk id, total number of chunks and the actual chunk data. The Link ID is used to distinguish uplink sessions, i.e. the uploading process generates new Link ID for every file transmission. In Compass OS implementation, if the receiving uplink service discovers a shift in the Link ID it deletes all already received chunks and creates a new uplink session. In theory the Link ID can be used to perform multiple simultaneous uplinks to one node.

Link#	Dir	File	Chunk#	Chunks	Chunk data
1-255	1B	1B	2B	2B	N bytes

Table 4.18: Uplink API Chunk

To decide retransmission, the uplink process periodically request the receipt chunk-map of the remote node. A state request contains three fields: 0-field to make the packet distinguishable from other uplink packets, *from chunk* field and *to chunk* field.

Link#	Chunk from	Chunk to
0	2B	2B

Table 4.19: Uplink API State request

Dir	File	Chunks	Received	Map from	Map len	Map
1B	1B	2B	2B	2B	2B	N bytes

Table 4.20: Uplink API State answer

When all chunks were received, the temporary file is merged into the desired target file. Hereby, the service traverses the temporary file and writes the chunks in the right order into the target file. Depending on the storage technology, this process may take from 1s (UWE-4 and NetSat using FRAM storage) up to several minutes (UWE-3 using NAND storage). After a successful merge, a *Merged Status* Packet is transmitted.

Dir	File	
1B	1B	0xFF 0xFF (2B)

Table 4.21: Merge finished packet

At ZfT the uplink process is handled in the *Uplink View* of the Compass Operations front-end, in which the operator selects target node, desired file location, local file contents and starts the transmission.

### 4.3.6 Log

The Log service is used to process incoming or forwarded log messages. In Compass OS many processes (file access, detected communication errors) create log messages. In default mode all UWE and NetSat subsystems suppress the transmission of own log messages. If the transmission is activated (via Model service, figure 4.9), all log messages are transmitted either to a default recipient node (mission server) or to the current operator's node. In addition the OBC (space relay) can be set-up to store own and forwarded logs from other subsystems to a local file.

The log service supports four different severity levels: *Debug*, *Info*, *Warning* and *Error*. The prefix of a log message can either be 'Debug:', 'Info:', 'Warning:' or 'Error:'. If none of those is found, the severity of the message is considered *Info*. The log service does not produce any answer to incoming log messages.

Payload
'Debug/Info/Warning/Error: ...'

Table 4.22: Log service: Message with optional Severy-String

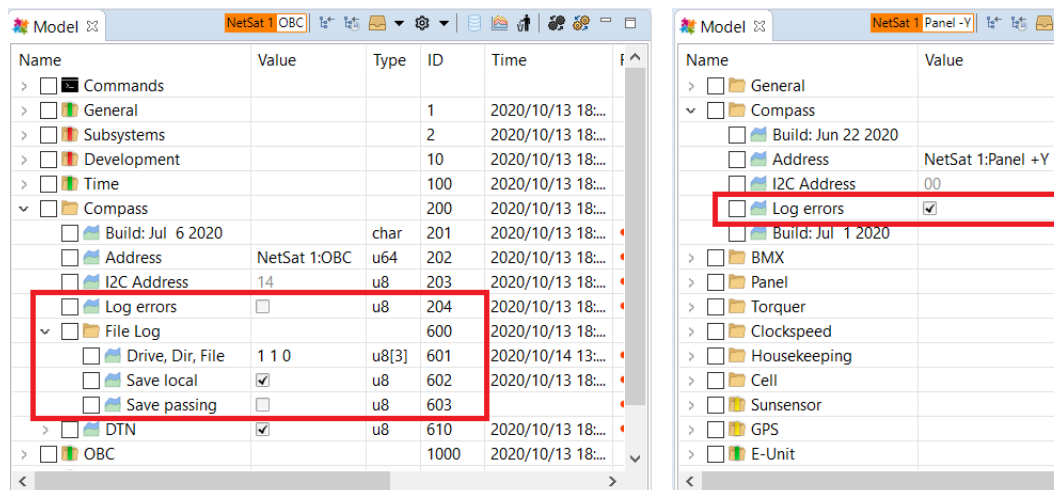


Figure 4.9: Log service: activation of log transmission and log storage using Model service

### 4.3.7 Unit-Test

The Unit-Test service is used to execute unit tests, inform listeners about the current test progress and the test result. In Compass OS unit tests are created with special macros that are collected during the compile process and become accessible by the Unit-Test service. In the current implementation up to 255 tests are supported. The Unit-Test service has following tasks:

- list available tests as a tree ordered list of id-name mappings
- execute selected tests on request
- stop test execution on request
- deliver current test progress

Type	Description	Parameters
0	Request a list of available tests	
1	Execute selected tests	N ID1 ...IDN
2	Stop testing	
3	List entry	ID ID-P PROG-LEN NAME
5	Test result	ID PROG SUCCESS MSG

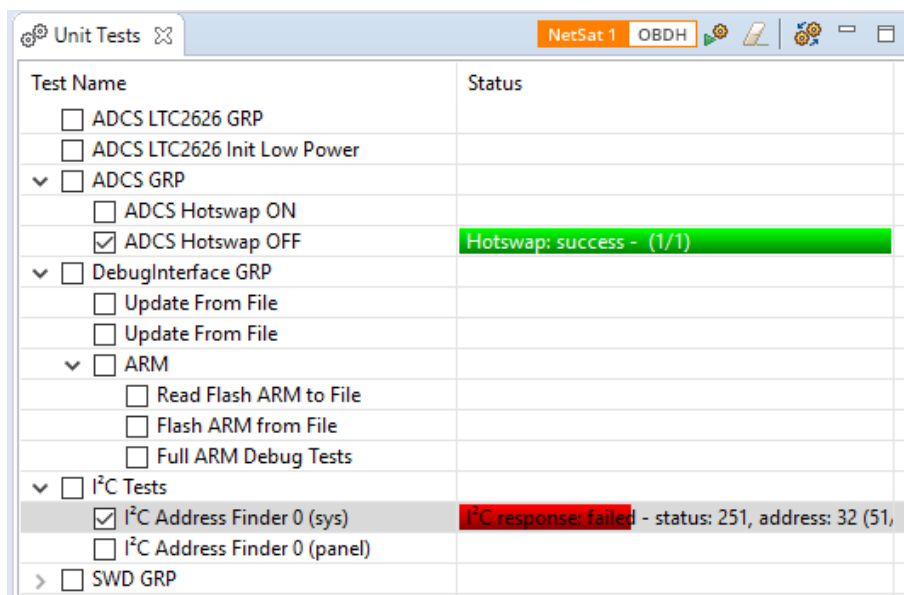


Figure 4.10: Unit-Test service: user front-end

### Request available tests

To receive available tests from a remote node a service packet with type 0 is transmitted. The target node answers with multiple packets, whereat each packet contains the description of one test (see table 4.23).

### Test execution

To execute remote tests a list of desired IDs is transmitted to the Unit-Test service. In the Compass OS implementation only one list can be executed at a time, i.e. if a new list arrives during the test execution, all current tests are canceled and the new list is processed. The payload of the request packet starts with packet type, followed by a number of IDs and a list of the actual IDs.

Payload	Type	ID	Parent ID	Progress Len	Name
0	3	1B	1B	1B	remaining

Field	Description
Payload	3 (= List entry)
Parent ID	ID of the parent test. Is used to hierarchically order tests
Progress len	Amount of the progress steps 0 – 255. During the test execution this number is used to calculate the progress percentage
Name	Remaining payload bytes represent the readable name of the test (ASCII)

Table 4.23: Unit-Test service: test entry description

Type	Number N	ID1	...	IDN	Type
1	1B	1B	...	1B	2

Table 4.24: Unit-Test service: execute selected tests and stop execution

During the execution, the service generates progress packets, which are transmitted to the requester. A test generates as many packets as are necessary to properly inform the user, i.e. larger tests generate intermediate progress messages with uprising progress numbers.

Type	ID	Progress	Success	Message
5	1B	1B	1B	remaining

Field	Description
Type	5 (= Test progress)
ID	ID of the test
Progress	In the range of Progress length (see Test list entry). Used for progress percentage calculation
Success	1 for success, 0 for failure
Message	The remaining payload bytes represent the progress/info/error message

Table 4.25: Unit-Test service: progress packet

### 4.3.8 Network File System

The Network File System (NFS) provides global write access to the local file system. This service is not a replacement for the File service, as it does not support delay-tolerant delivery. Therefore this service is feasible on robust links: subsystem-subsystem, workstation-workstation or for communication during the development process. It is for example used on the OBC subsystem to provide storage to subsystems without own persistent memory.

The service provides up to 255 *logical drives*. In Compass OS a drive is either backed by a memory chip (embedded systems) or is represented by a selected directory on a workstation (Windows, Linux, OSX). Every NFS packet starts with a packet type, followed by further parameters. The service never creates an answer, i.e. if acknowledged service communication is desired, DTN can be activated for NFS packets.

In the current ZfT missions the NFS service is never used directly by the operators. Compass OS offers a file system abstraction layer, which is used by embedded software components to uniformly access files on different local drives, i.e. drives based on different

- memory technologies (FRAM, NAND)
- chip vendors (different low-level drivers)
- file systems (Uwe File System and Fast File System – both implemented during this thesis)

In addition, Compass OS allows the definition of remote drives, which forward all file operations to a remote NFS service and are used (in software) in the same way as local drives. Since the AOCS subsystem of the NetSat does not have local persistent storage, three remote drives were defined in its software to enable persistent value storage (e.g. for logging the changes in the AOCS' model) – see figure 4.11.

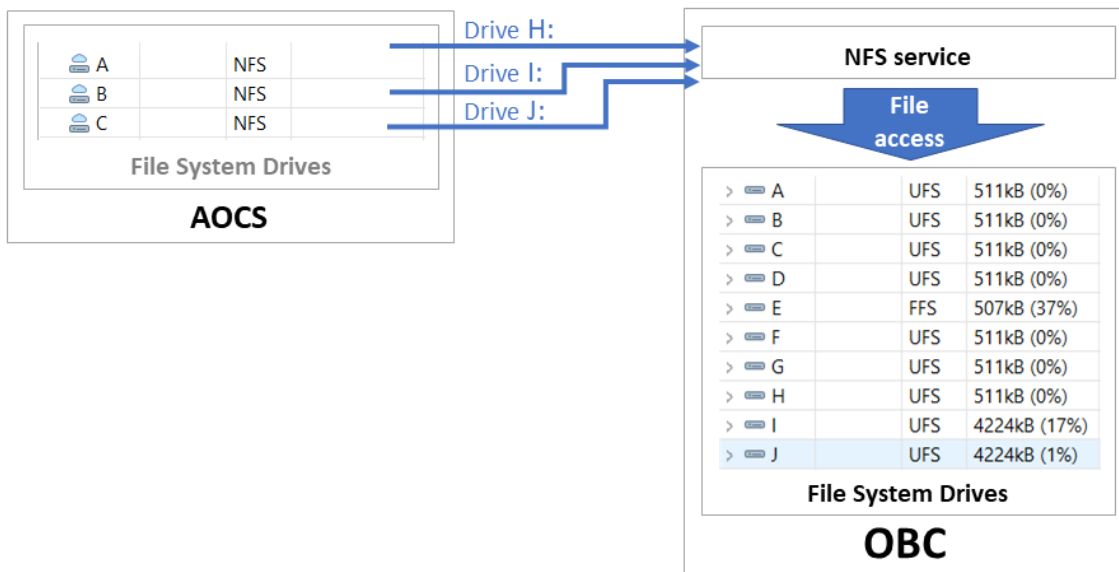


Figure 4.11: NFS service: drive configuration example

### Create file

This packet type is used to create a new file on the given drive. If the file already exists, it is only recreated if `Recreate` is set to 1. The file name is represented by the remaining payload bytes.

Type	DriveID	Recreate	File Name
0	1B	1B	remaining

Table 4.26: NFS service: Create file

### Append file

With this packet binary data is appended at the end of the given file on the specified drive. If the file is not existent, it is automatically created. The *file name* is stored along with the *file name size* and the remaining bytes are used as appending data. For example, for a 2-bytes file name format the **Name Size** is set to 2, followed by 2 file name bytes.

Type	DriveID	Name Size M	File Name	Data
1	1B	1B	MB	remaining

Table 4.27: NFS service: Append file

### Delete file

With this packet a specified file can be deleted. The file name is represented by the remaining payload bytes. The request is ignored if the file does not exist.

Type	DriveID	File Name
2	1B	remaining

Table 4.28: NFS service: Delete file

### Format drive

This packet can be used to clear the specified drive. If the logical drive is backed by a directory on a workstation, all directory contents are deleted. If the drive does not exist, the request is ignored.

Type	DriveID
3	1B

Table 4.29: NFS service: Format drive

### Close file

With this message a remote file is denoted as closed, which can be very useful for file consumers. For example, some local file is filled with image data by a remote node – as soon as the file is denoted as closed, the image is used for post-processing.

Type	DriveID	File Name
4	1B	remaining

Table 4.30: NFS service: Close file

### Cut file

With this request a file can be truncated either from the beginning (**From Head = 1**) or from the end (**From Head = 0**). Depending on the receiver's underlying file system, the implementation may not support one of the truncation modes. The **Length** specifies the amount of bytes to be removed (up to  $2^{32}$  bytes). The NFS service acts *progressively*, i.e. if the *Length* is larger than the file size, then entire file is truncated.

Type	DriveID	From head	Length	File Name
5	1B	1B	4B	remaining

Table 4.31: NFS service: Cut file

### 4.3.9 Tiny script

The Tiny service is used for live execution of compiled Tiny scripts as well as for creation and control of persistent Tiny threads. This service was extensively used in all three missions:

- UWE-3: test of several attitude detection and control algorithms
- UWE-4: multiple cooperating Tiny threads were used to perform orbit maneuver experiments
- NetSat: support LEOP operations, preliminary in-orbit bug fixes, scheduling of payload experiments (thruster, camera)

The service itself does not have any dynamic code execution capabilities. Instead, it acts as a bridge towards the *Tiny interpreter* component (see section 5.5 for more details) that has been developed in the scope of this thesis. The interpreter was deliberately implemented as a stand-alone component in order to make it runnable on the UWE-3 satellite, which at the beginning of this thesis was already in orbit. However, in UWE-4 and NetSat the interpreter was configured with external Compass-related functions, thus enabling the byte code to access local and remote models, create custom Compass packets, etc.

An overview of all currently implemented service functions is shown in table 4.32.

### Live Execution

The most straightforward functionality of the Tiny service is the live execution, i.e. the byte code in the payload is executed right-away with subsequent answer containing either the execution result or some error code.

Type	Byte-Code
61 (=)	remaining

Table 4.33: Tiny service: Execute Tiny byte-code



Type	as ASCII	Description
		Get List of available functions.
58	:	List of available functions.
61	=	Execute Tiny byte-code
62	>	Answer
63	?	List current Tiny threads
76	<i>L</i>	List of Tiny threads
65	<i>A</i>	Tiny thread answer
45	-	Delete Tiny thread
43	+	Create Tiny thread
83	<i>S</i>	Save Tiny thread to file
70	<i>F</i>	Load Tiny thread from file
85	<i>U</i>	Modify Tiny thread

Table 4.32: Tiny service packets

The answer contains the return value of the execution, which either can be a single value (return value or exception code) or a pointer to a Tiny value array. In latter case the content of the array is additionally placed at the end of the return packet.

Type	Value Type	Value	(Data)
62 (>)	1B	1-8B	remaining

Type	Size [B]	Description
0	1	int8
1	1	uint8
2	2	int16
3	2	uint16
4	4	int32
5	4	uint32
6	4	32bit float
7	4	error code
8	8	int64
9	8	uint64
10	8	64bit double
10	4	pointer to tiny array

Table 4.34: Tiny service: Execution Answer and Value Types

### Function listing

To properly compile Tiny script for some target node, Tiny compiler requires a list of available functions on that node. A list of available functions IDs is requested with an empty Tiny service packet. Function IDs must be unique within the mission network,

i.e. functions with the same ID on different machines must perform equally (input-output interface). Since only function IDs are listed by the Tiny service, the *Registry service* can be utilized to obtain their textual representation.

	Type	FunID1	...	FunIDN
0B	58 (:)	2B	...	2B

Table 4.35: Model service: Get function list (left) and list answer (right)

### Thread listing

Every Tiny-enabled node can be set-up to offer memory for so called *Tiny threads*. The memory can be filled with any number of threads until the memory is exceeded. In the Compass OS implementation every thread consumes:

- 20 bytes header data: file location, CRC and settings
- thread stack, e.g. 100b
- space to fit the byte-code

If local file storage is available the service can automatically load Tiny threads from files during the boot-up. All OBC and Panel subsystems of UWE-4 and NetSat missions support up to 5 persistent Tiny threads. For example on NetSat one persistent Tiny script is used to fix some detected misbehavior during the LEOP operations.

With packet type 63, the list of all currently available threads can requested. The answer contains: number of threads, auto-load settings and multiple thread information blocks (see table 4.37). Auto-load settings describe a file range from which persistent threads are loaded during the node's boot-up. This technique allows to create or delete persistent Tiny threads, i.e. threads that are automatically loaded during the boot-up process.

Type
63 (?)

Table 4.36: Tiny service: List Tiny threads

### Thread Control

A Tiny thread can either be created using a single packet or using the File service capabilities. The former procedure is applicable to smaller code fragments (under 250B) that can fit in a single Compass packet. For larger scripts, the script can be uploaded using the File service.

Type	CycleMS	RepeatS	Flags	Byte-Code
43 (+)	2B	2B	1B	remaining

Table 4.38: Tiny service: Add thread

Type	Num N	Free Space	Drive	Dir	FileFrom	FileTo	Thread1	...	ThreadN			
76 (L)	1B	2B	1B	1B	1B	1B	19-26B	...	19-26B			
ThreadID	CRC	Drive	Dir	File	Size	Cycles	CycleMS	RepeatS	Flags	State	ValType	Value
1B	2B	1B	1B	1B	2B	2B	2B	2B	1B	1B	1B	1-8B

Field	Description
Num N	Number of threads
Free Space	Free memory for threads
Drive	
Dir	
FileFrom	Auto-load settings
FileTo	
ThreadN	see rows below
ThreadID	Local ID of the thread
CRC	CRC of the loaded script
Drive	
Dir	Script location (if any)
File	
Size	Size of the script
Cycles	Number of cycles until execution pause
CycleMS	Resume execution after pause of
RepeatS	If not zero, repeat execution if finished after
Flags	Flags: 0b00000LRA
	L   Log active
	R   Repeat active
	A   Active (script disabled on 0)
State	0=finished, 1=in progress
ValType	
Value	Last returned value or error (see table 4.3.9)

Table 4.37: Tiny service: Tiny threads List

Every currently available thread can be deleted with a *Delete* packet. With `DeleteFile=1`, the thread is not only deleted from the execution memory but also from the file system (if existent).

Type	ThreadID	DeleteFile
45 (-)	1B	1B

Table 4.39: Tiny service: Delete thread

With *Update-Packet* settings of an existent thread can be changed: refresh rate, CPU load, repeating and enable-parameter.

Type	ThreadID	CycleMS	RepeatS	Flags
85 (U)	1B	2B	2B	1B

Table 4.40: Tiny service: Update thread

When a thread execution is finished, the service creates an answer packet, which is sent to the last operator's address. A thread answer is very similar to the direct execution answer – it additionally contains the thread ID. In repeated threads answers are transmitted periodically.

Type	ThreadID	Value Type	Value	(Data)
62 (>)	1B	1B	1-8B	remaining

Table 4.41: Tiny service: Thread result

## File Access

The file access capabilities of the Tiny service can be used for multiple scenarios:

- Create persistent threads that are loaded during the boot-up.
- Load larger threads that do not fit into one single Compass packet.
- Dynamic script loading, i.e. a script loads another script on demand.
- Store existent thread for later use.

Type	Drive	Dir	File
70 (F)	1B	1B	1B

Table 4.42: Tiny service: Load thread from file

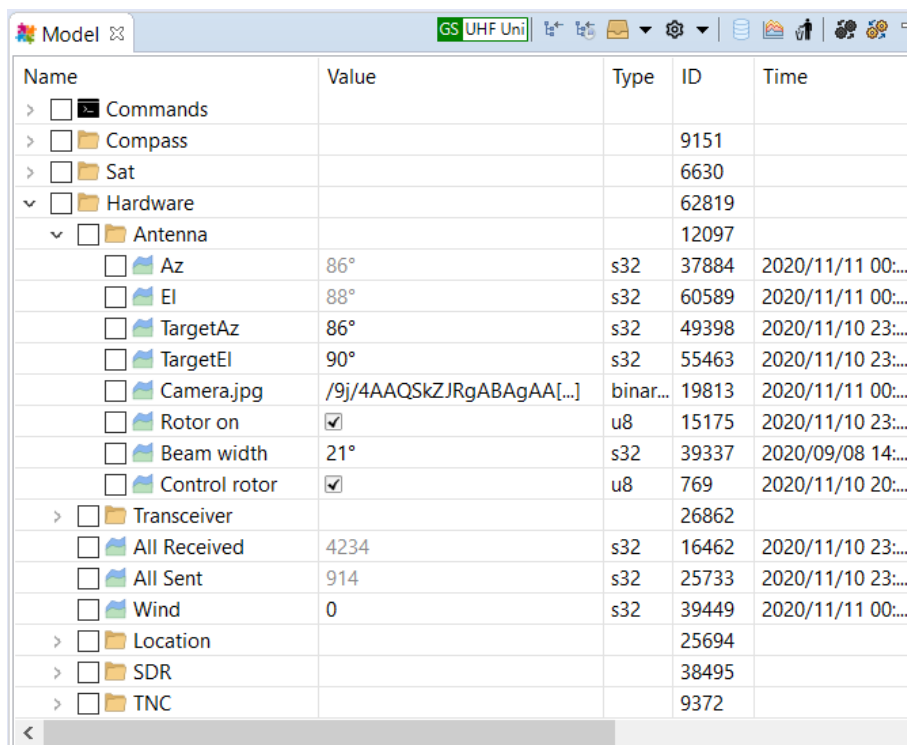
An existing script can be saved using the *Save-Packet* – it is particularly useful for smaller scripts that were uplinked using the *Add*-packet.

Type	ThreadID	Drive	Dir	File
83 (S)	1B	1B	1B	1B

Table 4.43: Tiny service: Save thread to file

### 4.3.10 Model

The Model service represents a tree of system's variables that can be remotely read or modified. The basic idea is that every node contains a local *model* – a hierarchically ordered set of name-value pairs: parameters, control values, sensor values, configurations, etc. The Model service can deliver the entire model structure on request, offers full access to values and offers subscriptions to model changes.



Name	Value	Type	ID	Time
> <input type="checkbox"/> Commands				
> <input type="checkbox"/> Compass			9151	
> <input type="checkbox"/> Sat			6630	
▼ <input type="checkbox"/> Hardware			62819	
▼ <input type="checkbox"/> Antenna			12097	
<input type="checkbox"/> Az	86°	s32	37884	2020/11/11 00:...
<input type="checkbox"/> El	88°	s32	60589	2020/11/11 00:...
<input type="checkbox"/> TargetAz	86°	s32	49398	2020/11/10 23:...
<input type="checkbox"/> TargetEl	90°	s32	55463	2020/11/10 23:...
<input type="checkbox"/> Camera.jpg	/9j/4AAQSkZJRgABAgAA[...]	binar...	19813	2020/11/11 00:...
<input type="checkbox"/> Rotor on	<input checked="" type="checkbox"/>	u8	15175	2020/11/10 23:...
<input type="checkbox"/> Beam width	21°	s32	39337	2020/09/08 14:...
<input type="checkbox"/> Control rotor	<input checked="" type="checkbox"/>	u8	769	2020/11/10 20:...
> <input type="checkbox"/> Transceiver			26862	
<input type="checkbox"/> All Received	4234	s32	16462	2020/11/10 23:...
<input type="checkbox"/> All Sent	914	s32	25733	2020/11/10 23:...
<input type="checkbox"/> Wind	0	s32	39449	2020/11/11 00:...
> <input type="checkbox"/> Location			25694	
> <input type="checkbox"/> SDR			38495	
> <input type="checkbox"/> TNC			9372	

Figure 4.12: Model service: example of the UHF ground station model

The advantage of the model-based approach is that most remotely cooperating tasks can be realized exclusively by performing mutual model modification – presupposed that the models are properly designed. For example, the AOCS subsystem can activate sun sensors on all panels by remotely setting the value `Sunsensor\Active=true` and subsequently read out remote `Sunsensor\Angle` values. Another example, the ground station configures during the overpass remote SDR nodes by changing their model values for modulation and frequency.

In order to reduce the communication overhead of the Model service, a model is logically separated into two parts:

- *Model structure* describes time invariant properties: model IDs, data types, value names, values ranges, critical limits, parent-child relationships and optional visualization hints
- *Data* describes current value content along with a time-stamp of the last change

Type	Short length	Length
1B		1/4B

Short Length	Description
0-12	Array of length 0-12
0xE	Array of length 16
0xF	Length is defined in next byte (or 4 Bytes for binary type)

Table 4.44: Model type+length encoding

The *Model structure* of a yet unknown node needs to be requested only once. All further model-based communication is performed by using only the model ID and actual data. All CompassNode-activated workstation software components (Compass Operations front-end, ground station, mission server, etc.) cache remote structures and persist this knowledge to the local file system. The caching mechanism also processes passing-by (forwarded) service packets to increase own knowledge. The totality of all known remote models is called *mission model* knowledge. Intermediate nodes that are always in the route of ground-space communication packets, such as the mission server, obtain over time most complete and most recent mission model knowledge. The workstation version of the Model service is implemented in such a way that the modification of a remote model copy automatically leads to the synchronization of the changed values in the copy with the corresponding target node (model owner).

## Data Types

The Model service supports several single and array data types, whereat a single value is realized as an array with `size=1`. All values are stored in *little endian* byte order.

The data type and array length are encoded with compact format shown in table 4.44. The upper 4 bits of the first byte represent the type as shown in the table 4.45. The lower 4 bits of the first byte either directly denote the number of array elements or denote, that there is an additional byte. The special handling of length 16 is due to the often occurring of arrays with length 16 (4x4 matrices).

## Packet Types

Several service packet types exist in two versions: local version and addressed version. Since a node can also hold a cached copy of some remote model, this copy can also be requested or modified. For example, the ZfT's mission server has *model caching* capabilities, i.e. over time it gains model knowledge of all nodes within the network, thus forming a *mission model*. It does so by processing detected Model service packets in the forwarded traffic. Due to the fact, that all ground-satellite packets are passing the mission server it obtains the most complete and most recent knowledge of the satellite states. The model

Code	Type	Size [B]	Description	Range from	Range to
0	void	0	Entry without a value, usually a group		
1	uint8	1	Unsigned integer	0	255
2	int8	1	Signed integer	-128	127
3	uint16	2	Unsigned integer	0	65535
4	int16	2	Signed integer	-32768	32767
5	uint32	4	Unsigned integer	0	$2^{32} - 1$
6	int32	4	Signed integer	$-2^{31}$	$2^{31} - 1$
7	uint64	8	Unsigned integer	0	$2^{64} - 1$
8	int64	8	Signed integer	$-2^{63}$	$2^{63} - 1$
9	dynamic	N	Dynamic model created on demand		
10	group	N	Byte array of all values within a group		
11	char	1	character array		
12	bin	1	Byte array		
13	address	5	Remote model root	0, 0	$2^{32} - 1$ , 255
14	float	4	Float with 6 decimal places precision	$1.2 \times 10^{-38}$	$3.4 \times 10^{38}$
15	double	8	Float with 15 decimal places precision	$2.3 \times 10^{-308}$	$1.7 \times 10^{308}$

Table 4.45: Data Types

caching drastically reduces the required communication to space systems, as many satellite model requests can be handled by the server itself.

Type	Answer	Description
1/10	11	Get value list
11		Value definition
5/50		Set value
2/20		Set without time
3/30	5/50	Get
6/60	5/50	Get to
4/40	5/50	Set and get
7/70		Set group
8/80	7/70	Get group

Table 4.46: Model service: local (left) and addressed (right) packet types

### Request Model Structure

The structure of the remote model can be requested using the *Get value list* packet. The remote system generates N answers with N being the number of model entries – see table 4.48.

Type	Type	Sys	Sub
1	10	4B	1B

Table 4.47: Model service: Get local (left) and addressed model list

### Model Entry

Model entries are transmitted as answers to *Get value list* packet. The format of an entry packet and the field description is shown in table 4.48. It provides all necessary information to allow a reconstruction of the whole model tree. In addition to the required fields in this packet, auxiliary semicolon-separated *GUI hints* can be placed between the name string and the 0-byte terminator: edit type, view type, unit, value range, limits and description. The format of GUI hints for a value with  $N$  elements is defined as follows (field descriptions are shown in table 4.49):

```

>EditType1, ...,EditTypeN;
ViewType1, ...,ViewTypeN;
Unit1, ...,UnitN;
Ranges1, ...,RangesN;
Limits1, ...,LimitsN;
Description1, ...,DescriptionN"

```



**Example: Names with GUI hints**

Numeric **Selection** value, whereat one element viewed as choice (e.g. drop down):

”Selection;;Ch;;One Two Three”

Numeric **Flags** read-only value with two elements viewed as bit-fields:

”Flags;R,R;Bit,Bit;;;some flags”

since all elements should be viewed in the same way, its enough to define it once

”Flags;R;Bit;;;some flags”

Type	LID	Entries	PUID	Name		UID	Type	Short Len	Len	Time	Data
11	2B	2B	2B	...	0	2B	1B		(1B)	6B	Len*Type

Field	Description
11	Entry list
LID	Local index of the value
Entries	Number of model values
PUID	Parent UID
Name	0-terminated value name
UID	Unique value ID
Type	
Short Len	see 4.44
Len	
Time	UNIX timestamp in ms
Data	Data bytes, $size = Len * SizeOfType$


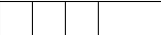
Table 4.48: Model service: Model list entry

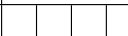

**Set**

*Set* packet is used to modify remote model values. It contains the target model ID, data and **Type+Len** of the included data (table 4.50). The latter is included for safety reasons to avoid wrong parsing if the included data does not match the type of the target value. The receiver of the set command change the last-updated timestamp of the updated value either to time defined in the packet or uses current local timestamp if no time is included or `time=0`. The remote machine does not answer to this packet – in contrast to *Set* and *Get* packets.

Hint	Value
EditType	W (read/write), R (read-only)
ViewType	Hex, Bit, Bool (check-box), Addr (Address), Time, Dur(duration), Ch (choice as defined in range)
Unit	e.g. mW, mV
Range	<i>ViewType</i> != Ch: FROM TO <i>DataType</i> is number: ChoiceStr1 ChoiceStr2 ... <i>DataType</i> is number: 0=ChoiceStr1 7=ChoiceStr2 ... <i>DataType</i> is char: ChoiceStr1 ChoiceStr2 ...
Limit	LowAlarm LowWarning HighWarning HighAlarm
Description	any text

Table 4.49: Model service: GUI hints of a list entry

Type	Sys	Sub	UID	Type	Short Len	Len	Data
2/20	4B	1B	2B			(1B)	Len*Type
				1B			

Type	Sys	Sub	UID	Type	Short Len	Len	Time	Data
5/50	4B	1B	2B			(1B)	6B	Len*Type
				1B				

Field	Description
5/50	Addressed/local model set
2/20	
Sys	Addressed system (only if <code>type=2/5</code> )
Sub	Addressed subsystem (only if <code>type=2/5</code> )
UID	Unique value ID
Type	
Short Len	see 4.44
Len	
Time	Last-update UNIX timestamp in ms
Data	Data bytes, $size = Len * SizeOfType$

Table 4.50: Model service: set addressed (2/5) or local (20/50) value with or without updating time

### Get (to)

*Get* packet is used to receive a remote model value and its children (recursive). The packet contains the desired model ID and *Max Age* in [ms]. If for example a value represents some sensor read-out, *Max Age* is used to decide either the last read-out is recent enough to deliver an answer or a new one is required.

Type	(Sys)	(Sub)	UID	Max Age
3/30	4B	1B	2B	4B

Table 4.51: Model service: get addressed (3) or local (30) value

Type	(Sys)	(Sub)	UID	Max Age	RUID
6/60	4B	1B	2B	4B	2B

Table 4.52: Model service: get addressed (6) or local (60) value

The answer is a *Set* packet with either the same model ID as in the request packet (type 3/30) or the specified RUID (remote model ID, type 6/60). The latter is useful for synchronization of two model values with the same data type but with different IDs. If the requested model contains children, then multiple *Set* responses are generated for every child entry.

### Set and Get

This packet has the same functionality as the *Set* packet. In addition a response *Set* packet is generated for the changed value.

### Set Group

Besides the packet type (7/70), the format of the *Set Group* packet is equal to *Set* packet. The defined value type is always *group* and the length represents the number of included data bytes. The data bytes represent recursively all values in the specified value/group. To avoid malfunctions, the receiver of the packet first checks whether the number of included bytes in the packet matches the number of value bytes in the corresponding model group. The values are updated recursively in the order of their appearance in the tree structure.

Type	Sys	Sub	UID	Type	Short Len	Len	Time	Data
7/70	5B	1B	2B	1B		(1B)	6B	Len*Type

Table 4.53: Model service: set addressed (7) or local (70) group

## Get Group

*Get Group* request is used to receive a compact value representation of a target value branch. The answer is a *Set Group* packet with the model ID being set to the specified RUID.

Type	(Sys)	(Sub)	UID	RUID
6/60	4B	1B	2B	2B

Table 4.54: Model service: get addressed (8) or local (80) group value

### 4.3.11 Recording and Reporting

The Automatic Recording and Reporting service (*ARR*) is an extension for Model service and is used to either record local model changes (*recording task*) or to automatically report model changes to remote nodes (*reporting task*). This service was utilized extensively in UWE-3, UWE-4 and NetSat missions to record model entries for: AOCS sensors, GPS, system states, etc. The recorded files were automatically downloaded during night passes by the auto-operator and later post-processed with Matlab.

## Clear

With a *Clear* request packet all currently existing ARR tasks are deleted. On success the node answers with a Clear Done packet.

Type	Type
112	113

Table 4.55: ARR service: *Clear* request (left) and *Clear Done* packet (right)

## List tasks

The list of the currently active service tasks is requested with the *List* packet. Currently up to 255 simultaneous tasks are supported. A separate *Entry* answer packet is generated for every existing task (see table 4.56).

## Get Task

With *Get Task* packet the information of one specific task can be requested. Despite the packet type (33), the answer is the same as for *List* requests (see table 4.56).

Type	TaskID	Flag	Settings	ConfigID	MUID	SamplePeriod	Last-Exec
33	1B	1B	1B	1B	2B	4B	4-8B

Table 4.57: ARR service: single task entry

Type 16	Type	TaskID	Flag	Settings			ConfigID	MUID	SamplePeriod	Last-Exec
				-	GF	FS				
	17	1B	1B	1B			1B	2B	4B	4-8B

Field	Description
17	List tasks
TaskID	local ID of the task
Flag	1=recording, 2=reporting task
GF	Group flag
FS	Force sampling
ConfigID	ID of the active recording/reporting config
MUID	UID of to be recorded/reported model
SamplePeriod	Sampling period of the active sampler [ms]
Last-Exec	Time of the last sampler execution, UNIX ms

Table 4.56: ARR service: request tasks (left) and task entry (right) packet

### Create Reporting Task

New reporting tasks are created with *Create Reporting* packets. On success, the remote node answers with newly created TaskID.

Type	MUID	FS	SamplePeriod	RUID	Sys	Sub
48	2B	1B	4B	2B	4B	1B

Field	Description
48	Create new reporting task
MUID	UID of the model to be reported
FS	Force sampling: force model GET Hook execution
SamplePeriod	Sampling period of the active sampler [ms]
RUID	Remote model UID to which the variable shall be reported on
Sys	System+Subsystem Address to report to
Sub	

Table 4.58: ARR service: create new reporting task

Type	TaskID
49	1B

Table 4.59: ARR service: Successfully created reporting task

## Create Recording Task

With this packet type, a new file recording task can be created. In contrast to the reporting tasks, the model updates are written to the system's local file system and can be downloaded later. During the recordings, one of two files is used for storage, i.e. as soon as the current file size exceeds *Max. file size* alternative file is re-created and selected for further recordings. This technique solves the problem of concurrent read (download) and write (recording) access.

The recording is often used in ZfT satellite missions to store data during in-orbit measurements and experiments. If successful, the remote node answers with newly created TaskID.

Type	MUID	FS	SamplePeriod	File1		File2		Drive	FileSize	Start	Duration
				File	Dir	File	Dir				
64	2B	1B	4B	1B	1B	1B	1B	1B	4B	8B	4B

Table 4.60: ARR service: create new recording task

Field	Description
64	Create new recording task
File1	Primary File
File2	Secondary File
Drive	Local file drive number
FileSize	Max. file size
Start	Start time as UNIX timestamp in ms
Duration	Recording duration in ms

Type	TaskID
65	1B

Table 4.61: ARR service: Successfully created recording task

## Task Deletion

Any existing task can either be deleted using its *TaskID* or using the ID of the model, to which the task is targeted. In latter case the value of the *Flag* field is used to define which task types should be deleted:

- 1: delete only recording tasks
- 2: delete only reporting tasks
- 3: delete all tasks

On success, the answer contains either the removed TaskID or the model UID of the removed task.

Type	TaskID	Type	TaskID
80	1B	81	1B

Table 4.62: ARR service: delete Task by its TaskID (left) and answer on success (right)

Type	Flag	MUID	Type	MUID
96	1B	2B	97	2B

Table 4.63: ARR service: delete Task by the model ID (left) and answer on success (right)

### 4.3.12 Database

The Database service is accessed to receive desired Compass traffic from a database-activated node. In UWE and NetSat missions the entire ground-space traffic is stored by the ground station and mission server nodes. The Compass Operations front-end provides a dialog (figure 4.13) to receive Compass packets from remote database-activated nodes. The format of the request and the corresponding answer is shown in table 4.64. Retrieved history packets are handled *passively* by the Compass implementation, that is:

- forwarding is deactivated
- no answer packets are generated
- services treat the packets differently (commands are not executed, etc.)

### 4.3.13 Registry

The Registry service provides synchronization capabilities for *key-value mappings*. It is used to store textual representation for different ID types: addresses, service names, command names, tiny functions etc. Therefore, it is similar to the functionality of a DNS (Domain Name Service) but is defined in more generic way.

#### **Example:**

New operator wants to access an existing mission network using Compass Operations front-end. To do so, some non-existing Compass address must be specified for the local node: 13:122. As soon as the node is connected to the mission network, every other operator would see 13:122 appearing in the nodes view. To improve the recognizability, the new operator decides to enter a new *address-text* mapping in the local address map: *13:122=Operator:Dombrowski*. Now, the local Registry service will automatically synchronize the new value with the mission server. After some time the Compass Operations front-end of other operators will also synchronize maps and receive the new address mapping – resulting in a more understandable representation of the 13:122 node.

Type	Source		Target		Service	Time	
	Sys	Sub	Sys	Sub		from	until
1	4B	1B	4B	1B	2B	8B	8B

Type	Source		Target		Service	Time		Amount	Packet 1			...
	Sys	Sub	Sys	Sub		from	until		DB time	Size	bytes	
2	4B	1B	4B	1B	2B	8B	8B	4B	8B	4B	NB	

Field	Description
Sender	Desired packet source or 0 for any
Receiver	Desired packet target or 0 for any
Service	Desired service ID or 0 for any
Time from	Lower age limit, or 0 for no limit. Format yyyy-MM-dd.HH:mm
Time until	Upper age limit, or 0 for no limit
Amount	Number of contained packets
DB time	Time at which the packet was stored in the database
Size	Packet size
bytes	Raw packet bytes

Table 4.64: Database service: request and answer packet

**Retrieve packets from remote database**

Please enter required fields

DB Node: ZfT:MS NetSat

From: NetSat-1:0

To: GS:0

API: Log

Hours:

Start date: 11.11.2020

Start time: 00:16:48

End date: 11.11.2020

End time: 01:16:48

OK Cancel

Figure 4.13: Database service: loading remote history packets with Compass Operations front-end



Every map has a MapID, which is unique (with respect to its type) in the entire Compass network. The Registry service is able to synchronize local maps with any remote Compass node with Registry support. The synchronisation follows the *master-slave scheme*, e.g. values on common nodes (mission server, compass gateway) have higher priority than the values on operators nodes.

Type	Type	MapID	Type	MapID	Key Len N	Key
0	1	4B	2	4B	1B	NB

Table 4.65: Registry service: list all, list map and list value

Type	MapID	Type	MapID	Key Len N	Key	Type	MapID	Key Len N	Key	Len M	Value
3	4B	4	4B	1B	NB	4	4B	1B	NB	2B	MB

Table 4.66: Registry service: clear map, delete value and create/update value

Currently map IDs [0-20] are reserved for default maps shown in table 4.67.

ID	Key	Value	Description
0	MapID	Name;Key0,...,KeyN;Col0,...,ColM;Target	Names of all maps are listed here
1	ApiID	Name;Description;Color(;ID;...)	All known Services
2	SysID(;SubID)	SysID;SubID;Name;Info;Color(;ID;...)	All known Compass nodes
3	SysID:SubID	Cmd0;...;CmdN	List of all known Commands of a node
4	CmdName	Arg0,...,Arg1;Help	Command usage help
6	SysID:SubID	FncID0;...;FncIDN	List of known tiny functions of a node
7	FncID	FncName;Arg0(type),...;ArgN(type);Help	Function usage help
8	FncName	JavaScript	Tiny function simulator
9	SysID:SubID	APIID0;...;APIIDN	List of known APIs of a node
...			
20		reserved	reserved

Table 4.67: Registry service: reserved table IDs

Registry service is currently only available in the CompassNode implementation, as there was no need to provide it on embedded devices (Compass OS). In the current mission network configuration, the mission server and the main Compass gateway are configured as *master* and all remaining nodes (Compass Operations nodes, simulations, etc.) as slaves. The registry update is performed automatically during the connection to the mission network and then repeated every 10 minutes.

### 4.3.14 Tunnel

The Tunnel service can be used to establish user-defined protocol connections between two Compass nodes. Some usage examples are:

- Make a non-Compass satellite accessible in to the Compass mission network.
- Forward packet traffic of a node to one or multiple nodes for monitoring purposes.

#	ID	APIID	Name	Description	Image	Color
0	1	0	Network	Network API		Blue
1	2	1	Echo	Simple Packe		Purple
2	3	2	Command	Command Int		Red
3	4	3	Uplink	File Uplink		Green
4	5	4	Downlink	File Downlink		Cyan
5	6	5	Log	Log API		Dark Green
6	7	6	Debug	Debug API (d		Dark Green
7	8	7	Variables	Variables API		Blue
8	9	8	EUnit	Unit Tests		Blue
9	10	9	GMS	Goal Manage		Purple
10	11	10	GLS	Generic Log S		Purple
11	12	11	NFS	Network File :		Pink
12	16	12	Fire	Fire API		Red

Figure 4.14: Registry service: example of the local Service map (Compass Operations front-end)

- Low-level access to remote hardware channels, e.g. tunneling of raw I2C messages from a sensor to the operator for debugging purposes.

At ZfT, Tunnel service is used between ground nodes for several purposes. So, during the satellite (auto-)operations all non-involved operators can see the entire traffic in their Compass Operations front-end. Without a tunnel this would only be possible for nodes that are located in the route between the satellite and the current (auto-)operator. The service is also used by multiple Software Defined Radios (SDR), which are implemented as separate nodes and act as backup receivers during the satellite overpasses. They use Tunnel service to inject all demodulated RF traffic to a predefined ground station for further higher-protocol decoding.

Type	Tunnel#	Type Len M	Type	Len N	Bytes
1	2B	2B	MB	2B	NB

Table 4.68: Tunnel service packet

## 4.4 Channels

This section describes the handling of stream-based channels and channels used in UWE-3, UWE-4 and NetSat systems.

### 4.4.1 Generic Byte Stream Channels

In general every protocol provides either stream (e.g. serial interface, TCP) or packet/message oriented (e.g. I2C, UDP, CSP, Compass) communication. If a packet-oriented protocol is located on top of a stream-oriented one, several problems occur during the conversion of stream bytes to packets/messages:

1. *Packet detection*, i.e. find the start and end of a packet within the stream

2. *Re-Synchronization* of packet detection on

- (a) Packet loss
- (b) Packet corruption due to *bit flips*
- (c) Packet corruption due to *bit loss*

Three basic techniques can be defined for detection of a packet start and end in a stream – see table 4.69. The last strategy is the simplest one, as it does not require a second channel or advanced control in the physical layer (e.g. transmitter hardware). The *Serial Line Internet Protocol* (SLIP) is a good candidate for encapsulation, as it has been utilized for TCP/IP protocol via serial channels and is also widely used in extended *KISS* format for communication with *Terminal Node Controllers* (TNC).

Strategy	Limitations
Using a second sync-channel	Complex and not applicable to all streams
Timing: longer transmission delay between two bytes is equal to packet start/end	Requires access to the physical layer and timing knowledge of the neighbor nodes
Use special separator byte	Packet bytes must be modified to avoid the occurrence of the separator byte

Table 4.69: Packet detection strategies

With SLIP a packet is initially transformed into a byte array, then all occurrences of END and ESC are converted to ESC ESC\_END or ESC ESC\_ESC respectively. Eventually, the end (and optionally the start) of the byte array is denoted with END byte. On the other end of the line the process is performed in reverse.

SLIP byte	Meaning			
0xC0	END			
0xDB	ESC			
0xDC	ESC_END			
0xDD	ESC_ESC			

		Byte	Encoding	
		0xC0	0xDB	0xDC
		0xDB	0xDB	0xDD

Table 4.70: SLIP Protocol (left) and encoding rules (right)

During the implementation of different channels types in CompassNode and Compass OS, the decision to use SLIP was made when:

- the channel is *stream-oriented*: serial connection, TCP
- the channel is *packet-oriented* and
  - a channel packet may be *too small* for common Compass packet sizes: *I2C*, *CAN*
  - a channel packet can transport multiple Compass packets: *UDP*

A *SLIP byte stream* is a stream that contains packets being encoded using SLIP. The bytes of the SLIP stream can also be transmitted as payload of another packets, thus allowing large Compass packets being transmitted via packet-oriented channels with much smaller packet sizes (e.g. *AX.25* or *Cubesat protocol*).

#### 4.4.2 TCP or UDP

Since TCP is stream oriented and UDP can potentially carry multiple Compass packets, both implemented channels in CompassNode and Compass OS utilize SLIP for transmission.

#### 4.4.3 I2C

I2C channels are widely used in embedded systems for communication with sensors and between microcontrollers. It can handle up to 127 nodes in 7-bit address mode (default) and is typically set-up in *master-slave* mode. Only master nodes can actively begin an I2C transaction, whereas only slave nodes can accept incoming transactions. Therefore the I2C standard defines addresses only for slave nodes.

A classic I2C network consists of one master and multiple slaves – for example a MCU and multiple sensors. There is also a concept of multi-master systems that enables multiple masters to be in the same I2C network. Depending on the implementation, master systems have to negotiate the permission on send (e.g. via separate wires). The I2C channel implementation in the Compass OS follows the *slave-only* approach. That is, every node is slave and every node can become a master. Since the I2C standard was specified with a focus on single-master networks, there is no I2C standardized way for a slave to detect the address of the master node. This addressing issue is solved by Compass devices in the I2C message payload. Since every I2C enabled Compass node is master *and* slave, they all possess an I2C address.

On very memory-limited devices the I2C buffer can be too small to carry one full mission-common Compass packet. Therefore, multiple transactions are performed to transmit a Compass packet. To do so it is first converted to SLIP bytes and then transmitted segment-wise in separate transactions.

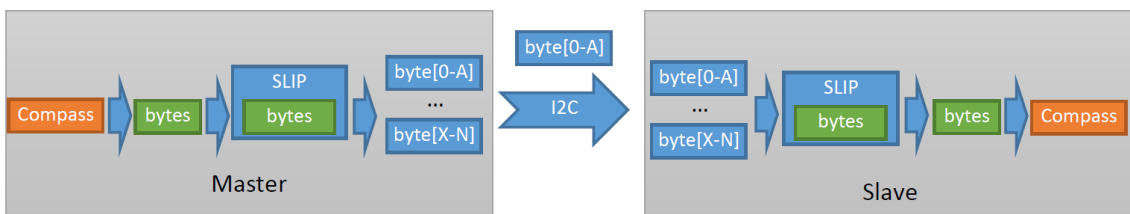


Figure 4.15: Advanced I2C communication

## Standard Communication

In the default transmission mode Compass packet transmission via I2C is performed asynchronously. That is, answers from Compass services are transmitted in separate I2C master transactions. The aim of this approach is to release the I2C bus as fast as possible, thus improving the entire network reactivity. For *exceptional* situations the *urgent transmission mode* is used.

In Compass OS the master communication is conducted by the I2C channel implementation as follows:

1. Convert Compass packet to SLIP bytes
2. Begin transaction with the target node
3. Put own I2C address and SLIPed bytes into the I2C message payload.
4. Read-in the answer of the remote I2C channel handler in the same I2C transaction and
  - If `Received=0`. The packet was rejected, as the receivers waits for another master.
    - Wait  $(200 + RND)ms$  with RND being a random number in the range [0-64].
    - Restart transmission.
    - Cancel after 3 tries.
  - If `1<Received<Transmitted`. The receiver was not able to store all bytes.
    - Wait  $100ms$ .
    - Start new transaction with non-received bytes.
    - Cancel transmission if no additional data bytes could be transmitted during the last 3 *direct subsequent tries* (reset try counter if at least 1 byte received).
  - If `Received=Transmitted` and `Response Len > 0`. Urgent answer.
    - Read in answer bytes.
    - Check if a full packet is included (ends with `SLIP_END`).
    - Add received Compass packet into the receive buffer.

I2C Address	Data
2B	N bytes

Table 4.71: I2C Master Packet

The slave communication procedure performs the following:

1. Read-in masters address from the I2C message.
2. Reject the packet (return an answer with `Received=0`), if the channel waits for another master.
3. Copy data bytes from the I2C message into the buffer.

4. Check if the payload ends with *SLIP\_END*.
  - If *false*, consider the current transmission as not finished. That is
    - Answer *in the same transaction* with the number of received bytes. (address length + data length) and **Response Len** = 0x0000 (see Table 4.72)
    - Wait until the next packet with remaining contents from the *same sender*.
    - Reject other masters during this time.
    - Cancel waiting after a specified period of time (e.g. 500ms).
  - If *true*, check if the Urgent header flag is set in the received Compass packet. If so
    - Immediately pass the packet to the local Compass packet handler
    - Convert Compass service answer packet to SLIP bytes.
    - Answer *in the same transaction* with the number of received bytes (address length + data length), **Response Len** and response data (see Table 4.73).
  - If *true*, and no Urgent flag
    - Answer *in the same transaction* with the number of received bytes (address length + data length) and **Response Len** = 0x0000 (see Table 4.72).

Received	Response Len
2B	0x0000 (2B)

Table 4.72: I2C Slave Response

### Urgent Communication

As already stated, the urgent communication mode is only used in *exceptional* situations. In this mode answers from Compass services are transferred in the same I2C transaction, i.e. the bus remains *stalled* during the packet handling, thus making the reliability of the entire satellite bus being depended on service and user code.

Received	Response Len N	Data
2B	2B	N bytes

Table 4.73: I2C Slave Urgent Response

## 5 | Space Segment

In Chapter 4 the definition of all services was shown that were used to establish all required functionalities in ground and space systems during the UWE and NetSat missions. Even though all space and ground nodes appear as uniform nodes in the mission network, they are implemented using entirely different hardware. Ground systems are rather based on high-powered machines, for which higher languages and operating systems are available. Many satellite subsystems are running 24/7 and are restricted by power, space, heat dissipation, and communication constraints. Due to these restrictions space systems offer lesser high-level functionalities compared to the ground systems. They, for instance, do not offer database support, model variables with large data blocks or services that are not feasible in space.

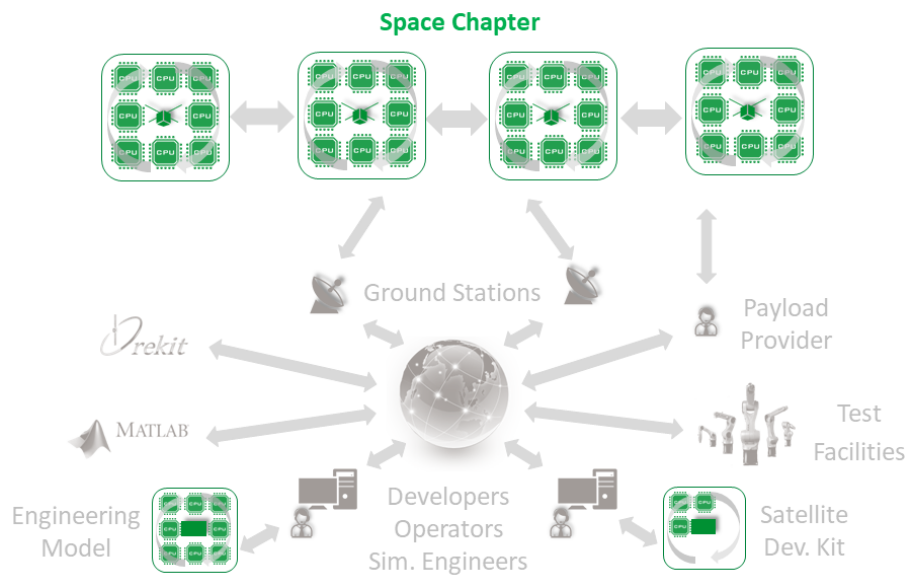


Figure 5.1: Coverage of the space segment chapter

The Compass middleware (protocol, services and execution environment) was implemented in Java as *CompassNode* and in C as *Compass OS*. The former is used for ground systems, as it allows much faster implementation of new services, straightforward extension of GUI elements and is runnable on any Java-enabled platform (Windows, Linux, MacOS,

Raspberry PI etc.). The Compass OS implementation is designed in such a way, that it is compilable with any GCC compiler, and is therefore not only runnable on embedded microcontrollers but also on workstations – presupposed there exist an implementation of the Compass OS’ Hardware Abstraction Layer (HAL). During this thesis, HAL was implemented for 16 bit MSP430, 32 bit Atmel ARM microcontrollers as well as for Windows and Linux based systems. The runability of the embedded code on a workstation enables *in-the-loop* testing as proposed in section 3.1.6.

Details of the low-level implementation are not shown in this chapter, as it would go far beyond the scope of this document. Instead, it will be shown how Compass and services were stitched together to enable the required functionality (see Requirements section and Approach chapter) and how the implementations were made runnable on multiple platforms. Thereafter, a description will follow of how dynamic code invocation was enabled on virtually any relevant microcontroller with the new *Tiny* scripting language. All examples are based on the NetSat mission.

## 5.1 Hardware environment

Since UWE-3 the ZfT’s and Universities team has a module satellite platform with a standardized *UNISEC satellite bus* [BS17]. All subsequent missions (UWE-4, NetSat, TOM, QUBE, CloudCT etc.) are based on the UNISEC standard and share many of the (updated) subsystems from the UWE missions.

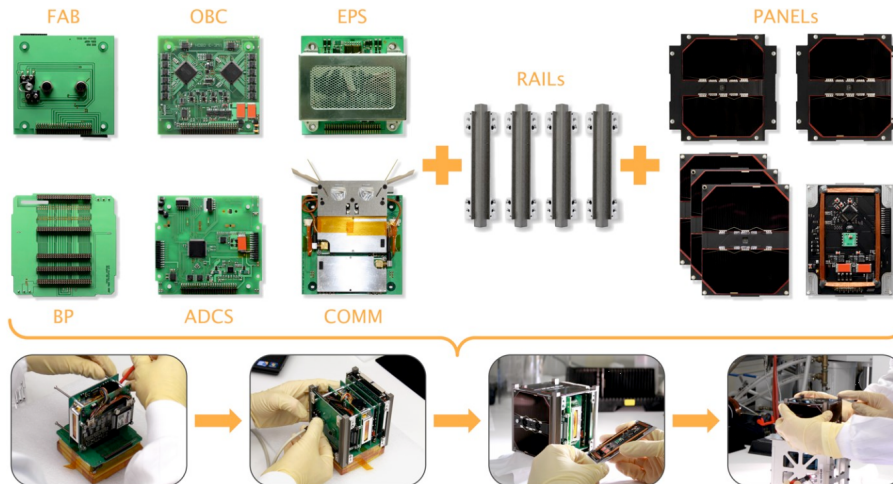


Figure 5.2: Modular pico-satellite bus (image source: [BS17])

All ZfT’s missions can be split into two categories: 1U (UWEs) and 3U UNISEC-enabled satellites (NetSat, QUBE, TOM). For all missions a variety of subsystems was designed, which are selected depending on the specific mission requirements. Some of the subsystems, such as radios, were purchased off-the-shelf.



Subsystem	Hardware	Amount	$\mu\text{C}$
On Board Computer (OBC)	16 bit MSP430	1	2
Attitude Determination And Control (ADCS)	16 bit MSP430		
Attitude and Orbit Control (AOCS)	32 bit AtmelSAM	1	2
Power Processing Unit, NanoFEEP (PPU)	16 bit MSP430		
Propulsion Control Unit, Enpulsion IFM	16 bit MSP430	1	2
Electronic Power System v1 (UWE)			
Electronic Power System v2 (3U EPS)		3	
1U Panel	32 bit AtmelSAM	1	1
3U Panel	32 bit AtmelSAM	4	4
Communication Module 1 (Lithium 1)			
Communication Module 2 (Gomspace AX100)		2	
Computing Module	Raspberry PI 0	1	1
Computing Module supervisor	16 bit MSP430	1	1

Table 5.1: Currently developed satellite subsystems and their amount in *one* NetSat satellite.  $\mu\text{C}$  shows total number of Compass OS enabled microcontrollers on one subsystem

As can be seen in the table 5.1, one single NetSat satellite contains 13 programmable microcontrollers. Since all of them are running Compass OS, the NetSat formation mission with 4 satellites forms a space network with 52 nodes. Since UWE-3 and UWE-4 satellites are also operated in the same mission super network, over 60 discretely accessible Compass nodes are currently in space. Every subsystem is embodied in a separate PCB board with a UNISEC connector and a *hot-swap* controller, which is approached by the OBC to power on or off the subsystem. One exception to that is the OBC itself – since it must be active 24/7, it does not have a hot-swap.

### 5.1.1 Microcontrollers

*In the scope of this work* there is a separation between three microcontroller types:

- *Low-power Node*: 16 bit microcontrollers (e.g. MSP430)
- *Mid-power Node*: 32 bit microcontrollers (e.g. AtmelSAM S70)
- *High-level Node*: computation modules with high-level operating system (e.g. Raspberry Pi) and all ground systems.

Since the approach is to enable advanced Compass functionality on every single node in the mission network, the software implementation had to be performed in such a way as to enable its runability on both: nodes with only 2 kB of RAM and high-level systems.

### 5.1.2 Payload

Depending on the mission goals, a payload of one mission can be a part of the satellite's bus system of another mission. For example the ADCS system was seen as a payload in the UWE-3 mission. Since then, its updated version is viewed as a part of the satellite's

Style	Arch	Speed	RAM	ROM
Low-power	16 bit RISC	16 MHz	2-16 kB	32-256 kB
Mid-power	32 bit ARM	300 MHz	384 kB	2 MB
High-level	32/64 bit ARM	1+ GHz	512-2048 MB	64+ GB

Table 5.2: Different MCU types

bus in the ZfT’s formation missions, with new payload being a to-be-tested component: imager, laser communication and so forth.

Furthermore, there is a distinction between two payload types: *Compass-enabled* and *custom payload*. With respect to the communication, accessibility and operability a Compass-enabled payload does not differ from any other subsystem, i.e. in ideal case any Compass-activated system can be inserted into the satellite without any change in the remaining subsystems. Custom non-Compass subsystems cannot be directly accessed from ground, instead some other Compass-enabled subsystem needs to care about the communication with the payload. Depending on the mission, the payload may also have own communication links that are operated separately by the payload provider – as is the case in the QUBE mission[Hab+18].

## 5.2 Communication

The satellite communication links can be divided into three domains: intra-satellite, inter-satellite (space-space) and space-ground, whereat every domain has its limitations and difficulties.

### 5.2.1 Satellite bus

In NetSat, all subsystems can communicate with each other via two redundant I2C interfaces. UWE-1, UWE-2 and UWE-3 satellites followed a static master-slave communication scheme, i.e. only the OBC could act as a bus master. Thus, all other subsystems could only communicate on demand when asked by the master. As a consequence, the satellites were represented by a single OBC subsystem. Moreover, since other subsystems (ADCS, active panels) could not be directly accessed by the operator, the functionality of those subsystems had to be reflected in the OBC’s code, thus unnecessarily bloating its code and software complexity.

Starting with Compass, an *any-master* approach was introduced in all post UWE-3 missions: initially all subsystems act as slaves on the I2C bus but every subsystem can temporarily become a master to approach any other one. Every subsystem became a full-fledged mission network node. Since the I2C standard does not define addresses for master devices, the communication scheme has to be extended as described in the section 4.4.

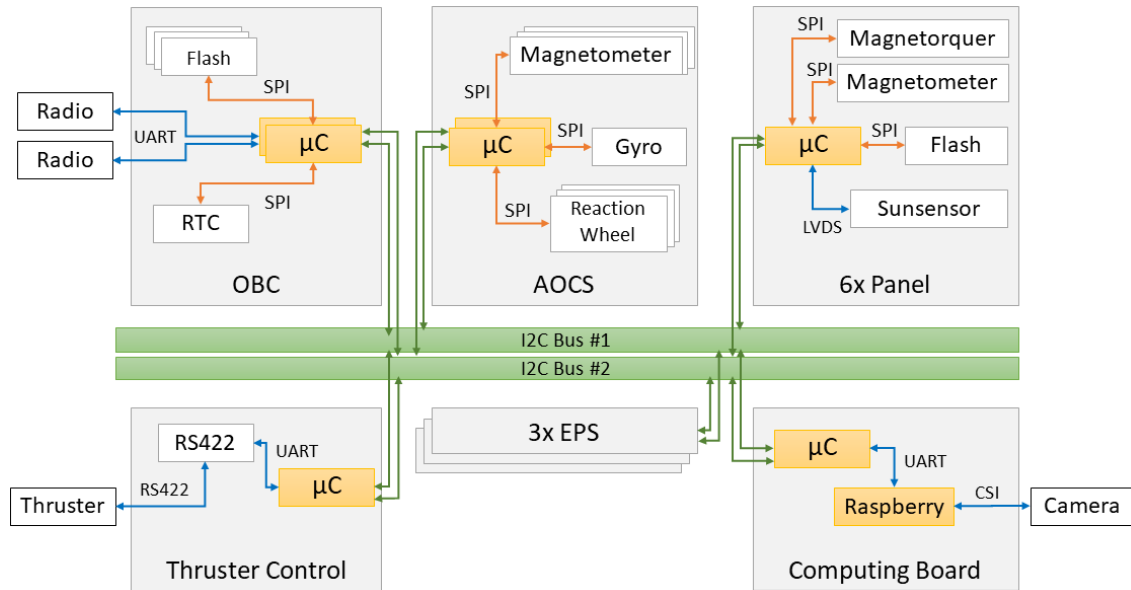


Figure 5.3: Subsystem configuration of one NetSat satellite. Yellow boxes denote Compass OS enabled components

## 5.2.2 Space-Ground

Beginning with UWE-3, half-duplex communication space-ground links were introduced, with uplink *and* downlink being operated in the UHF band (435 MHz). The UWE satellites were equipped with *AstroDev Lithium* and the NetSat satellites with *Gomspace AX100* transceivers. Unfortunately, the Gomspace transceivers compel usage of *Cubesat Protocol (CSP)* on Layer 2, which is suitable for smaller networks but could not support large dynamic networks as provided by Compass. Therefore, on all space-ground hops Compass communication needs to be enveloped in CSP, which fortunately does not introduce too much overhead.

### UHF

From experience with UWE satellites, the UHF communication is very error prone. Sometimes only 5 – 15% packet success rate can be achieved. Moreover, the packet success rate was mostly asymmetric – at times the downlink rate was much higher than the uplink. Since uplink and downlink are performed on the same frequency, the communication throughput is also highly affected by the switching time of the ground relay hardware. For full-duplex communication, different radio bands are required, thus extensively increasing the complexity of the communication hardware: doubled number of transceivers, different antennas etc. These problems render classic acknowledgement-based communication (also Compass DTN) practically impossible. The solution was to use the *bulk-acknowledgment* technique in the Compass File service, which drastically improved the throughput during the file transmissions.

## Laser communication

In the QUBE mission quantum key distribution (*QKD*) on board of a single 3U satellite will be shown in 2021[Hab+18]. The mission is performed in collaboration with DLR Institute of Communication and Navigation (*DLR-IKN*), Max Planck Institute for the Science of Light (*MPL*) and Ludwig Maximilian University (*LMU*) Munich. One of the payload components is the DLR’s Optical Space Infrared Downlink System (*OSIRIS*). The first iteration the *OSIRIS* was tested on board the Flying Laptop and *BIROS* satellites. In the current iteration the system now takes only 0.3U of space and provides up to 100 MBit/s by consuming only 8 W. In QUBE mission the optical links will not carry Compass packets, instead the additional payload optical link is established directly to the Optical Ground Station (*OGS*) located in Munich. Nonetheless, the housekeeping and experiment activation will be performed via Compass links.

### 5.2.3 Inter Satellite Link

The ISL communication can be realized in different ways. The simplest approach is to use already existing hardware for space-ground communication. This approach has been selected for the NetSat mission, as the data throughput requirements are comparable low. For all other missions the space-ground transceivers will act as a backup for dedicated ISL links.

Compass-based ISL communication was successfully proven in the NetSat mission during the first weeks until the distance between the satellites became too long in November 2020. For future formation missions with higher throughput requirements, development of a small S-Band transceiver is currently in progress. Ideally, if small enough, every panel will be equipped with a separate S-Band transceiver, thus bypassing the problem of unsuitable relative orientation of the sender and receiver during the inter-satellite communication.

## 5.3 Compass OS

Before this thesis, separate software projects were implemented for every single subsystem (in UWE-3), whereat only limited library functions were shared among those projects (no hardware abstraction layer). At that time this modus operandi was maintainable, as all subsystems were enabled by the same microcontroller family. Beginning with UWE-4, some of the UWE’s subsystems were upgraded (*ADCS*, *Panels*), such that a new additional microcontroller family was introduced. Having the formation missions in mind, a maintainable way had to be elaborated for small development teams to implement software for numerous different subsystems.

One of the outputs of this thesis is the *Compass OS*, which bundles all embedded implementations to a single package. An overview of the structure is shown in figure 5.4 – where it is also compared with the Java implementation of the middleware. *Compass OS* can also be used on top of an existing embedded operating systems – in later mission *FreeRTOS* will be considered as context-switching (“multi-threading”) system for the *Compass OS*.

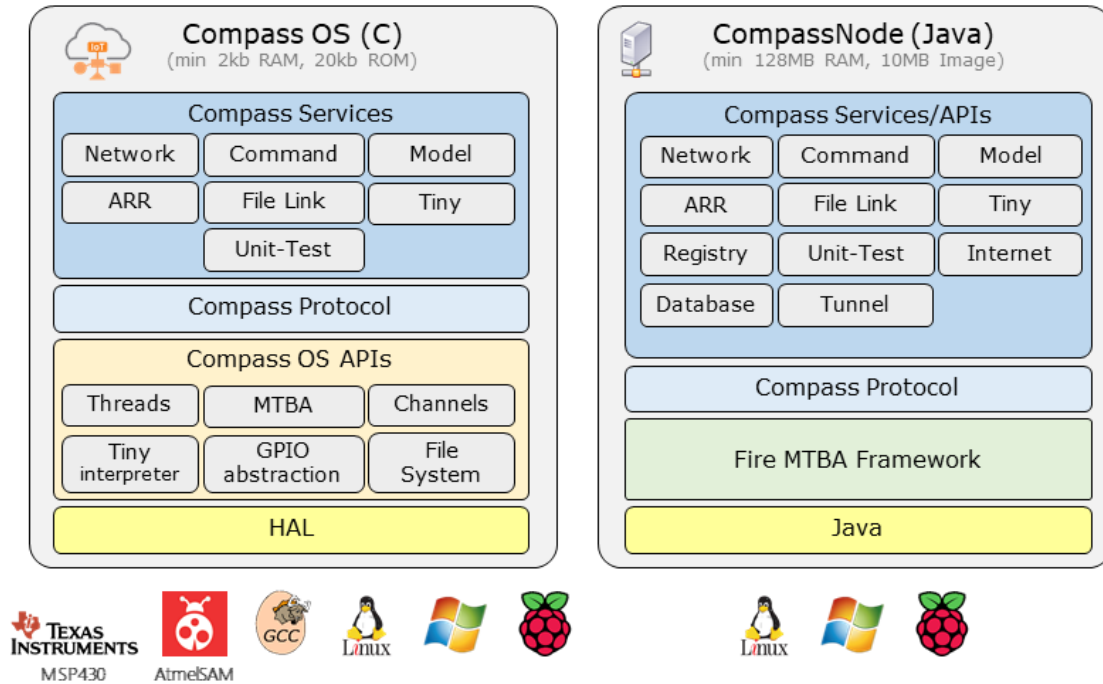


Figure 5.4: Comparison of the embedded and Java-based Compass implementations

### 5.3.1 Hardware Abstraction Layer

A common way to make embedded software runnable on multiple platforms is to detach the code from the platform-dependent low-level system calls. This mechanism is called Hardware Abstraction Layer (*HAL*), it creates a layer between the written software and the low-level system calls, such as:

- bus access: I2C, SPI, serial
- GPIO access
- time functions

In the scope of this thesis several HAL abstraction implementations were made for MSP430 and AtmelSAM families as well as for Windows and Linux based devices (GCC). Please consider figure 5.5, which shows the conversion scheme. In addition to the low-level calls, there are further abstraction blocks on higher layers. The *File System API* enables uniform file access, independent of the underlying storage technology and the utilized file system. The *Channels API* allows uniform byte-oriented communication via any supported channel (I2C, Serial, Radio etc.).

### 5.3.2 Embedded File Systems

In the current hardware design, the OBC microcontroller is connected via SPI interface to 10 non-volatile storage chips of different types (8x FRAM and 2x NAND). The FRAM

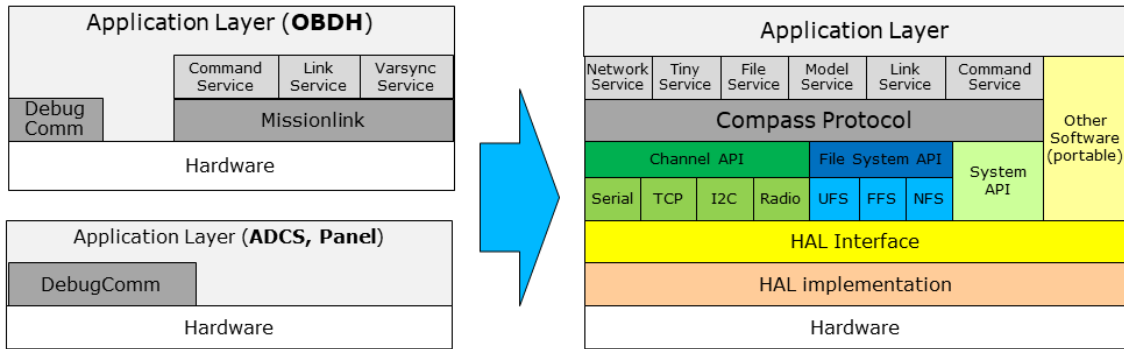


Figure 5.5: Conversion from UWE-3 software to Compass middleware

chips can be accessed byte-wise, both in read and write mode. In contrast, NAND chips offer more storage (4 Mb vs. 500 Kb per chip) but require more complex data handling (e.g. block-wise delete) and more sophisticated file systems with *Page Utilization Tables* (PUT) etc. The developed drive abstraction layer enables uniform access to different storage chips (see figure 5.6). Different file systems were designed and implemented to exploit the advantages of the specific storage technology:

- *UFS* (Uwe File System), file-oriented read and write data access for NAND chips. Has been extensively tested in-orbit on board the UWE-3, UWE-4 and NetSat satellites. UFS is best suited for growing files that are rarely (or never) deleted, such as logs or recordings.
- *FFS* (Fast File System), file-oriented read and write access for FRAM chips. Supports only static file sizes. Offers byte-wise read/write and overwrite access. Best choice for short-lived files, e.g. for buffering Compass packets (DTN), file upload location and Tiny scripts.
- *NFS* (Network File System), offers file interface to remotely located file systems. It redirects all file operations toward a remote NFS service. Is used on all subsystems without flash storage (e.g. on the AOCs), e.g. to log model value changes.

All three file system types were extensively used on UWE-4 and NetSat satellites. On UWE-3 only UFS was available on the OBC subsystem. In NetSat satellites the OBC, Panels and Computing Board have physical access to storage chips. In all other subsystems NFS drives are configured to point to different OBC's drives (figure 5.7).

### 5.3.3 Channels

A channel is, in the context of Compass, a communication pipe, which can be used for byte-oriented communication. The Compass OS Channel interface enables uniform access to any existing (and implemented) channel type. This is of particular interest for the routing functionality of the *Network service*, as its routing table is dynamically filled with

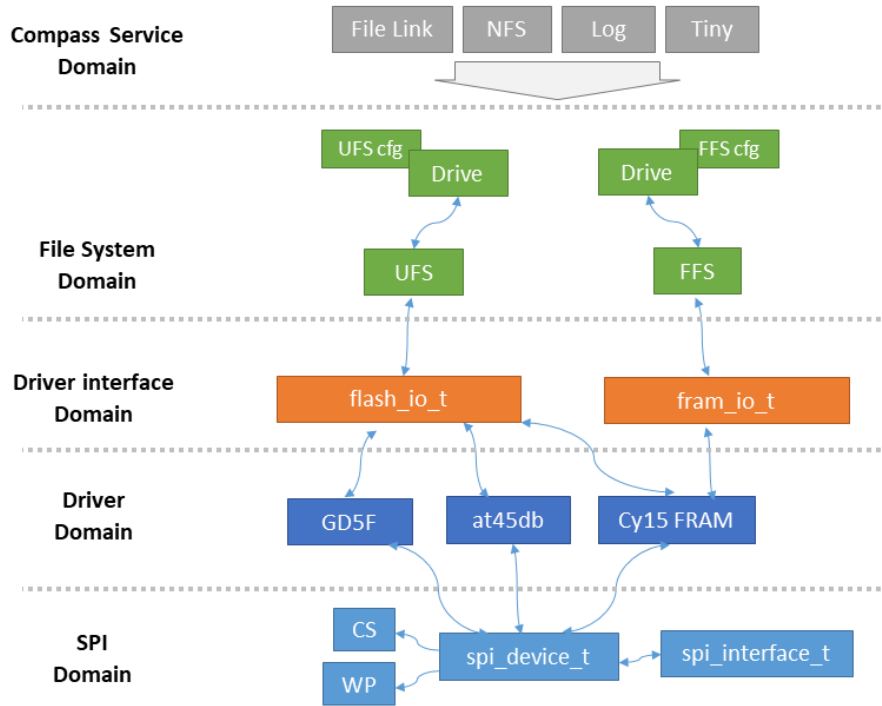


Figure 5.6: Different domains of the Compass OS File System implementation

Name	Status	Info	Size	CRC
▼ A		UFS	511kB (0%)	
▼ 1				
1	✓	1.bin	10015	
2				
3				
4				
5				
> B		UFS	511kB (0%)	
> C		UFS	511kB (0%)	
> D		UFS	511kB (0%)	
> E		FFS	507kB (37%)	
> F		UFS	511kB (0%)	
> G		UFS	511kB (0%)	
> H		UFS	511kB (0%)	
> I		UFS	4224kB (17%)	
> J		UFS	4224kB (1%)	
> K		NFS		
> L		NFS		

Figure 5.7: Available drives on the NetSat OBC subsystem visualized in the File View of the Compass Operations front-end

Name	Embedded	Windows, Linux
Serial	x	x
I2C	x	
AX100 Radio	x	
Lithium1 Radio	x	
TCP/IP	x	x

Table 5.3: Available channels in Compass OS

Compass addresses and channels via which they are accessible. That is, the Compass implementation can receive and transmit packets via any channel without knowing its actual type.

In practice all available and desired channels are registered in the Compass OS configuration file. The protocol handler checks at runtime all available channels for incoming data. If a packet has been detected on some specific channel, new routing entry is created with the sender's address, timestamp and channel identifier. All implemented channels are listed in the table 5.3.

## 5.4 Compass services

The implementation of the Compass services was performed based on the Compass definition in Chapter 4.3 and on specific requirements mentioned in section 2.1. For several reasons some of the defined services were not implemented in the Compass OS – Database, Registry and Tunnel services are only offered in the CompassNode version of the middleware (see table 5.4). The Registry service is only of particular use on systems with GUI capabilities and systems with Database service require access to a full-fledged SQL database.

The functionality of the services will not be explained in detail in the following subsections, as it has already been done in Chapter 4.3. Instead, a brief description will be shown about how the services were engaged to reach specific mission aims.

### 5.4.1 Network service

The Network service offers advanced communication capabilities, such as DTN, neighborhood detection, network beacon transmission, incoming beacon parsing and routing. Depending on to the memory restrictions of a subsystem, some of the functions must be limited. For instance the DTN store-and-forward technique can only be offered by subsystems with enough memory (RAM) or directly connected fast persistent memory (FRAM, Flash). For the dynamic routing, a subsystem requires enough memory to hold the routing entries.



Service ID	Name	Embedded	Java
0	Network	x	x
1	Echo	x	x
2	Command	x	x
3	Uplink	x	x
4	Downlink	x	x
5	Log	x	x
8	UnitTest	x	x
11	NFS	x	x
13	Tiny	x	x
14	Model	x	x
16	ARR	x	x
18	Database		x
19	Registry		x
21	Tunnel		x
22	File Link	x	x

Table 5.4: Services implemented in Compass OS and in CompassNode

**Example:**

A satellite is built up of the following systems: OBC, AOCS and 6 panels. The OBC is the only subsystem with access to the radio hardware and the only one with activated dynamic routing map. The routing table of every subsystem is pre-filled with static entries pointing to other subsystems, in which the OBC acts as a gateway. The table below shows routing entries of all subsystems during the operations, which are simultaneously performed by two operators (`GS:Alex` and `GS:Slavi`).

Subsystem	Routing-Entry	Channel	Flags
OBC	GS:Server	radio	static, gateway
	AOCS	I2C	static
	PanelX	I2C	static
	...		
	PanelZ	I2C	static
	GS:Alex	radio	dynamic
	GS:Slavi	radio	dynamic
AOCS	OBC	I2C	static, gateway
	PanelX	I2C	static
	...		
	PanelZ	I2C	static
Panel 1-6	OBC	I2C	static, gateway
	AOCS	I2C	static

Table 5.5: Routing entries example

In Compass OS the subsystem software developer must make a decision at compile time about how much memory should be reserved for the Compass routing map. A number between 10 and 20 turned out to be a sufficient value for NetSat satellites. The routing map can also be pre-filled with *static* routing entries, which always remain in the routing map (i.e. are never replaced) but still receive an inactive flag if no packets were received during some specified time period (60 seconds). By definition, the first static entry is a *gateway*, i.e. if a subsystem needs to transmit data to an unknown address (no suitable routing entry available), it passes the packet to the gateway node. Current routing table configuration of the NetSat OBC subsystems is shown in figure 5.8.

```
CompassMapEntry_t Compass_Map[COMPASS_MAP_SIZE] = (
// #####
// ##### GS #####
// #####
{.address = {SUB_BROADCAST_FLAT, SYS_GS}, .channel = &ChannelRadio, .channelHWAddress = 1, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_ZFT}, .channel = &ChannelRadio, .channelHWAddress = 1, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_OPERATOR}, .channel = &ChannelRadio, .channelHWAddress = 1, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},

// #####
// ##### INTRASAT NETWORK #####
// #####
{.address = {SUB_GENERIC_AOCS, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_AOCS},
{.address = {SUB_GENERIC_MIDPLANE, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_MIDPLANE},
{.address = {SUB_GENERIC_COMP_BOARD, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_COMPBOARD},

{.address = {SUB_GENERIC_PANEL_X, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_PANEL_X},
{.address = {SUB_GENERIC_PANEL_X, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_PANEL_X},
{.address = {SUB_GENERIC_PANEL_Y, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_PANEL_Y},
{.address = {SUB_GENERIC_PANEL_Y, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_PANEL_Y},
{.address = {SUB_GENERIC_PANEL_Z, 0}, .channel = &ChannelI2C_2, .channelHWAddress = I2C_PANEL_Z},

// #####
// ##### Radio buddies #####
// #####
{.address = {SUB_BROADCAST_FLAT, SYS_NETSAT1}, .channel = &ChannelRadio, .channelHWAddress = 4, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_NETSAT1}, .channel = &ChannelRadio, .channelHWAddress = 4, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_NETSAT2}, .channel = &ChannelRadio, .channelHWAddress = 5, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_NETSAT2}, .channel = &ChannelRadio, .channelHWAddress = 5, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_NETSAT3}, .channel = &ChannelRadio, .channelHWAddress = 6, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},
{.address = {SUB_BROADCAST_FLAT, SYS_NETSAT4}, .channel = &ChannelRadio, .channelHWAddress = 7, .flags = COMPASS_MAP_FLAGS_ALWAYS_ACTIVE},

{.address = {SUB_BROADCAST_FLAT, SYS_GS}, .channel = &ChannelSerial_3, .channelHWAddress = 0},
// ATTENTION: if not the entire map is user-defined, place this line to denote the end of the user defined area:
{.channel = 0}
);
```

Figure 5.8: Configuration of the NetSat OBC routing table with static entries

## 5.4.2 Command service

The Command service and the Model service are the most used services, as they allow straightforward implementation of the majority of functions that must be accessible from the outside. For example, in the flight software of the NetSat satellites the OBC offers 78, AOCS 36 and Panels 38 different commands – from which 34 are built-in Compass OS commands for file system operations, software module control, model modification and debugging (e.g. I2C bus scan report).

New command can be created by using the `COMMAND_DEFINE` macro (figure 5.9) at any place in any software source file. During the compilation the *Collect and Expand (CollExp)* pre-compiler checks all included sources, detects the macro and registers the newly defined command in a central command list of the Command service. From now on the command can be executed from any node in the mission network. The CollExp-enabled definition technique is also used by the Model service to collect all defined Model variables, Unit-Test

service to collect defined Tests and Tiny service to collect all defined external functions. Some examples of the existing standard commands are:

- File access: `cd`, `dir`, `format`, `cut`, `copy`, `append` (append), `merge` and `delete`.  
Example: append text to a file: `append A:\1\2 Hello World`
- File links: `dwc/upc` (cancel current down- and uplink), `dwstat/upstat` (link status)
- MCU functions: `mcustat` (get current MCU status and sw version), `tmcu` (toggle MCU on redundant subsystems, such as OBC)
- Beacon functions: `bcn` (request beacon)

```
COMMAND_DEFINE(cmd, hello, "", "Say hi" )
{
    Buffer_write(out, "Hi");
}
```

Figure 5.9: Command definition one-liner

During the satellite software development an appropriate balance had to be found between using commands and using the model, i.e. to decide either some functionality should be available via the command interface or via the node's model tree. The Command service can be accessed via the dedicated *Command View* in the Compass Operation front-end.

### 5.4.3 Model service

The Model service is extensively used in all subsystem software implementations to maximize the outcome of the MTBA approach. The service not only allows to request or set single values, but also entire value groups using extremely compact representation. During the subsystem software development it had to be continuously decided either some value should be implemented as a normal C variable or rather be stored as a model value. Similar to commands, a model variable is created with a special `GDS_DEFINE` macro. An optional *get-hook* and *set-hook* can be assigned to a model variable. In the example below a model value is created along with a group and a get hook.

```
GDS_DEFINE(100, GTD_VOID, time_grp, 0, "Time Group",
           GDS_NOHOOK, GDS_NOHOOK, TOP, 0)

GDS_DEFINE(102, GTD_UINT32, uptime, 1, "Uptime seconds since boot",
           GDS_NOHOOK, UPTIME_GETHOOK, 100, 1)

GDS_DEFINEHOOK(UPTIME_GETHOOK){
    GDS_TYP_uptime uptime = gds_get_fromHook_uptime();
    uptime.value[0] = rtc_getUptimeSeconds();
    uptime.time = gds_getCurrentTime();
    gds_set_fromHook_uptime(uptime);
}
```

Figure 5.10: Model group and value definition with a get hook

The get-hook is called every time the value is read via the Model service and is useful for values that should only be updated on demand. The set-hook is called when the value has been set via the Model service and can for example be used to control an actuator depending on the value.

Name	Value	Type	ID	Time
> Commands				
> General			1	
> Subsystems			2	
> Development			10	
<input type="checkbox"/> PowerControl Update	<input type="checkbox"/>	u8	1302	2020/10/09 0...
<input type="checkbox"/> Panel Activator	<input checked="" type="checkbox"/>	u8	1320	
<input checked="" type="checkbox"/> Antenna Update	<input checked="" type="checkbox"/>	u8	1520	
<input checked="" type="checkbox"/> RadioMng Update	<input checked="" type="checkbox"/>	u8	1633	
<input type="checkbox"/> EPS Update	<input type="checkbox"/>	u8	2005	
> Time			100	2020/10/15 1...
> Compass			200	
> OBC			1000	2020/10/02 1...
<input checked="" type="checkbox"/> Is Flight Version	<input checked="" type="checkbox"/>	u8	125	2020/10/02 1...
<input type="checkbox"/> Reset reason	BOR	u8	126	2020/10/02 1...
<input type="checkbox"/> MCU	B	u8	127	2020/10/06 1...
<input type="checkbox"/> MCU ID	8D 77 7...	u64	128	2020/10/06 1...
<input type="checkbox"/> Do reset	None	u8	129	2020/10/02 1...
<input type="checkbox"/> Watchdog enable flag	1 1	u8[2]	1005	2020/10/06 1...
<input type="checkbox"/> Chip Select Bus 1	0	u8	1006	2020/10/06 1...
<input type="checkbox"/> Chip Select Bus 2	0	u8	1007	2020/10/06 1...
> OBC's ADC12 Backup me	4594 0 ...	s16[8]	1360	2020/10/02 1...
> SEU detection			1800	2020/10/06 1...
> Beacons			1270	
> E-Unit			60000	

Name	Value	Type	ID	Time
> General			1	
> Compass			200	
> BMX			10010	
> Panel			20000	
> Torquer			20030	
<input type="checkbox"/> Timeout to switch off	800ms	u16	20032	2020/10/06
<input type="checkbox"/> PWM Clock	100000...	u32[3]	20033	
<input type="checkbox"/> PWM Cycle	100 0	u32[2]	20034	2020/10/06
<input type="checkbox"/> PWM Direction +-1 or 0	0	s8	20035	
<input type="checkbox"/> Mounting Direction +-1	1	s8	20036	
<input type="checkbox"/> PWM in [-1, +1]	-6.9273...	float	20031	2020/10/06
> Clockspeed			20060	
> Housekeeping			20070	
> Cell			20071	2020/10/07
<input type="checkbox"/> Read rate	0ms	u16	20072	2020/10/07
<input type="checkbox"/> IUP Cell	0mA 16...	s16[3]	20073	2020/11/04
<input type="checkbox"/> IUP Track A	0mA 0...	s16[3]	20074	2020/10/07
<input type="checkbox"/> IUP Track B	0mA 0...	s16[3]	20075	2020/10/07
<input type="checkbox"/> Track	A	u8	20076	2020/10/07
<input type="checkbox"/> Switch track on delta	0mV	u16	20077	2020/10/07
> GPS			20200	2020/10/07
> Data			700	2020/10/07
<input type="checkbox"/> ECEF	406390...	s32[3]	708	2020/10/07
<input type="checkbox"/> ECEV	0 0 0	s32[3]	709	2020/10/07
<input type="checkbox"/> Time	2006/0...	u64[2]	710	2020/10/07
<input type="checkbox"/> Validity	INVALID	u8	711	2020/10/07
> Configuration			20201	2020/10/07

Figure 5.11: Model examples of NetSat-2 OBC and Panel +X subsystems

In the subsystem implementations model values are created for any monitor, parameter or control value that needs to be accessible from the outside – either by an operator or another node. Therefore, numerous model values are created for sensor values (e.g. gyro, sun vector, temperatures), internal values (e.g. time, states) or control values (e.g. gyro, attitude). Within a satellite the Model service is used by subsystems for cooperating tasks. The AOCS subsystem, for instance, continuously reads sun sensor values from all panels and calculates the attitude.

In a formation the Model service is used by the master satellite to periodically receive the states of all formation partners and to distribute calculated orbit control values. Model service significantly simplifies cooperative tasks, reduces the complexity of code by decoupling the task implementation from the communication process.

The *model reporting* functionality (ARR service) can be used to create multiple model reporting task, which automatically transmit model values to the defined address for the defined period of time and frequency. So a node can avoid frequent value polling and create a remote listener for any model value in the mission network.

With *model recording* task (ARR service) one or multiple model values can be selected for storage. During the define time period every value change is then automatically stored in the defined file. Nodes without own file system, can access the Network File System

capabilities of the OBC or the AOCS.

UWE-3, UWE-4 and NetSat satellites periodically (once per minute) transmit *house-keeping beacons* with current mode, states, temperatures etc. In contrast to UWE-3 and its specific beacon format (i.e. beacon service), a Compass-enabled satellite transmits a *model group* value and can therefore be automatically parsed by the receiver's Model service. In the Compass OS the Model service supports so called *logic groups*, which allows a value to be present in multiple groups. In UWE-4 and NetSat satellites a *Beacon logic group* was defined (on the OBC) with all panel temperatures, power system parameters, uptime and status flags.

#### 5.4.4 Unit Testing

In the context of Compass, a unit test is a function that can be remotely called to test some specific functionality. A unit test can generate multiple progress messages during the execution and eventually transmits a success or fail message. Similar to commands and model values, unit tests are created with a `EUNIT_DEFINE` macro.

```
EUNIT_DEFINE(test, test_test_grp, "Test Group", 0, EUNIT_ROOT){}

EUNIT_DEFINE(test, test_test, "Success Test", 1, $test_test_grp) {
    ESUCCESS("Test", "Successful");
}

EUNIT_DEFINE(test, fail_test, "Fail Test", 1, $test_test_grp) {
    EFAIL("Test", "Failed");
}
```

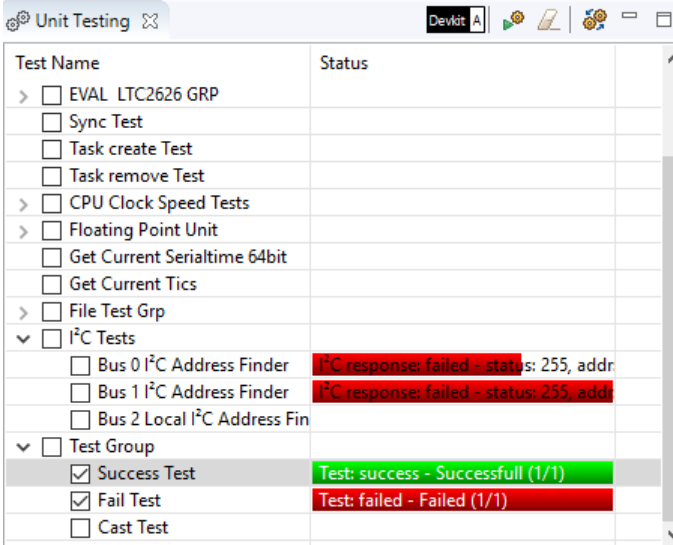
Figure 5.12: Unit test definition

Some of the basic tests are present on every subsystem: I2C communication test, file system tests (if present), clock tests etc. Unit tests are employed for multiple reasons:

- To test a recently assembled satellite. For this task a specific unit test group is defined. Test examples: I2C, spi, clock.
- To test some specific (higher-level) functionality, e.g. antenna deployment, file system healthiness, network file access.
- To identify the source of some problems (in-space unit testing).

#### 5.4.5 File service

The Downlink service and the Uplink service are only available on nodes with connected persistent flash memory (NAND, FRAM) – OBC and Panels on NetSat. The operator can perform basic file operations using the file commands, such as list files, change directory, get file information etc. Thereafter a file download or uplink can be initiated.



Test Name	Status
> <input type="checkbox"/> EVAL LTC2626 GRP	
<input type="checkbox"/> Sync Test	
<input type="checkbox"/> Task create Test	
<input type="checkbox"/> Task remove Test	
> <input type="checkbox"/> CPU Clock Speed Tests	
> <input type="checkbox"/> Floating Point Unit	
<input type="checkbox"/> Get Current Serialtime 64bit	
<input type="checkbox"/> Get Current Tics	
> <input type="checkbox"/> File Test Grp	
▼ <input type="checkbox"/> I <sup>2</sup> C Tests	
<input type="checkbox"/> Bus 0 I <sup>2</sup> C Address Finder	I <sup>2</sup> C response: failed - status: 255, addr
<input type="checkbox"/> Bus 1 I <sup>2</sup> C Address Finder	I <sup>2</sup> C response: failed - status: 255, addr
<input type="checkbox"/> Bus 2 Local I <sup>2</sup> C Address Fin	
▼ <input type="checkbox"/> Test Group	
<input checked="" type="checkbox"/> Success Test	Test: success - Successful (1/1)
<input checked="" type="checkbox"/> Fail Test	Test: failed - Failed (1/1)
<input type="checkbox"/> Cast Test	

Figure 5.13: UnitTest example

Both services engage the bulk acknowledgement technique, which solves the problem of half-duplex links with highly asymmetric packet error rates. However, this technique requires memory to hold a list of already received file chunks (chunk bitmap). Thus, the maximum file size for an uplink is dictated by the memory allocated for chunks bitmap at the receiving side. In the NetSat OBC and Panel implementations the embedded Uplink service can handle files with a maximum of 1600 chunks. Having a chunks size of 200 bytes, this results in a maximum file size of 320 kB. Larger files are split into sub-files, uplinked separately and merged in-orbit using the `merge` command. The Downlink service does not have these limitations, as the received chunk bitmap is hold by the receiver (high-powered ground system).

In UWE and NetSat missions the Uplink service was used to either uplink comparable large software update images (up to 300 kB) or smaller Tiny scripts (1-2 kB). With Downlink service logged sensor data, sun sensor images or stored model values (*ARR service*) are downloaded.

During this thesis Uplink and Downlink Views were created in the Compass Operations front-end. Both support file link by automatically requesting missing chunks during the downlink or retransmitting chunks during the uplink. In addition a *File View* was implemented to combine both services and to provide further file operations that otherwise were conducted via the Command service (see figure 5.14).

## 5.5 Dynamic Code Execution with Tiny

In contrast to machine code that is directly executed by the target machine, an interpreter is used to either directly execute human-readable scripts or to execute pre-compiled intermediate code. Moreover, large amount of the pre-compiled code can be stored in the

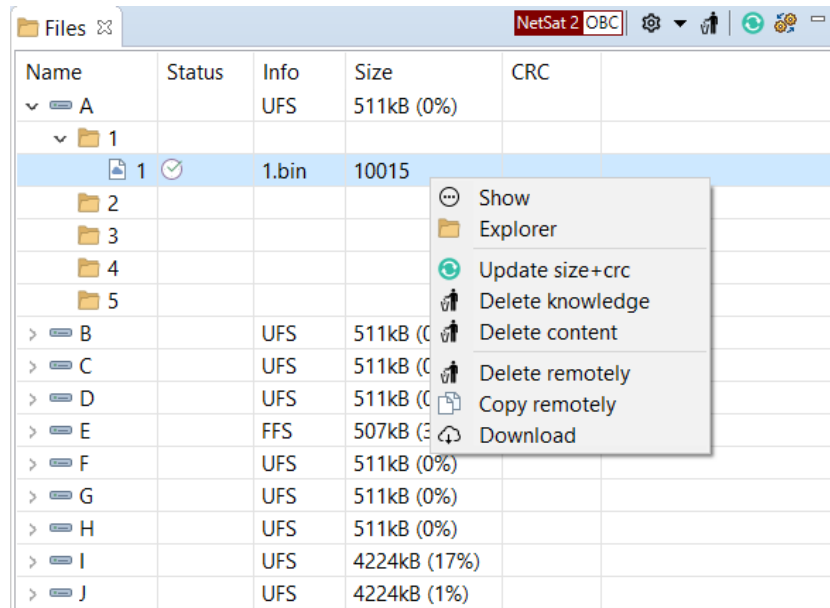


Figure 5.14: Download remote files with File View in the Compass Operations frontend

external flash memory, thereby overcoming the program memory limitations.

Due to the extensive memory and computing power limitations of the UWE and NetSat subsystems, the required interpreter must have a footprint of maximum several kilobytes and process pre-compiled intermediate code. Available script languages such as *Lua*, *Squirrel* and *Hedgehog* have been discarded either due to the size, lack of required functions or complex integrability into the existing software image. Therefore, a new *Tiny* script language was designed and implemented during this thesis. The language was improved multiple times by adding support for advanced functionalities, such as 64 bit support, advanced external functions and exception handling. *Tiny* is currently running in space on board the *UWE-3*, *UWE-4* and NetSat satellites and will also be used in future formation missions (TOM and CloudCT).

### 5.5.1 Tiny Language

*Tiny* is a script language for which following components were developed: *compiler*, *decompiler*, *interpreter*, *debugger* and a compact Java-based integrated development environment – *Tiny IDE*. The compiler is used to translate human-readable scripts into the intermediate *Tiny byte-code*. This approach is beneficial, as the byte-code is much more compact and the target platform does not need to directly parse and interpret the readable script (which would result in a much higher computation overhead).

The byte-code is platform independent and can be executed on any system for which a *Tiny* interpreter exists. Sometimes it is highly advantageous to make byte-code human readable again. For this purpose, a decompiler has been developed to enable reversing of the compilation process and reconstruct the source code. Even though the variable and

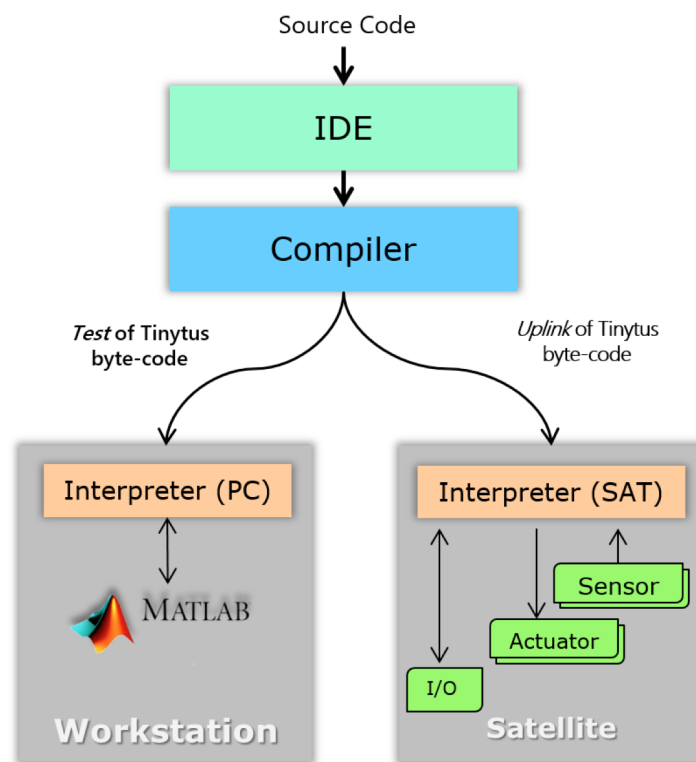


Figure 5.15: Tiny development cycle



function names cannot be restored (due to the nature of the compilation process), the generated script remains readable and offers the developer a comfortable tool to understand given byte-code.

Tiny interpreter has been successfully tested in-space on board the UWE-3, UWE-4 and NetSat satellites[DB15]. It is a part of the On Board Data Handling (OBDH) and Attitude Determination and Control (ADCS) software. It enables complex tele-commanding and continuous execution of high-level attitude control algorithms [Ban+15][Ban+16]. In 2020 Tiny was used to execute *NanoFEEP thruster* experiments on board the UWE-4 satellite. In NetSat mission, Tiny was as yet (November 2020) used to enable on-board activity scheduling and payload experiment execution.

The interpreter has a footprint of several kilobytes and extends the functionality of a low-power 16 bit microcontroller without the need of the hazardous in-space software image update. The sand-box design eliminates the threat of the potential software crashes. This concept has dramatically improved the outcome of the scientific research - since more algorithms (ADCS and orbit control) could be tested during the UWE missions. Therefore the idea of a sand-box interpreter was expanded to fulfil the requirements of the NetSat, TOM, CloudCT and further formation flying missions with respect to the distributed formation control, in-space database and operations.

The generated byte-code can be executed using the *Tiny interpreter*. Currently there exist two implementations: a GNU-C based and a Java based interpreter. The former can be used for any platform for which the software is developed in C language – for instance Atmel or TI microcontrollers. In the nominal setup the interpreter contains ~600 lines of code (without defines). The Java version supports more advanced development tools and is included in the Compass Operations front-end. The debugger, for example, is used to test the script on a workstation before uploading it to the embedded node. Usually, many external functions that are called within the script, are available only on the target platform (e.g. read sensor values, actuator feedback etc.). In order to make the written code testable on a workstation, these functions can be implemented as Matlab or JavaScript functions, which are automatically called by the workstation's interpreter/debugger via the Matlab or JavaScript interface respectively.

The compiler, decompiler and debugger are available within the Java-based Tiny IDE (see figure 5.17). As there are no external dependencies, the IDE can be started and used right away. Nonetheless, it is also integrated in the *Tiny View* of the Compass Operations front-end.

Tiny supports out-of-the-box signed/unsigned (8, 16, 32 and 64 bits) and floating point numbers, character strings and arrays. It offers various internal instructions:

- Arithmetic operations: addition, subtraction, multiplication, division, increment, decrement and modulus
- Boolean operations: equal, not-equal, larger than, smaller than
- Control structures: if, while, do, try, yield
- Variable declaration: u8, u16, u32, u64, s8, s16, s32, s64, float and their array form
- Variable access and conversion: cast, clear, memcpy, ptr
- Internal function call
- External function call with variable number of arguments

Depending on the requirements, various number of external C functions can be registered in the compiler and thereby made accessible within the Tiny code. Currently one, two and

variable argument functions are supported. The former two can be used to call standard C-library functions (e.g. arithmetic functions), the latter to implement any other custom functionality. Additionally an unregistered function can be called during runtime by using its absolute memory address pointer as a parameter for `vcallv` (no parameter, no return) or `dcalld` (float parameter, float return) instruction. However, this kind of external function access is *highly discouraged* as it cannot be guaranteed at runtime in the sand-box that the target function meets the defined conventions, thus violating the sand-box nature.

Variables			Controls		1-arged functions			2-arged functions			Var-arged functions		
name	#	instr	name	instr	name	instr	sim	name	instr	sim	name	instr	sim
A	[0]	1	-3	150	++	32		==	96	x	getTime	167 10	
B	[1]	2	-2	151	--	33		!=	97	x	getTimeS	167 11	
C	[2]	3	-1	152	len	34	x	>	98	x	wait	167 12	x
D	[3]	4	0	153	vcallv	35		<	99	x	waitBusy	167 13	
E	[4]	5	1	154	dcallv	36		+	100	x	log	167 14	
F	[5]	6	2	155				-	101	x	memfree	167 15	
G	[6]	7	3	156				*	102	x	<b>gds_get</b>	<b>167 20</b>	
H	[7]	8	if	160				/	103	x	gds_set	167 21	
I	[8]	9	if_else	161				%	104	x	gds_setAt	167 22	
J	[9]	10	try	162				set	105		gds_getRetmote	167 23	
K	[10]	11	while	163							gds_setRetmote	167 24	
L	[11]	12	do	164							send	167 30	
M	[12]	13	call	165							command	167 31	
N	[13]	14	=	168							suppressErrors	167 32	
O	[14]	15	memcpy	169							file_delete	167 40	
P	[15]	16	throw	170							file_write	167 41	
Q	[16]	17	var	172							file_read	167 42	
R	[17]	18	s8	173									
S	[18]	19	char	174									
T	[19]	20	u8	174									
U	[20]	21	s16	175									
V	[21]	22	u16	176									
W	[22]	23	s32	177									
X	[23]	24	u32	178									
Y	[24]	25	flt	179									
Z	[25]	26	err	180									
A2	[26]	27	s64	181									
B2	[27]	28	u64	182									
C2	[28]	29	dbl	183									
D2	[29]	30	\$	186									
			&	189									
			clear	192									
			cast	193									
			as	194									
			sptr	195									
			yield	254									
			function	255									

Figure 5.16: Tiny instruction set shown in the IDE's help window

### 5.5.2 Tiny IDE

The Tiny IDE is a very light-weight Java-based development environment. It offers compilation, de-compilation, execution and local debugging abilities. The Tiny IDE can either

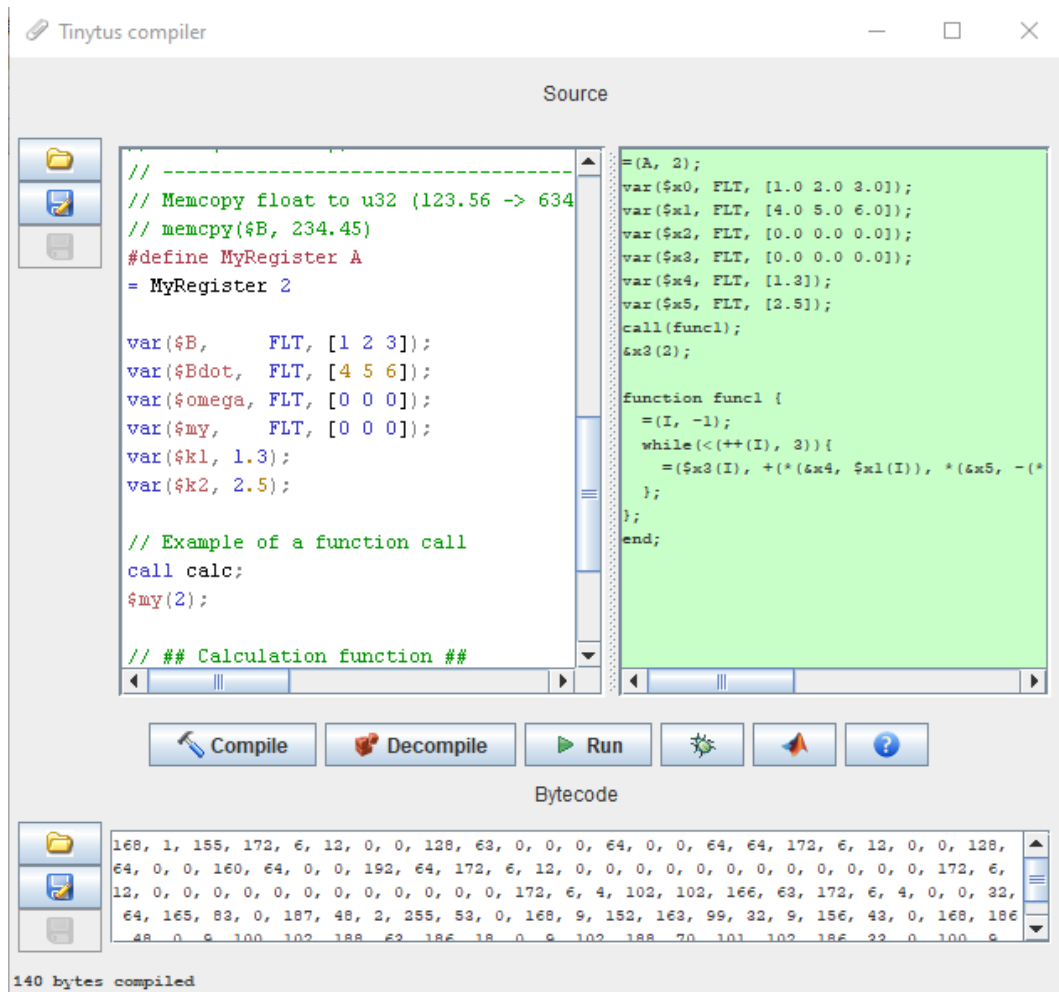


Figure 5.17: Tiny IDE with a source editor (top left), compiled byte-code (bottom) and de-compiled code (top right)

be used as a stand-alone application for creating executable Tiny byte-code or be used in the *Tiny View* of the Compass Operations front-end.

Tiny IDE supports syntax highlighting, live compilation and de-compilation during the script editing and can access existing embedded software implementations (Tiny configuration files) to identify registered non-standard functions. The interface of the Tiny IDE has been designed in such a way, as to enable its usage in other Java-based applications. This way Tiny IDE was integrated in the Operations software, which combines the functionality of the IDE with the Tiny service (see Chapter 6):

- Uplink compiled scripts to a distant node.
- Fetch remote byte-code for revision (de-compilation).
- Use Tiny service to receive a list of available external functions, which are then offered in the Tiny IDE.

### **Tiny Compiler**

The compiler is used to convert human readable scripts into the corresponding intermediate byte-code. To some extent the Tiny byte-code instructions are similar to the assembler language, whereat some instructions have higher-level characteristics allowing much smaller program foot prints.

The instructions were designed in such a way, that the byte-code can be easily created without the use of the compiler. For example, mission planning tools can generate runnable Tiny code directly – the operator does not need to write any script, instead the tasks can be created using graphical user interface (time-line, boxes etc.).

### **Tiny Decompiler**

Tiny byte-code is intrinsic, i.e. besides function and variable names, the conversion between byte-code and source-code is one-to-one. The decompiler is an assistant tool to view already existing byte-code in a human readable and more understandable form.

### **Tiny Interpreter**

This component is responsible for the execution of the byte-code on the underlying hardware and software platform. The Java-based Tiny development environment contains an interpreter, which is used to test the designed scripts. If external functions are used within the script, a corresponding emulation function can be additionally written in Java or Matlab to mimic the functionality of the target node's hardware. Additionally a cross-compilable GCC interpreter exists to execute Tiny byte-code on arbitrary hardware.

The interpreter uses a dedicated stack for every concurrent process. This technique has multiple advantages compared to the system's stack. First, all memory accesses are checked in advance and thus memory overflows are detected without compromising the stability of the whole system. Second, the interpretation process can be paused and continued later using the preserved stack state. The interpretation can either be paused automatically after a predefined number of instructions or by using the yield instruction within the script itself. The yield instruction is of particular interest if the developer needs to avoid the interruption of certain code sections. Moreover, multiple Tiny processes can be executed in parallel, i.e. each process periodically receives some predefined amount of

CPU time (number of instructions to execute) – whereat major processes receive more than the subordinated ones.

### 5.5.3 External Functions

Tiny provides numerous built-in functions for Boolean, arithmetic and trigonometric operations. However, the language was not designed to replace the entire (on-board) software implementation. Instead, up to some specific abstraction level the on-board software is implemented in a classic way. The developer needs to keep in mind, that some of the functions should be made accessible by the Tiny functions. An appropriate implementation style is to design a hardware accessing function in pure C (sensor read, actuator handling, handling hardware interfaces) and offer a more abstract interface function for Tiny scripts.

In the scope of the current missions all hardware interfaces (I2C, serial, SPI) are handled in pure C by the Compass OS. Hence, to initiate the communication (e.g. value request from a remote node) inside a Tiny script, only the target (sub)system address and the desired payload is passed to an external function that in turn decides via which hardware interface the packet must be transmitted (Compass routing).

All external functions must conform to the following signature:

```
Value functionName(VarArgs params)
```

Whereat the return Value-struct holds a data-type along with the corresponding value (*reflective* data types). The content can either be one single numeric value (of type `u8-u64`, `s8-s64`, `float`) or some exception code number (*exception* type). The *VarArgs* incoming struct is generated by the interpreter and contains a pointer to the Interpreter instance (= context) and a variable number of parameters.

All external functions are registered in the Tiny interpreter header file along with the corresponding function IDs. By convention all functions should have unique IDs throughout all systems within a project, thus making any compiled byte-code runnable on all systems – provided that all external functions called from the byte-code are present on the corresponding system. The Tiny-IDE can either be pointed to such a header file to read-in registered function or the list of the available functions are fetched via Compass using the Tiny service.

### 5.5.4 Compass bonding

The Tiny IDE can either be used in an entirely Compass-agnostic way as a standalone application or be integrated into another GUI-application to extend its functionality. The IDE is a part of the Compass Operations software and can be utilized for nodes with Tiny execution capabilities. The Tiny service offers:

- Receive list of available functions
- Direct byte-code execution (one-time execution)
- Creation and control of one or multiple Tiny threads
- Load Tiny threads from the local file system
- Store threads to the local file system
- Auto-load Tiny threads on boot-up (persistent threads)

Please approach section 4.3.9 for more detailed information about the Tiny service.

### 5.5.5 Remote Function Execution

Due to the sand-box nature of the interpreter and depending on the configuration, the execution may be done at one stroke or be interrupted multiple times to transfer the CPU resources to other tasks. Since Tiny has its own virtual stack, the execution can be interrupted periodically after the pre-set number of instructions. This enables concurrent execution of multiple scripts without the need of a multi-threading operating system (such as Rodos [BGM14], Free RTOS or Compass OS threads). After the execution is accomplished, the last interpreter value is returned – in most cases it is the value after the return-statement at the end of the script.

Tiny service provides a basis for distributed computing:

- Divide-and-Conquer, divide a complex problem into smaller ones, distribute the sub-problems (multiple Tiny scripts) and combine the (asynchronously) received results
- A low-power satellite subsystem can call a (switchable) more powerful subsystem to calculate essential data – e.g. orbit dynamics, re-plan current mission tasks or complex database operations.

```

// define System, Subsystem and API IDs
#define NetSat1  4
#define NetSat2  5
#define NetSat3  6
#define NetSat4  7
#define ADCS     3
#define API_VARIABLE_READ  6
#define ID_TEMPERATURE  54

var($systems, U8, NetSat1 NetSat2 NetSat3 NetSat4);
var($payload, U8, ID_TEMPERATURE);
var($waiting, U8, 1);
var($result_vector, FLT, [0.0 0.0 0.0 0.0]);

= A 4
while(--(A)) {
    compass_send($systems(A), ADCS, API_VARIABLE_READ, $payload, $result_vector(A), $waiting);
    while($waiting)
        yield; // release
}

// do calculations...

```

Figure 5.18: Tiny distributed execution

In the current working design the distributed computation is enabled by using the external `compass_send()` function (Figure 5.18). The function is called with a node's address and service ID as well as the desired payload. Due to the asynchronous nature of the communication, the incoming answer is detected by polling the `waiting` variable. The `yield` instruction provokes an intermission of the script interpretation and releases the CPU resources to other system tasks (non-blocking wait).

In most cases the remote computation is done by calling a service that offers some dedicated functionalities (such as variable access). However, to perform custom-made

computations a pre-compiled Tiny byte-code can be sent to the remote Tiny service with subsequent result handling (Figure 5.19).

```
#define NetSat1 4
#define NetSat2 5
#define NetSat3 6
#define OBDH 1
#define API_TINY 13

// Precompiled remote Tiny-code
var($payload, U8, [166 1 3 1 2 3 166 6 4 0 0 0 166 1 1 1 255]);
var($result, FLT, 0.0);
var($waiting, U8, 1);

compass_send(NetSat1, OBDH, API_TINY, $payload, $result, $waiting);
while($waiting)
    yield;

// Return the result
$result;
```

Figure 5.19: Tiny remote code execution





## 6 | Ground Segment

In contrast to space nodes, ground systems are much less restricted by power, space, communication throughput and computational constraints. As a consequence, higher (non-embedded) operating systems and more capable programming languages, such as Java, can be used as a basis for the Compass implementation.

Approach chapter and Protocol chapter describe the general approaches, solutions and protocol interfaces that were followed to implement a distributed and uniform infrastructure. This Chapter shows an overview of all mission-relevant ground systems, how they were combined to a uniform Compass mission network and the implementation details of the Compass Operations front-end, which not only is capable of operating satellites, but is rather used to monitor and control the entire mission network.



Figure 6.1: Coverage of the ground chapter

## 6.1 Environment

Since the software conversion to Compass, the same software configuration is used to control all satellites – UWE-3, UWE-4 and NetSat formation. That is, a *mission super network* was established, which is capable of tracking multiple satellites with different protocol configurations. From the communication point of view and in the scope of Compass, a satellite formation is a space-located Compass sub-network, consisting of multiple nodes with dynamic communication links. All formation and mission-specific tasks are based on standard services: satellite models (Model service), commands (Command service), scripts (Tiny service) and GUI components (Compass Operations front-end).

### 6.1.1 Before this thesis

After the launch of the UWE-3 satellite in November 2013, the ground segment was scarcely populated (figure 6.2). The protocol configuration is shown in figure 3.1. The satellite tracking, Doppler correction, Terminal Node Controller (TNC) handling and satellite communication were handled by a *UHF ground station Linux server*. At that time all these tasks were performed with different software modules – Linux built-in functionality (KISS ports), software from the amateur radio community and separate Java modules for other specific tasks, such as Fault Detection and Recovery component (*FDIR*).

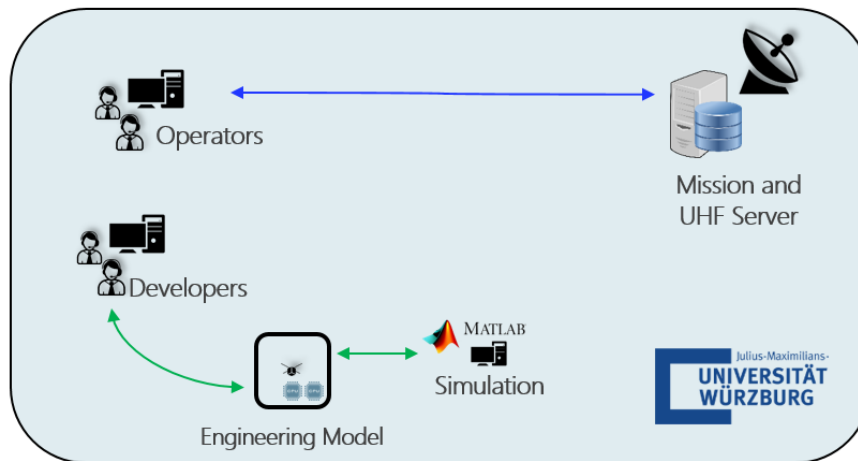


Figure 6.2: Ground systems overview before this thesis (spring 2014)

The *workstations* of the UWE team members were used for operations, development and simulation. However, the realization of the communication during the operations was different from the communication with the engineering satellite model (green and blue links in figure 6.2). In both cases a logical one-to-one (operator to satellite) network topology was established, whereat the developer was required to switch manually between the radio link for operations and serial interface for the development. Due to the one-to-one limitation it was not possible to establish communication to multiple satellite models at the same time – which at that time was not a requirement.

There existed several simulations in Matlab for ADCS algorithms, visualizations of sensor values (magnetometer, sun sensors etc.) and experiment scripts that have been used during the engineering phase to measure the performance of sensors and to calibrate sensors (sun, magnetic) and actuators (magnetorquer). During the in-orbit phase Matlab was used to post-process recorded values. The required satellite values were either injected using files that were downlinked from the satellite or by creating a direct Matlab-satellite link with special scripts (engineering model only).

### 6.1.2 During this thesis

The elaborated uniform network topology and protocol configuration are shown in figure 6.3 and 3.12 respectively. All relevant ground systems were made accessible in the Compass network. Instead of using specific communication link implementations between two counterparts, every node can now access any other node in the decentralized and dynamic network. If a new software or hardware component needs to access some remote node's values, it enters the network as a node – and can therefore also be accessed from the outside.

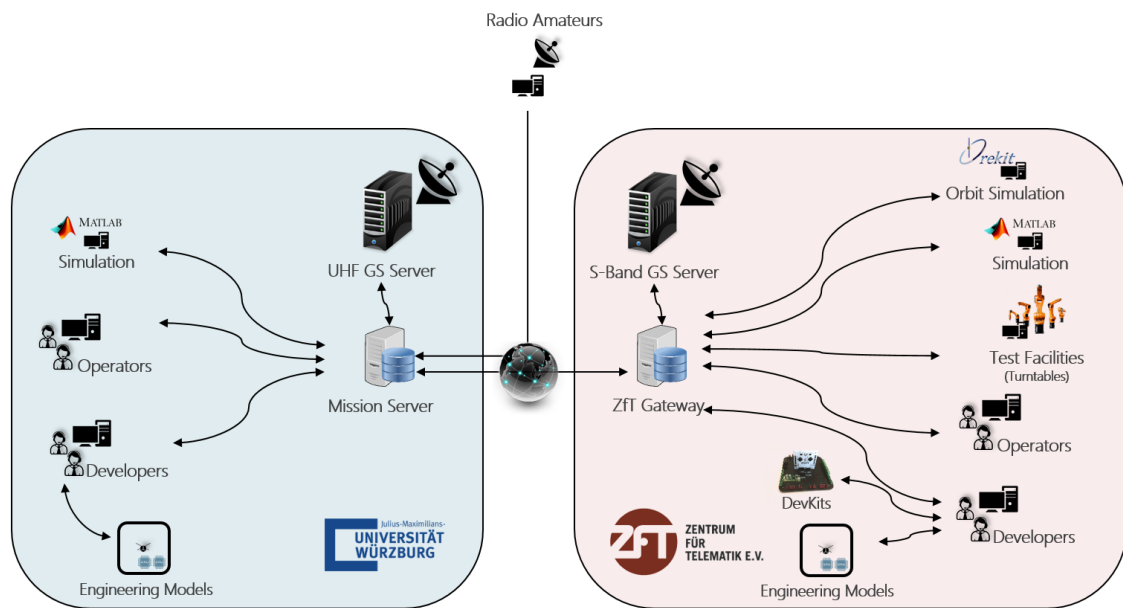


Figure 6.3: Ground systems overview at the time of writing (mid 2019). Left: University's sub-network, right: ZfT's sub-network

All available ground segment systems can be separated in following types:

- *Ground station server instance* (UHF, S-band etc.): satellite tracking, ground station hardware monitoring and control.
- *Developer's instance*: execute unit tests, run IDEs to implement software for sub-systems.

- *Compass Operations instance*: monitor and control any remote node, commanding, file transfer, create Tiny scripts, visualization of multiple satellites and creation of auto-operations schedule
- *Mission server instance*: store satellite traffic in a mission-specific database, cache model knowledge from all nodes with the entire network.
- *Matlab instance*: post-process satellite data, execute simulations and calibration algorithms.
- *Orekit instance*: perform orbit simulations, simulate maneuvers vs. propellant consumption etc.
- *Gateway instance*: used to access remote Compass sub-networks via secured channels, reduces the amount of TCP/IP channels between Compass sub-networks.
- *Test facility*: e.g. two high precision motion simulators at ZfT, which are low-level controlled by a dedicated workstation.
- *Radio amateur*: injects received satellite packets via a web service into the mission database.
- *Satellite development kit (DevKit)*: simplifies software development for single satellite subsystems and makes them available in the mission network.

In the list above the word *instance* is used intentionally to accentuate the possibility of running multiple instances on the same workstation, i.e. create multiple Compass nodes on the same machine. A team member with multiple duties can for example run one Matlab and one Operation node on the same machine. In addition, all Compass-related development tasks (e.g. unit testing, logging) can be performed using the Operations front-end instance, as it already offers GUI for all standard services.

## 6.2 Java Implementation

As described in the section 5.4, the Compass protocol was implemented in C and in Java. Most of the mentioned ground systems are capable of running Java, thus drastically simplifying the implementation and version distribution procedures. Driven by the *keep it simple and stupid* (KISS) philosophy, the entire Java Compass middleware implementation is packed in a single *CompassNode.jar*-file (10Mb size), which is runnable out-of-the-box with only Java Runtime Environment (JRE) as a prerequisite. The same *CompassNode.jar* file can be used in Matlab, Orekit and any other instances listed in the table 6.1 to enter the Compass network and make use of all available services.

In Matlab only several lines of code are required to enter a Compass network, register service listeners and to transmit packets. An example of node creation is shown in the appendix 9.6. This technique is used to connect Matlab models with live data from satellites and test facilities. For example, during the calibration of the camera-based sun-sensors of the UWE-4 satellite, a Matlab node, the turntable node and the satellite node (installed in the turntable) are connected to the mission network. All these nodes were interacting with each other, whereat the calibration procedure was delegated by a Matlab node: set the orientation of the turntable, read out the sun sensor value from the satellite and update the calibration matrix (see section 7.3 for more details).

Any Java-based application can also easily enter the Compass network with less code – see appendix 9.6. This technique is used in Java-based Orekit simulations, operations

Name	CompassNode	Compass OS
Ground station server	x	
Developer's workstation	x	x
Operations workstation	x	
Mission server	x	
Matlab instance	x	
Orekit instance	x	
Gateway instance	x	
Test facility	x	
DevKit		x
Satellite engineering model		x

Table 6.1: Java-ability of ground systems

software, small experiments and student projects, which later become part of the live-system.

Non Java-enabled systems are: DevKits, satellite engineering models and amateur radio stations. The DevKit is the implementation of the *Flat Sat*-approach, it contains an embedded microcontroller running Compass OS and acts as a *USB-to-UNISEC gateway* for all inserted satellite subsystems. An engineering model has the same software configuration as a flight model – the Compass multi-channel capability allows any flight model to become an engineering one when connected via the serial interface.

### 6.2.1 Fire Framework

The implementation of the CompassNode was performed using the Java-based *Fire Framework*, which was developed at *privateflag* and is now distributed by *Embedded Smartware*. It strictly follows the Model-Tree-Based-Architecture (MTBA) concept and provides interfaces for *modules* and *wires* [Dom15a]. A module is a software component with defined inputs and outputs, which are used to communicate with other modules via *dynamic* wires (changeable at runtime). A software solution is achieved by instantiating multiple modules for dedicated tasks, connecting them via wires and configuring the modules by changing their parameters. All instantiated modules, wires, preferences and values are then accessible in the model tree. The model tree can be partially or fully injected (and continuously synchronized) into the model tree of one or many remote Fire instances (*model ghosting*). With this technique a module from one Fire instance can be connected to a module (or multiple modules) in a remote instance, thus forming a distributed software implementation (distributed *super instance*). The entire development process can either be done entirely programmatically or by using the Fire GUI, which is able to securely access Fire instances all around the world. Fire does not distinguish design from runtime, instead the solution is composed (e.g. using the Fire GUI) at runtime and is later loaded from the automatically stored *XML* file.

Multiple module implementations are grouped to task-specific *Toolkits*, e.g. commons toolkit, space toolkit, raspberry toolkit and so forth. Once written, a module can be reused in any software project by dragging the module from the toolkit palette to the

*model drawer*. The Fire framework runtime (without toolkits) is closed source and free of charge. The toolkits can be directly purchased from the company or through the soon available *Fire toolkit store* in the Fire GUI.

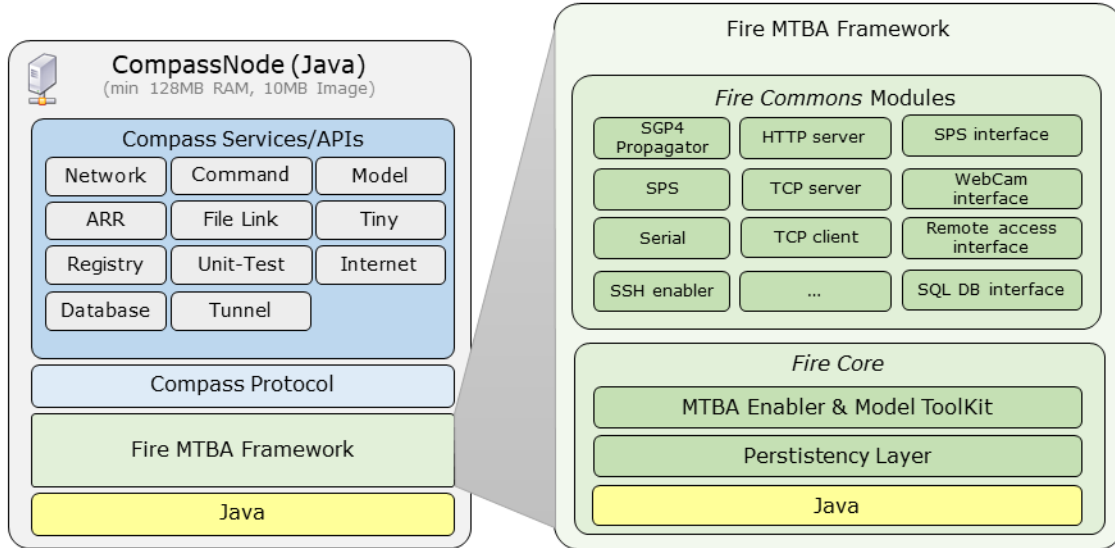


Figure 6.4: Structure of the CompassNode software

During this thesis all Java-based implementations were made on top of the Fire framework, thus enabling the re-usage of all available modules in a straightforward way: Compass protocol, services, Channels, GS hardware control, satellite tracking, auto-operations, e-mail service, web front-end and so forth.

### 6.3 Ground Station Server

Since all current ZfT's missions rely on the UHF communication, the S-Band ground station at ZfT is currently not in use. Therefore, only the UHF ground station from the University of Würzburg is described in detail. A simplified hardware architecture of the ground station is shown in figure 6.5. A more detailed information about the involved components is described in [Dom10].

The ground station software is based on the CompassNode and has following duties:

- Hold a *configuration list* for multiple satellite: name, NORAD-ID, frequency, RF modulation, Compass address, protocol chain and tracking priority.
- *Propagation*: automatically update TLEs, run SGP4 propagation, calculate Doppler shift.
- *Network-update*: create and propagate new active routing entries when one or more satellites are in sight; deactivate afterwards.
- *GS hardware control*, that is monitor and control:

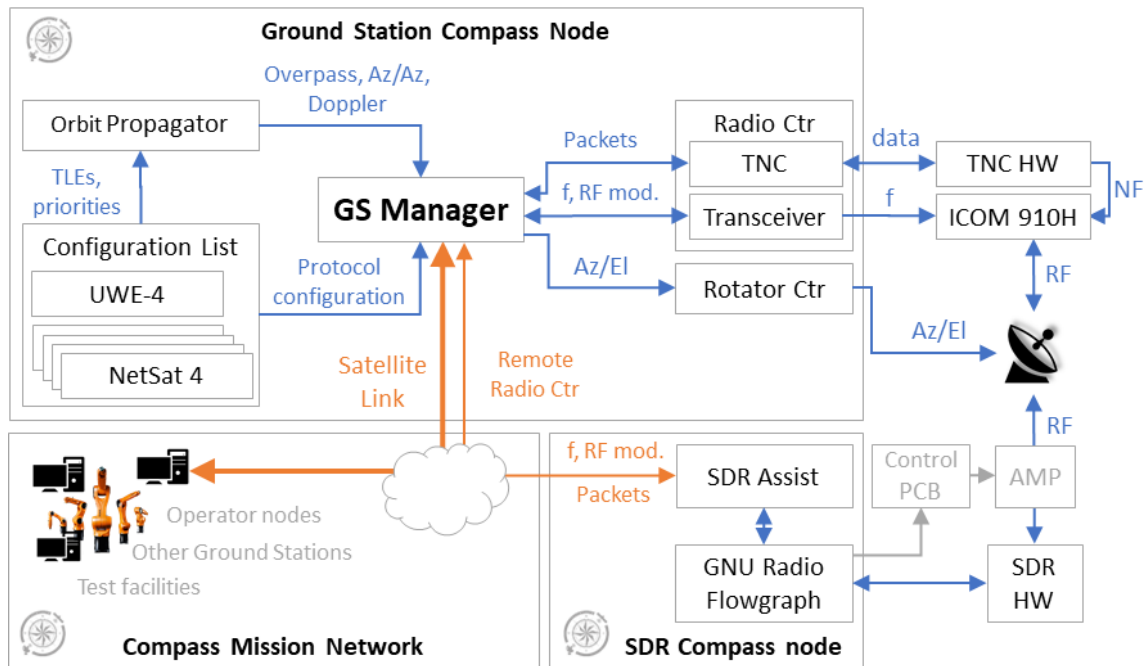


Figure 6.5: Ground Station of the University of Würzburg

- antenna rotator
- transceiver’s band, frequency and amplification
- remote Software Defined Radio (SDR) nodes
- send and receive packets to/from the terminal node controller (TNC)

Before this work the tasks were performed using several non-interconnected software solutions. The first implementation step was to create software modules for all relevant tasks shown in figure 6.6:

- *Satellite Configurator* holds a list of desired satellites configured with: name, NORAD-ID, frequency, RF modulation, Compass address, protocol chain and tracking priority
- *Propagator* calculates overpasses and current Doppler shift for all satellites listed in the *Satellite Configurator* and performs periodic TLE updates from defined sources.
- *GS Manager*: configures *Radio*, *Amplifier*, *Rotator*, *TNC* and remote *SDR* components to enable communication with currently overpassing satellites.
- *Radio* driver for ICOM IC910, ICOM IC9100 and Kenwood radios, which are connected via serial interface. Converts modulation and frequency configuration from *GS Manager* to low-level control messages.
- *Amplifier* driver for *BEKO* amplifier. Converts transmission strength configuration from *GS Manager* to control the amplifier hardware.
- *UHF Rotator* and *Dish Rotator* driver for antenna rotator at the University Würzburg and ZfT respectively. Uses relative satellite position from *GS Manager* to orient the antenna appropriately.
- *SDR*: uses RF and modulation configuration from *GS Manager* to set-up remote

Compass-enabled SDR receivers and injects received raw-data from SDRs to the GS Manager's packet decoder.

- *Encoder/Decoder*: decodes received raw-data using the protocol configuration of the currently tracked satellite and injects them as Compass packets into the mission network.
- *SiDS interface*: receives raw-data from external radio amateur ground stations and injects them to the GS Manager's packet decoder.
- *Heuristic Decoder*: decodes raw-data without knowledge about its source. Tries all protocol stack configurations from the *Satellite Configurator* for decoding and perform plausibility checks.

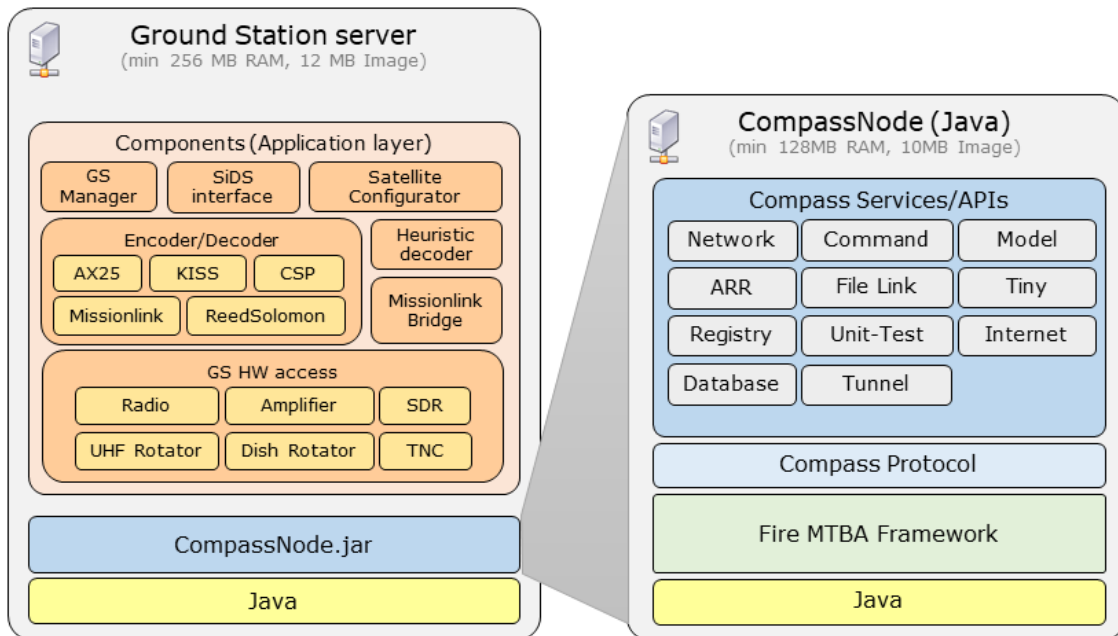


Figure 6.6: Software structure of the Ground Station server

Initially, the UHF GS server implementation was running on a dedicated high-power workstation. Due to the lightweight nature of the CompassNode component and to simplify the hardware set-up, the entire server implementation is running since 2017 on a single Raspberry PI (currently Raspberry Pi 3) with only 4 – 20% CPU load. Due to its modular structure, the same implementation with another *Rotator* configuration is also used to control the hardware of the second (backup) UHF ground station at ZfT.

The satellite tracking algorithm was significantly improved to support formation operations and was made *traffic-sensitive*. The basic idea of the tracking is: if multiple listed satellites are in range, then the one with the highest priority is selected for tracking. In a formation, all satellites are prioritized equally – if multiple satellites with the same priority are in-range then the first ascending one is selected. During the operations, the GS server scans outgoing (ground-space) traffic and detects whether another in-range satellite with the same priority as the currently tracked one is desired by the (auto-)operations and



automatically switches the antenna tracking. The *encoding* of the outgoing ground-space Compass packets is performed as follows:

1. The target address is used to find the corresponding satellite in the *Satellite Configurator* list.
2. If found: the protocol chain configuration is used for the encoding.
3. If not found: the protocol chain of the currently tracked satellite is used.

All software components of the GS server are accessible for monitoring and control via the Model server. An example of the GS server model visualization in the *Model View* of the Compass Operations front-end is shown in figure 6.7. In addition to the Model View the front-end also provides different monitoring components, which can be used to visualize remote model values in a convenient way. In the current missions the *Hardware* model branch of the GS server is mainly used for monitoring, i.e. the values are only changed manually for testing purposes. The *Satellite* model branch contains all currently configured satellites. The operator can change satellite priorities, protocol chains (for testing purposes) or the transmission signal strength.

Name	Value	Type	ID
> <input type="checkbox"/> Commands			
> <input type="checkbox"/> Compass			9151
▼ <input type="checkbox"/> Sat			6630
<input type="checkbox"/> SAT current	NetSat4	char[7]	44606
> <input type="checkbox"/> Custom Sat			12065
> <input type="checkbox"/> UWE-4			34553
> <input type="checkbox"/> UWE-3			62906
> <input type="checkbox"/> UWE-4E			27951
> <input type="checkbox"/> NetSat			13066
▼ <input type="checkbox"/> NetSat-1			2268
<input type="checkbox"/> TLE	1 46506U 200...	char[1...	30433
<input type="checkbox"/> Uplink Strength	100%	s32	64031
<input type="checkbox"/> NORAD ID	46506	s32	46227
<input type="checkbox"/> Priority (0=min)	9	s32	57820
<input type="checkbox"/> Protocol	AX25_CRC32_...	char[2...	60708
<input checked="" type="checkbox"/> Auto create subsys		u8	34304
> <input type="checkbox"/> NetSat-2			30509
> <input type="checkbox"/> NetSat-3			49905
> <input type="checkbox"/> NetSat-4			24370
> <input type="checkbox"/> Parking			42189
> <input type="checkbox"/> NetSat-EM			59992
<input type="checkbox"/> Default tracking from	-3.0°	double	39821
▼ <input type="checkbox"/> Advanced			29070
<input type="checkbox"/> Use heurisitc conve		u8	30853
> <input type="checkbox"/> Hardware			62819
> <input type="checkbox"/> Database			29559

Name	Value	Type	ID
> <input type="checkbox"/> Commands			
> <input type="checkbox"/> Compass			9151
> <input type="checkbox"/> Sat			6630
▼ <input type="checkbox"/> Hardware			62819
▼ <input type="checkbox"/> Antenna			12097
<input type="checkbox"/> Az	292°	s32	37884
<input type="checkbox"/> El	1°	s32	60589
<input type="checkbox"/> TargetAz	290°	s32	49398
<input type="checkbox"/> TargetEl	2°	s32	55463
<input type="checkbox"/> Camera.jpg	/9j/4AAQSkZ...	binary...	19813
<input type="checkbox"/> Rotor on	<input checked="" type="checkbox"/>	u8	15175
<input type="checkbox"/> Beam width	21°	s32	39337
<input type="checkbox"/> Control rotor	<input checked="" type="checkbox"/>	u8	769
▼ <input type="checkbox"/> Transceiver			26862
<input type="checkbox"/> f	435603490Hz	s32	11115
<input type="checkbox"/> RFPower	100%	s32	32237
<input type="checkbox"/> Alarms	0	s32	3706
<input type="checkbox"/> CSP Address Sat	1	s32	65124
<input type="checkbox"/> Downlink Channel	<input checked="" type="checkbox"/>	u8	21318
<input type="checkbox"/> Uplink Channel Act	<input type="checkbox"/>	u8	36973
<input type="checkbox"/> All Received	161	s32	16462
<input type="checkbox"/> All Sent	914	s32	25733
<input type="checkbox"/> Wind	0	s32	39449
> <input type="checkbox"/> Location			25694
▼ <input type="checkbox"/> SDR			38495
<input type="checkbox"/> Radio Node	1:40	char[4]	24729
<input type="checkbox"/> Connected	<input checked="" type="checkbox"/>	u8	24407
<input type="checkbox"/> Received SDR1	268	s32	1204
<input type="checkbox"/> Received SDR2	0	s32	31110
> <input type="checkbox"/> TNC			9372

Figure 6.7: Model service used to monitor and control a ground station

## 6.4 Mission Server

In contrast to the Ground Station server, the *Mission Server* is decoupled from the ground station's hardware. The software is based on the CompassNode and was created *without additional code*, i.e. already available Compass services and channels were configured to support its responsibilities:

- *Traffic Database*: store relevant incoming and outgoing Compass traffic in the *Traffic database* and offer *Database service* to enable other nodes to request history packets.
- *Model Database*: store all detected model changes (by processing forwarded Model server packets) in the *Model database*.
- *Gateway* between multiple ground stations on the one side and operation nodes on the other
- *Model caching* of all nodes within the mission network
- *Registry service* hosting
- *Command service* for to restart channel connectors or to repair SQL database connection.
- Connect multiple Compass sub-networks (e.g. ZfT and University of Würzburg, Chair 7) via secure SSH channel to one common mission network

Currently one Mission Server instance is used for UWE and NetSat satellites. For all ongoing missions separate Mission server instances were created. The Compass Operations front-end provides a selector for mission network entry points (see figure 6.8), which is used by the operators to connect to different networks.

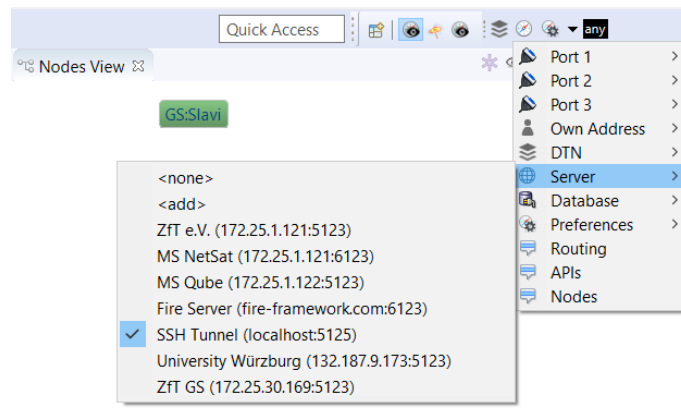
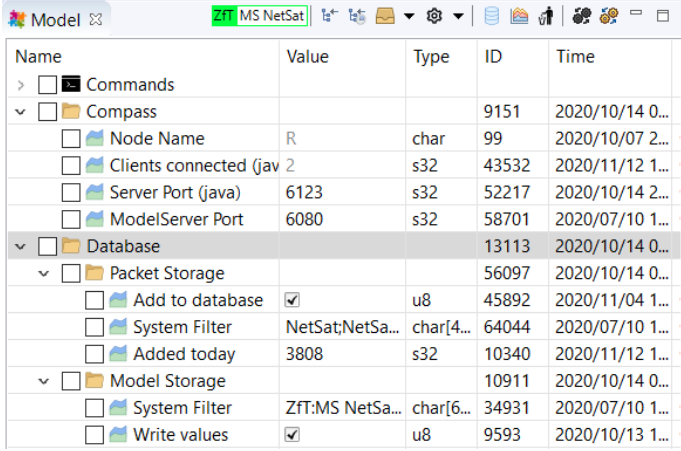


Figure 6.8: Compass Operations front-end: selection of a mission network entry point.

All relevant monitoring and control values of the Mission Server are accessible via the Model service – see figure 6.9 for more details. In practice only the **Added today** entry is monitored at ZfT to get informed about the amount of written model values.



Name	Value	Type	ID	Time
> <input type="checkbox"/> Commands				
▼ <input type="checkbox"/> Compass			9151	2020/10/14 0...
<input type="checkbox"/> Node Name	R	char	99	2020/10/07 2...
<input type="checkbox"/> Clients connected (jav	2	s32	43532	2020/11/12 1...
<input type="checkbox"/> Server Port (java)	6123	s32	52217	2020/10/14 2...
<input type="checkbox"/> ModelServer Port	6080	s32	58701	2020/07/10 1...
▼ <input type="checkbox"/> Database			13113	2020/10/14 0...
▼ <input type="checkbox"/> Packet Storage			56097	2020/10/14 0...
<input type="checkbox"/> Add to database	<input checked="" type="checkbox"/>	u8	45892	2020/11/04 1...
<input type="checkbox"/> System Filter	NetSat;NetSa...	char[4...	64044	2020/07/10 1...
<input type="checkbox"/> Added today	3808	s32	10340	2020/11/12 1...
▼ <input type="checkbox"/> Model Storage			10911	2020/10/14 0...
<input type="checkbox"/> System Filter	ZfT:MS NetSa...	char[6...	34931	2020/07/10 1...
<input type="checkbox"/> Write values	<input checked="" type="checkbox"/>	u8	9593	2020/10/13 1...

Figure 6.9: Model service used to monitor and control a mission server

## 6.5 External Ground Stations

Shortly after the UWE-3 launch, the UWE team was facing very high packet loss ratio in both uplink and downlink directions. After a long-lasting in-orbit software update, *frequency sweep* tests were performed to detect frequency ranges with comparable less noise, such that a frequency with tolerable performance could be selected. Nevertheless, many communication windows were missed during the LEOP phase of the UWE-3 mission. In a formation mission this situation may lead to a mission fail.

During the last year of the UWE-3 mission, options were investigated in the scope of this thesis for accessing external ground stations as a backup for UWE ground station and to increase the amount of communication windows. At that time many radio amateurs supported the UWE team by sending e-mails with received satellite packets (KISS frames). Thus, as a part of this thesis realization, a new interface was defined that can be used to automate the process of packet forwarding, called *Simple Downlink Share Convention* (SiDS, can be downloaded under [Dom15b]). The interface is not limited to a specific satellite – instead it can be used to forward packets from any satellite to the corresponding satellite provider’s server.

As already described in section 3.2.2, Mike Rupprecht has implemented the proposed interface in his widely used software for visualization of received satellite beacons. Later the convention was ported by Daniel Estévez to *GNURadio*. Today SiDS is also one of the incoming data interfaces of the open-source *SatNOGS platform* – a global network of satellite ground-stations, which continuously attracts worldwide attention in the satellite community [Whi+15]. In contrast to the default decentralized SiDS application, all received satellite packets are additionally forwarded by the radio amateurs to a central SatNOGS server. So instead of creating a separate SiDS server, satellite providers can log into the SatNOGS web interface and configure the *Grafana*-based visualization dashboard, which is fed with packets forwarded by external ground stations. Our team started to use SatNOGS as an additional visualization tool for UWE-4 and NetSat beacons and it is planned to use the platform as backup for future ZfT’s missions. A screenshot of the

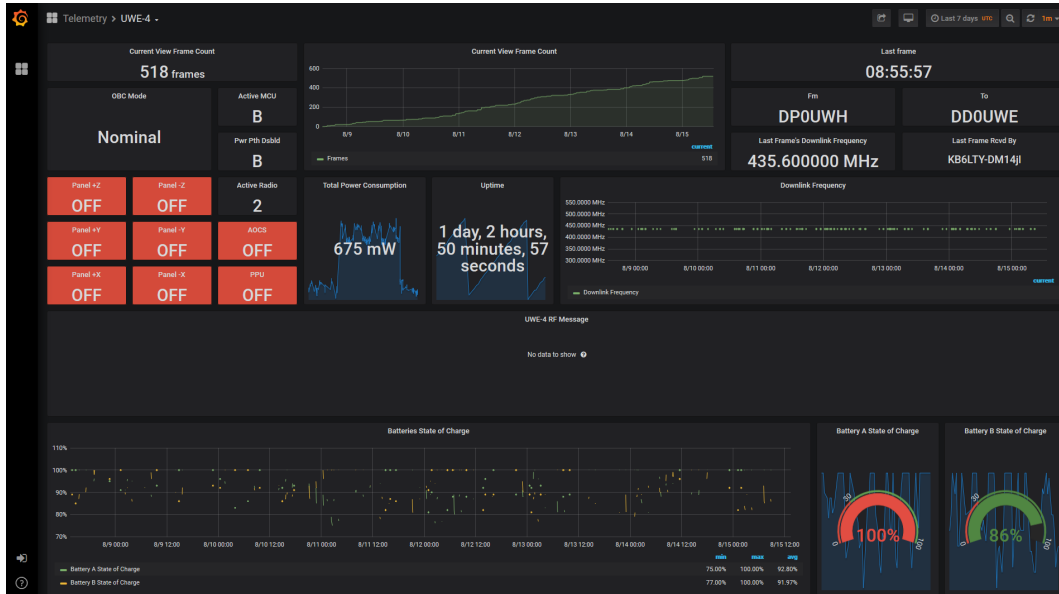


Figure 6.10: Visualization of UWE-4 beacons with SatNOGS dashboard

dashboard that has been elaborated by Philip Bangert to visualize UWE-4 data, is shown in the figure 6.10.

In contrast to other available ground station network solution, such as Genso [SK07], the hardware of SiDS ground stations is not accessible from the outside. Therefore a radio amateur never loses the control over the hardware, making the SiDS concept spread fast in the community.

To date over 440.000 UWE-3 and UWE-4 packets were received from external ground stations as direct SiDS packets (see figure 6.12). Also live data was received during the local overpasses, therefore reducing the downlink packet loss ratio. Since the SiDS interface is intentionally designed to be protocol-agnostic, all packets are forwarded as Layer 2 byte streams. Nevertheless, the SiDS server component of the GS software automatically generates Compass address for all external ground stations based on their *callsigns*. Every time a packet is received from a radio amateur, its auto-generated Compass address is added to the packet's route list. An example of how these packets were visualized in the Compass Operations front-end is shown in the screenshot 6.11.

## 6.6 Compass Operations front-end

Before Compass, the UWE team used the *UWE-X Operations Software*, which was build on top of the *Eclipse Rich Client Platform* and was runnable on Windows and OSX. The protocol configuration (shown in figure 3.1) included *Missionlink*, which was used between the ground relay and the UWE-3's space relay (OBDH subsystem). Several views were provided by the UWE-X GUI to access remote Missionlink services. The protocol and the views are described in more detail in [Dom12]. Missionlink allowed only *one-to-one*

The screenshot shows the 'Compass Packet View' application window. It features a table with columns for API, Rec [UTC], Time, From, Via, To, Value, Ref, Channel, ID, and CRC. The table lists 15 packets, most of which are 'Beacon' type and 'Uwe-3 beacon' value, originating from various ground stations like UWE-3:OBDH and UWE-4:OBDH. The last entry is a 'GDS' packet with the value 'Set remote 1280'.

API	Rec [UTC]	Time	From	Via	To	Value	Ref	Channel	ID	CRC
Beacon	15:46:49.000		UWE-3:OBDH	EU1X:X -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	32848	
Beacon	15:46:49.000		UWE-3:OBDH	UX5U:L -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	32848	
Beacon	15:47:55.000		UWE-3:OBDH	EU1X:X -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	32849	
Beacon	15:49:45.000		UWE-3:OBDH	EU1X:X -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	32851	
Beacon	17:19:03.000		UWE-3:OBDH	G7GQ:W -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	32942	
Beacon	03:24:40.000		UWE-3:OBDH	UX5U:L -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	16408	
Beacon	03:27:36.000		UWE-3:OBDH	UX5U:L -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	16411	
Beacon	03:29:37.000		UWE-3:OBDH	UX5U:L -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	16413	
Beacon	12:54:00.000		UWE-3:OBDH	EU1X:X -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	49245	
Beacon	14:28:04.000		UWE-3:OBDH	EU1X:X -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	49341	
Beacon	14:28:15.000		UWE-3:OBDH	UX5U:L -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	49341	
Beacon	14:49:23.000		UWE-3:OBDH	UX5U:L -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	16528	
Beacon	18:06:19.000		UWE-3:OBDH	G7GQ:W -> GS:Server	GS:Slavi	Uwe-3 beacon	*	*	32857	
GDS	03:16:45.000		UWE-4:OBDH	VK5HJ -> GS:local	GS:Slavi	Set remote 1280	*	*	13184	8C 25

Figure 6.11: Compass packets injected by external ground stations

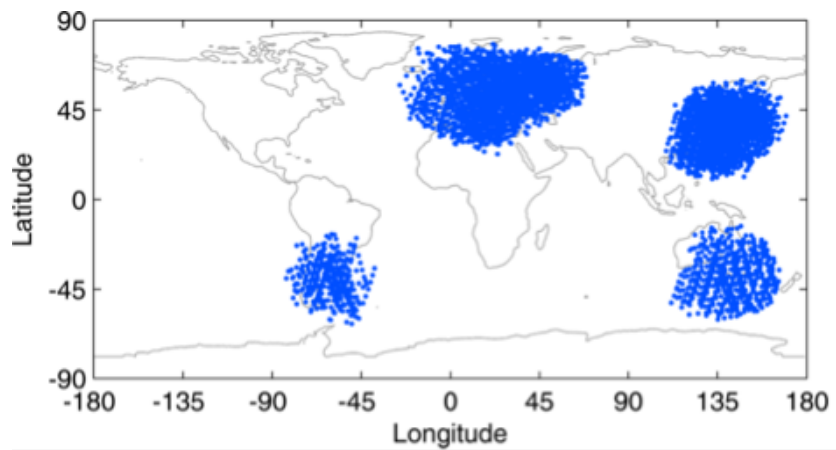


Figure 6.12: Receive locations of UWE-3 packets submitted by radio amateurs

communication scheme, i.e. the entire Missionlink network consisted only of one ground node and one space node (UWE-3 OBDH). The protocol lacked intrinsic capabilities to hold the sender's and receiver's addresses, i.e. the packets from different UWE-X GUI instances could not be distinguished. So, the UWE-X GUI was only capable of providing operation functions for only one remote satellite.

With Compass, a node concept was introduced with all mission network participants being uniformly accessible. The Missionlink has become obsolete, as the Compass protocol was designed with all Missionlink advantages in mind. As a part of this dissertation the UWE-X GUI was used as a template to create a new *Compass Operations front-end* (or just *Operations software*) to support dynamic Compass networks and uniform operation of all nodes within the network. As any other Compass-enabled Java software, the new front-end is based on the *CompassNode* and offers additional graphical interface for all available services.

A Compass Operations instance is represented as a node in the mission network, i.e. the software enters the network by connecting itself to some existing node. Out-of-the box the software supports 5 configurable channels:

- 3 *serial* channels, which can be used to directly connect multiple Compass-enabled satellite subsystems or UNISEC Development Kits.
- A *TCP client* channel is used to enter an existing mission network via a TCP entry point.
- A *TCP server* channel offers a TCP entry point for other nodes.

The operator can configure all channels depending on needs and decide to either span a local-only Compass network, or to (additionally) connect to an existing Compass network. At ZfT two separate Compass networks are in use: the common network and the NetSat network. The former is used for simulation nodes, test facilities, development nodes, etc. The latter consists of ground stations, mission server and operator nodes. As yet most of the time both networks are connected together (for convenience) but since the number of Compass nodes continuously rises and further missions are ongoing, at some point in future both networks will be disconnected.

The Operations Software is based on dynamically selectable and arrangeable *views*. The software offers default Compass views, i.e. views provide GUI interface for existing services:

- *Nodes View*
- *Packet View*
- *Command View*
- *Model View*
- *Uplink View* and *Downlink View*
- *Unit Test View*
- *Tiny View*
- *Echo View*
- *Value Monitors*

In addition to that the software also provides more specialized views that were designed for specific tasks:

- *Scheduler View* is used to record auto-operations schedules.

- *Schedule* visualizes satellite overpasses and current auto-operation tasks.
- *Formation View*: visualizes the orbit position of all formation nodes.
- *Attitude View*: visualizes the attitude of all formation nodes.

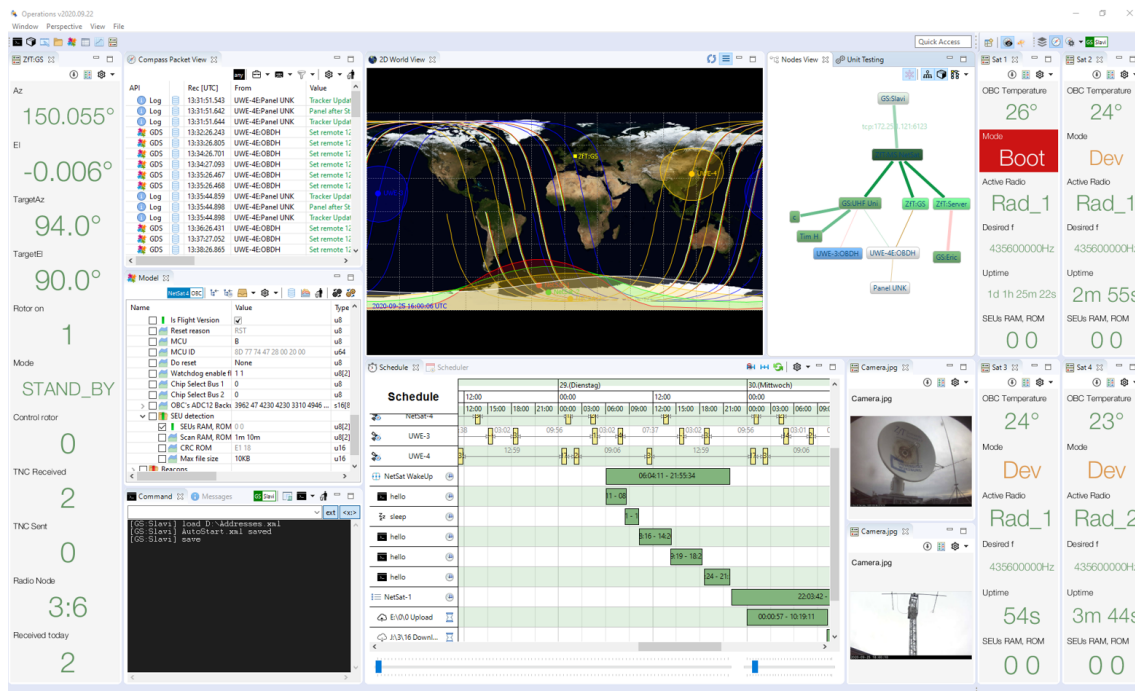


Figure 6.13: Compass Operations front-end

Since the Operations Software was designed to control any Compass-enabled node, most of the existing and recently created hardware and software components at ZfT were enriched with Compass capabilities. For example, both high precision motion simulators (turntables) are now represented as separate nodes in the mission network, such that all other Compass-enabled simulation nodes can gain access to their monitor and control values [Ruf+17][DRS18]. The turntables can also be controlled directly from the Operations Software via the Model service.

To speed up the process of changing the preferences, most of the basic Compass settings are accessible via the *Compass Drop-Down Button*, located in the main toolbar near the *global node selector* (see figure 6.14). All other settings are done in the corresponding views.

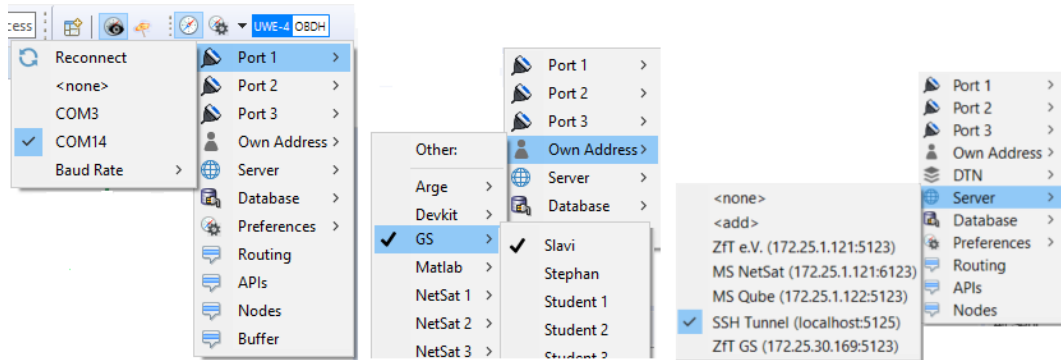


Figure 6.14: Compass Drop-Down Button with some of the available preferences

### 6.6.1 Node selection

The Compass Operations front-end contains multiple views, whereat most of them can be configured to control or view some selected Compass node (e.g. `NetSat1:OBC`). By clicking on the node-address, a node selector appears, thus allowing the change of the desired address (see figure 6.15). The functionality of the correspondent view is then targeted at the selected node.

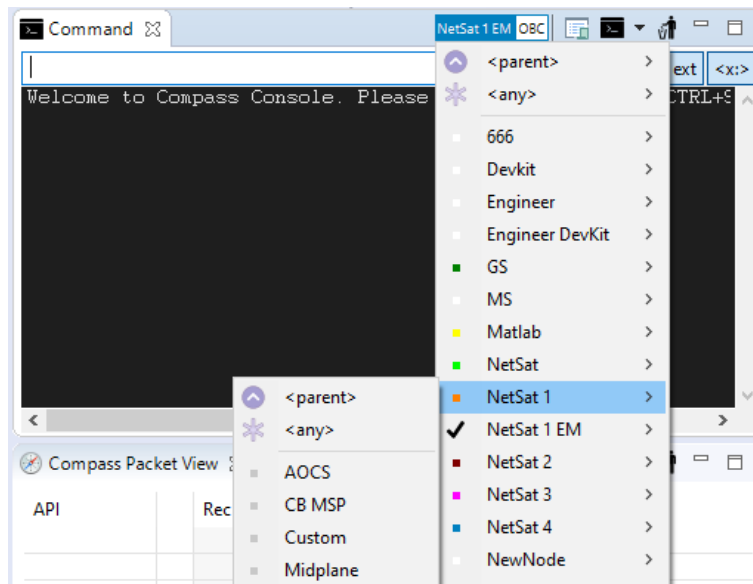


Figure 6.15: Node selector in a single view

To simplify the selection of the node in multiple views, the node selection of the desired views can be set to *parent*. From now on these views will listen to the *global node selector* shown in figure 6.16. The node selection can also be performed in the *Nodes View* by right-clicking on the desired node (figure 6.6.2) and selecting the desired function in the



appeared pop-up menu, which in turn brings up the corresponding view and preselect its target.



Figure 6.16: Global node selector

## 6.6.2 Nodes View

The Nodes View shows the local knowledge of the Compass network, which is dynamically derived from the incoming packets and network beacons. The View directly accesses the rows in the Network table of the Compass implementation and gets automatically informed about its changes. All nodes can be right-clicked to allow fast operations, e.g. model monitoring/control or commanding. Green lines denote an active and red lines yet inactive connections. In the tree-styled visualization, the local node appears on top of the network tree. Compass network can have more than one connection route to some specific node but only shortest routes are visualized.

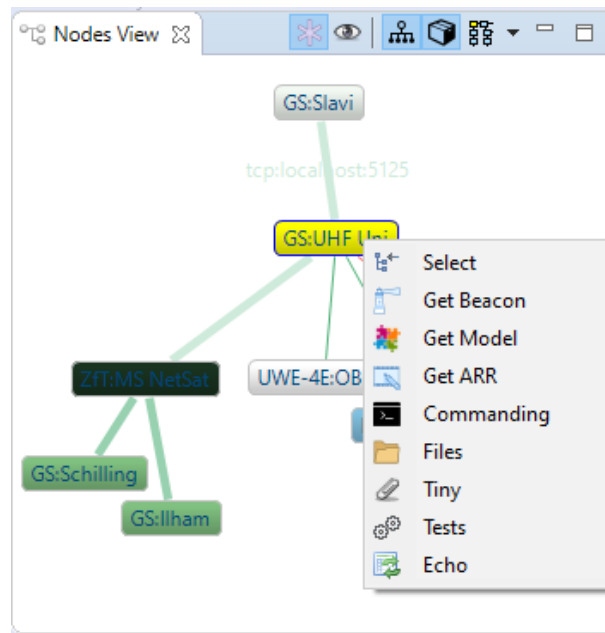


Figure 6.17: Network View with direct right-click node control

The connection lines blink every time a compass packet is transferred via the corresponding route. If no packet was transferred from a specific node within 60 s, the connection is considered inactive (red). Every Compass node automatically transmits network

beacons every 10 s, thus holding the line active. The line thickness denotes the maximum possible speed of the connection – on mouse over the actual speed value is shown in a tooltip.

The view's toolbar contains following buttons (figure 6.18, from left to right):

1. *Show connection age*. Connection colors fade out with time if no packet is received, thus signaling its age.
2. *Show packets*. Nodes and connection lines blink on packet transmission.
3. *Visualization style*: auto-arrangement of nodes. Available styles are: None, Spring, Radial, Tree (default).



Figure 6.18: Nodes View buttons

### 6.6.3 Packet View

This view is used for the monitoring of the incoming and outgoing Compass traffic. It is divided into four collapsible tabs:

- *API stats*: packet statistics grouped by the service.
- *Packet creator*: create a user defined packet or view/modify a selected one.
- *DTN buffer*: shows all currently buffered packets that were set with DTN flag and could not be delivered yet
- *Packets*: here all incoming and outgoing packets are listed. Answer packets are displayed as child elements of the request packets (see command packet in the screenshot 6.19).

The view's toolbar contains the following buttons (figure 6.20, from left to right):

1. *Node selector*. Can be used to show traffic from/to specific node
2. *Tools* (figure 6.21, left)
  - Load packets from DB: load packet history from remote node
  - Load packets from File: load packets from existing traffic file
  - Packet wizard: tools to decode HEX of any packet
  - Post-process traffic file: loads all packets from a traffic file and creates text files and model folders with CSV. The created folder is shown when finished.
3. *Recording* (figure 6.21, right): can be used to record traffic
  - Auto-Start. If set the recording is started automatically after reboot. Always active recording
  - Use separate files for started recordings

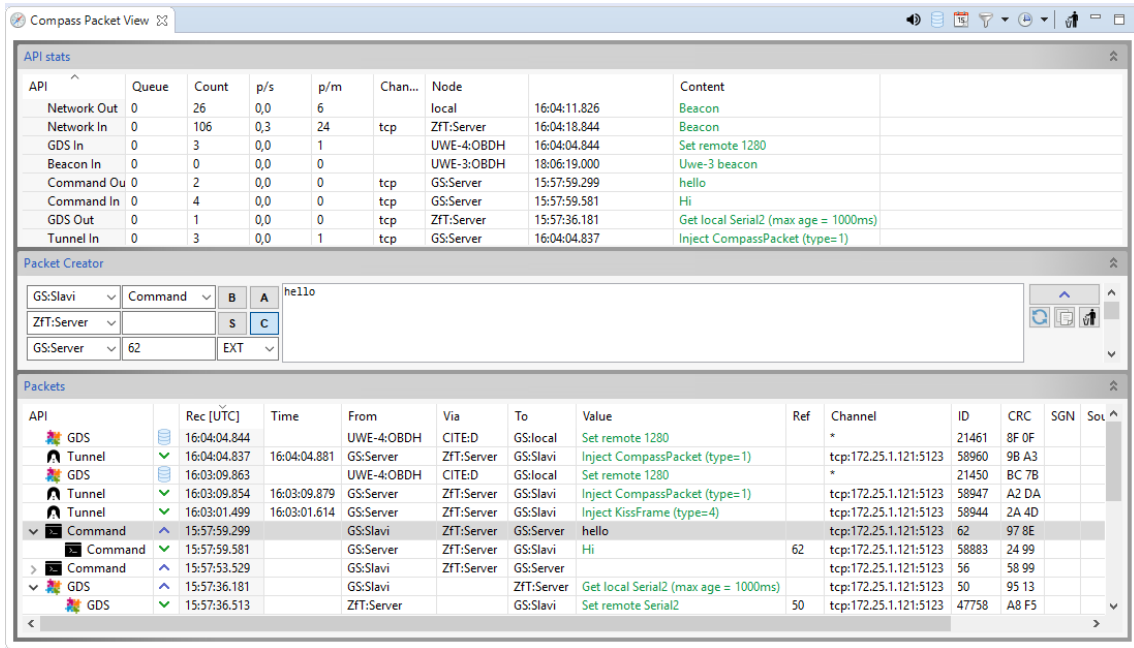


Figure 6.19: Packet View with three collapsible groups: service statistics, packet creator and packet list

- Path target path into which the recordings should be stored
4. Show/hide selected API packets
  5. Settings
  6. Clear entries in the packet view



Figure 6.20: Compass Packet View buttons

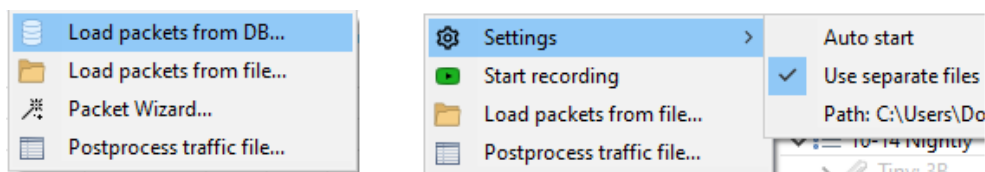


Figure 6.21: Packet View: tools and recording menus

The Packet View can for example be used to create user-defined packets for testing purposes, view incoming log messages or to investigate malfunctions by analysing the service traffic. Several actions can be performed on one or multiple selected packets by opening a context menu with the right mouse button:

- *Ignore API*: hide all packets from the service defined in the selection.
- *Ignore API from sender*
- *Copy* content to clipboard – the format and type is selected in the sub-menu (figure 6.22)

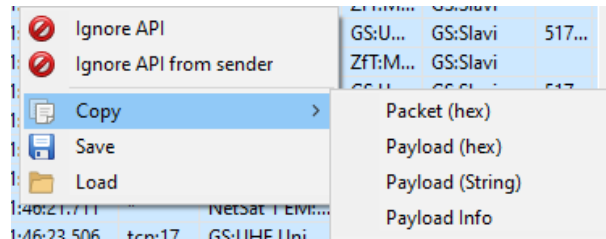


Figure 6.22: Packet View: available actions for selected packets

## DTN functionality

Since NetSat, both Compass OS and CompassNode-enabled systems support per-packet DTN functionality. DTN guarantees that packets marked with DTN flag:

- will be delivered to the target either now or as soon as the target becomes available
- packets will be delivered in the same order as they were created/sent (*ordered DTN mode*)
- *Pass To Next DTN* (deactivated by default): if the target is not accessible, but on the way to the target there exist another DTN-able node, the responsibility will be shifted from local to that node.

If *Pass To Next DTN* is active and an embedded DTN node (e.g. OBC) is in the route towards an inactive node, the embedded node will take over the responsibility of the DTN retransmission. In this case the local buffer will become empty, i.e. it is the remote node guarantees that the packets are now in its buffer. Embedded nodes with activated DTN may only hold limited amount of packets (100 in the NetSat:OBC) in the buffer as the result of RAM/Flash limitations.

### Example: Pass to Next DTN

Current communication route between the operator and target node is:

$$Local(DTN - able) \rightarrow GS \rightarrow NetSat2 : OBC(DTN - able) \rightarrow Panel(inactive)$$

With activated DTN the operator can send DTN-packets to yet inactive Panel, which initially will be buffered on local machine and after some *Local* ↔ *OBC* interaction

(ACK) will be buffered on the OBC. As soon as Panel becomes active, the OBC will flush all buffered Packets to Panel.

The DTN functionality is activated with the DTN button in the main toolbar (figure 6.23). This enables DTN for all *user-created* packets, such as commands or model access. Local DTN buffer can be viewed and controlled in the Packet View, thus allowing the operator to see the current state and to delete either single packets or all packets for a specific target. As soon as the target node becomes reachable, the local DTN handler tries *every second* to transmit a packet until an ACK is received. The repeating process is not limited in time – but since the operator can comfortably control the buffer, this is the best compromise as compared to complex per-packet configuration (max repeat etc.).

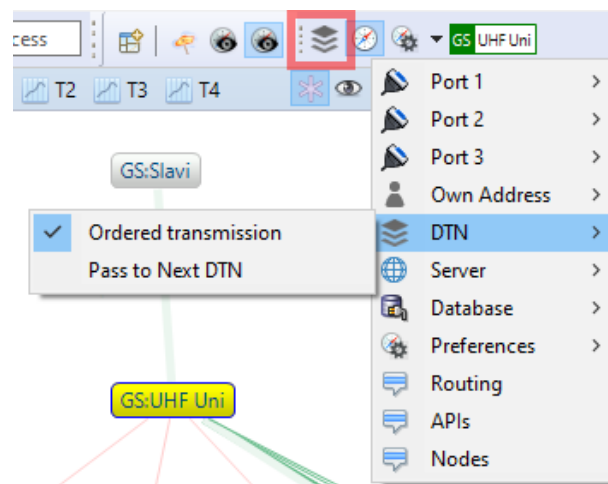


Figure 6.23: DTN Activation and configuration

### 6.6.4 Command View

This view provides access to *human-readable services*, e.g. to execute commands on remote nodes. Remote commands can also be executed in the Model View (figure 6.25). The view's toolbar contains following buttons (figure 6.24, from left to right):

1. *Node selector*
2. *Service selector*: Command (default), Chat, Tiny and Echo. Depending on the selection the entered text is sent either as Command service packet or any other selected. Echo is the fastest way to test if the remote node is really accessible.
3. *Clear console*

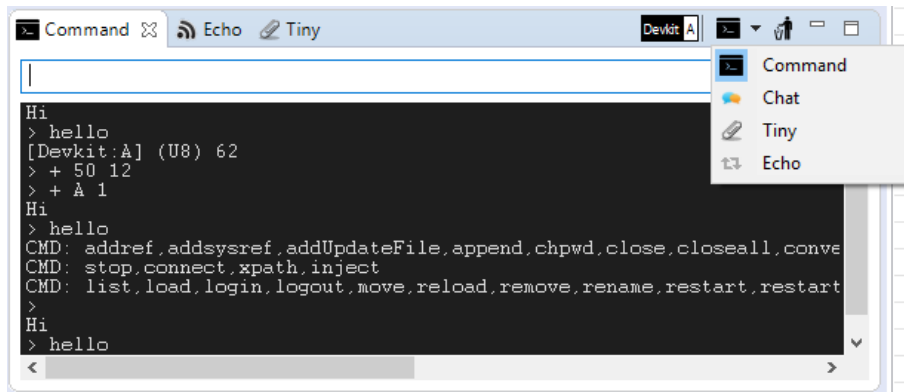


Figure 6.24: Command View with selectable human-readable services: Command service, Chat service, Tiny service, Echo service

A list of available commands can be received from the target node by sending an empty command, i.e. press **ENTER** without any text in the command line (see section 4.3.3 for more details). The Command View will make proposals on **CTRL+Space** and automatically fix the letter case of the entered commands before the transmission (Command service is case sensitive in the current implementation). All received command lists are cached in the view and are available after restart, thus a command list request is normally required only for new remote nodes and for nodes with changed commands.

In the Tiny mode the entered text line is compiled with the integrated Tiny compiler, followed by the transmission of the byte-code. This way more complex or constrained operation commands can be created, for example: delete a file if it is larger than 100 kB or switch off the propulsion system if the energy level is below 10%.

The screenshot shows a table with the following columns: Name, Value, Type, Time, and ID. The table contains the following data:

Name	Value	Type	Time	ID
Commands				
add		+		
COMPCH		+		
COMPMAP		+		
COMPSET		+		
hello		+		
General			1970/01/01 01:01:18...	1
Compass			1970/01/01 01:01:18...	200
Some test group			1970/01/01 01:01:18...	666C
New-styled group			1970/01/01 01:01:18...	666E

Figure 6.25: Command execution using the Model View

### 6.6.5 Model View

The Model View is used to monitor and to modify remote model values. It does not need any a priori knowledge, instead it can request yet unknown models and cache them for later use – also after the restart. It can be used to control one or many models at the same time, as shown in the figure 6.27. By selecting the target node in the view’s toolbar, the current cached model of the node is shown. If not available, the model scheme can be requested with the corresponding button in the view’s toolbar.

The model values can be modified directly in the Value column. This generates an appropriate model packet, which is transmitted to the selected node. Until the remote node acknowledges the modification, the value is displayed as **pending**...

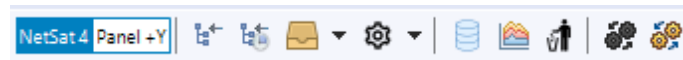


Figure 6.26: Model View buttons

Most view’s functions are accessed in the toolbar (see figure 6.26, from left to right):

1. Node-selector
2. Poll once selected values
3. Poll periodically selected values
4. Remote subscription menu
5. Settings (e.g. polling period)
6. Automatically fetch history data for selected values
7. Open a simple graph for selected values
8. Delete selected elements from the view (not from the remote system)
9. Request target’s entire model knowledge
10. Request target’s own model only

With the Model View the entire Compass network can be monitored and controlled at the same place. All Compass Operations front-end users can access nodes without having a deeper knowledge of their particular implementation. At ZfT all satellite-related functional interfaces between *cooperating nodes* are entirely described by the *global model path* (Compass address plus local model path) and the corresponding *value range*.

### Advanced Visualization

If the remote node provides *GUI hints* for its model values as described in section 4.3.10, the value representation is enriched with colors (current limits), read-only handling, units and convenient controls: check-box, drop-down, date/time selector or address resolution (see comparison in figure 6.28).

Name	Value	Type	Time	ID	REC
> <input type="checkbox"/> NetSat 1 EM:Panel Uninit				41:13	
> <input type="checkbox"/> NetSat 1:AOCs				4:3	
> <input type="checkbox"/> NetSat 1:OBC				4:1	
▼ <input type="checkbox"/> NetSat 1:Panel +X				4:7	
> <input type="checkbox"/> General				1	
> <input type="checkbox"/> Compass				200	
> <input type="checkbox"/> BMX				10010	
> <input type="checkbox"/> Panel				20000	
> <input type="checkbox"/> Torquer				20030	
> <input type="checkbox"/> Clockspeed				20060	
> <input type="checkbox"/> Housekeeping				20070	
> <input type="checkbox"/> Cell			2020/10/07 16:36:39.801	20071	
> <input type="checkbox"/> GPS			2020/10/07 16:57:01.875	20200	
> <input type="checkbox"/> E-Unit				60000	
> <input type="checkbox"/> Sunsensor				20090	
> <input type="checkbox"/> NetSat 1:Panel +Y				4:9	
> <input type="checkbox"/> NetSat 1:Panel -X				4:8	
> <input type="checkbox"/> NetSat 1:Panel -Y				4:10	
> <input type="checkbox"/> NetSat 1:Panel -Z				4:12	
> <input type="checkbox"/> NetSat 1:Panel Uninit				4:13	
> <input type="checkbox"/> NetSat 2:AOCs				5:3	
> <input type="checkbox"/> NetSat 2:OBC				5:1	
> <input type="checkbox"/> NetSat 2:Panel +X				5:7	
> <input type="checkbox"/> NetSat 2:Panel +Y				5:9	

Figure 6.27: Model View with currently known (cached) Compass network model

Name	Value	Type
> <input type="checkbox"/> Commands		
▼ <input type="checkbox"/> General		
<input type="checkbox"/> OBC Temperature	4°	
<input type="checkbox"/> LED Enabled	<input checked="" type="checkbox"/>	
> <input type="checkbox"/> Model storage action, ID	SAVE 1641	
<input type="checkbox"/> Mode	Dev	
> <input type="checkbox"/> Subsystems		
> <input type="checkbox"/> Development		
▼ <input type="checkbox"/> Time		
<input type="checkbox"/> Absolute Time	2020/10/02 12:05:54.313	
<input type="checkbox"/> Uptime	1h 57m 58s	
<input type="checkbox"/> Tics per second	1029	
<input type="checkbox"/> External RTC	RTC_2	
<input type="checkbox"/> Sync Time	RTC_1	
> <input type="checkbox"/> Compass		
> <input type="checkbox"/> OBC		
> <input type="checkbox"/> Beacons		
> <input type="checkbox"/> E-Unit		

Name	Value	Type
> <input type="checkbox"/> Commands		
▼ <input type="checkbox"/> General		
<input type="checkbox"/> OBC Temperature [°]	4	s8
<input type="checkbox"/> LED Enabled	1	u8
> <input type="checkbox"/> Model storage action, ID	2 1641	u16[2]
<input type="checkbox"/> Mode	2	u16
> <input type="checkbox"/> Subsystems		
> <input type="checkbox"/> Development		
▼ <input type="checkbox"/> Time		
<input type="checkbox"/> Absolute Time	1601633154313	u64
<input type="checkbox"/> Uptime [s]	7078	u32
<input type="checkbox"/> Tics per second	1029	u16
<input type="checkbox"/> External RTC	2	u8
<input type="checkbox"/> Sync Time	0	u64
> <input type="checkbox"/> Compass		
> <input type="checkbox"/> OBC		
> <input type="checkbox"/> Beacons		
> <input type="checkbox"/> E-Unit		

Figure 6.28: Model View: NetSat OBC model shown with (left) and without (right) GUI hints

## Value Recording

The view also provides support for value recording, which can be started and stopped with a single click on the red *recording-circle* of the desired value (see figure 6.27). Depending



on the current view's settings, all value recordings are stored either in a CSV or a binary file. The recorded files can then, for example, be post-processed or visualized in Microsoft Excel or Matlab.

### Model Update Subscriptions

With *model subscriptions* value updates are automatically received from remote nodes. The subscription and un-subscriptions functions are accessible in the view's menu (figure 6.29). Following subscription types are possible:

- *selected subscription* to single values
- *local subscription* all local values of the target node
- *global subscription* to the *entire knowledge* of the target node

The subscriptions are persistent, that is the target node memorizes the subscription permanently. At ZfT it is common to use one *global subscription* to the Mission Server knowledge. The Mission Server caches models from all available nodes and has *local subscriptions* to all ground stations, thus all Compass Operations front-ends receive automatically all monitor and control value updates of all mission-relevant space and ground nodes.

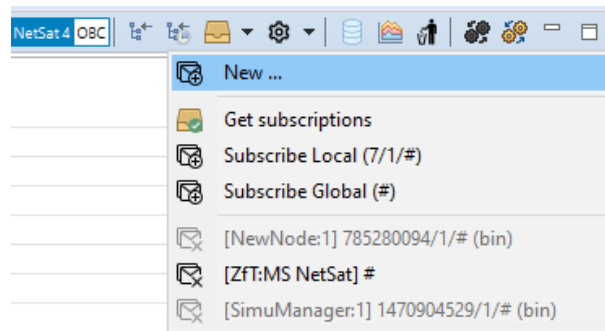


Figure 6.29: Model View: model subscription menu

### 6.6.6 Uplink and Downlink View

The Uplink and Downlink Views have both a very similar appearance. Both can have multiple file tabs, whereat every tab shows the link data and the status of the transmission. The views were ported from the UWE-X GUI and extended with Compass functions – more in-depth details of both views is described under [Dom12].

New link is created by pressing the *plus*-button, which opens a dialog with all required information. The view is composed of (please approach figure 6.31):

1. *Target node*
2. *General information*: file, chunk size.
3. *Chunk visualization*. Every chunk is displayed by one outer and one inner circle. The outer circle indicates the send state, and the inner circle indicates the receive state (see figure 6.30). A chunk row corresponds to a byte in the chunk bitmap. Multiple rows can be selected by the user.

4. *File buttons*: save, save-to and show contents, which opens up a dialog with file content.
5. *Send buttons*: transmit all, selected or unreceived chunks.
6. *Request buttons*: manually request the receive state to update the chunk visualization.
7. *Link monitor*: indicates the current data rate and the link activity.



Figure 6.30: Available Chunk States

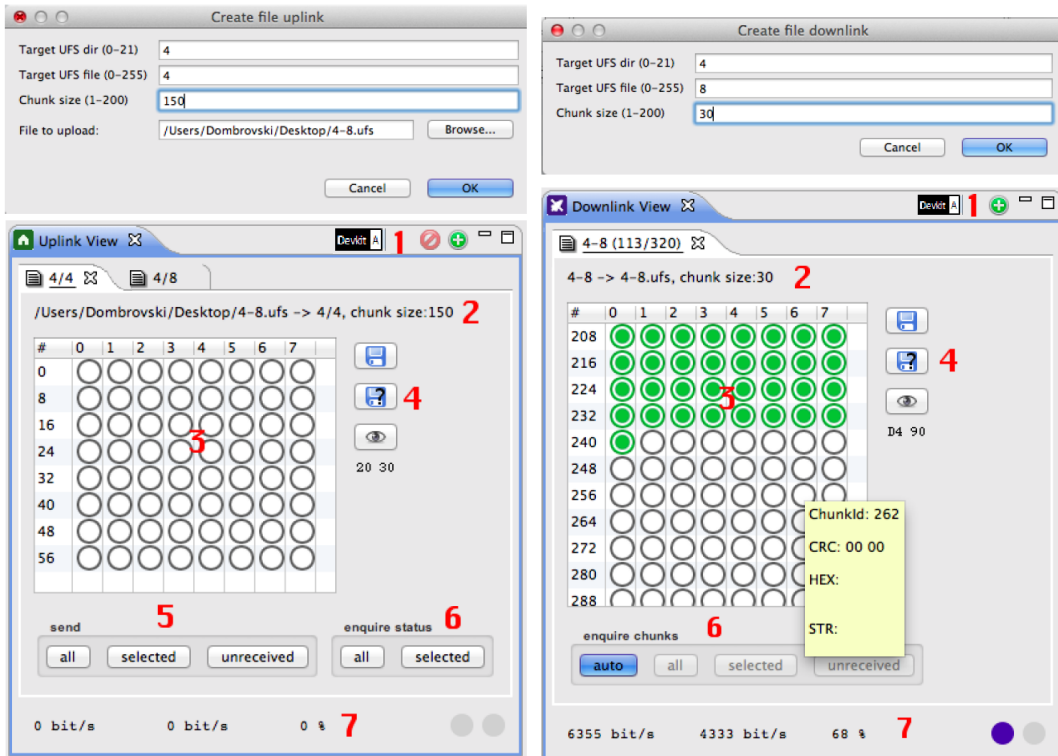


Figure 6.31: Uplink dialog and Uplink View (left), Downlink dialog and Downlink View (right)

### 6.6.7 Unit Testing

With the UnitTest View a list of all available unit tests can be requested from the selected node. Afterwards the tests and test groups can be individually selected for execution. The test execution is performed one-by-one and the received test process packets (see section 4.3.7 for more details) are used internally for test status visualization.

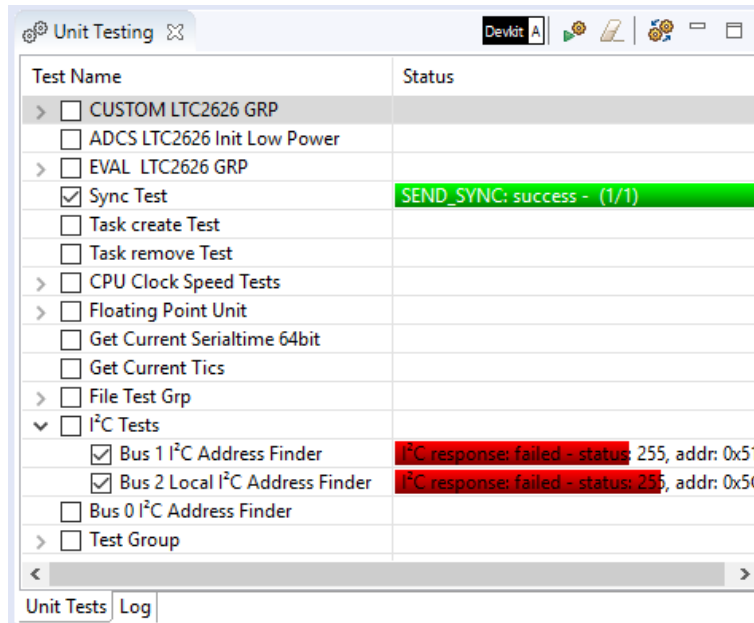


Figure 6.32: Unit Testing View with several executed tests

This view can also be utilized to perform all relevant tests after a satellite has been fully assembled. In addition, unit tests are frequently executed on flight-spare models to test software updates before uplinking the software image. During the development a unit test is a handy tool to try out different techniques and algorithms.

### 6.6.8 Value Monitors

The operator can create multiple Value Views, whereat every view shows a list of desired values being cached by the Model service. As soon as the Model service receives value updates, all corresponding value monitors become an update event. A value monitor visualizes the current value status (green, warning, alarm), if the corresponding model value contains *GUI hints*.

Next overpass	Next overpass	Tracking	Azimuth [deg]	Elevation [deg]	Range [m]	f Uplink [HZ]	f Downlink [HZ]	Uplink Strength [°]	TNC Received	TNC Sent	Value	Image
05:16:31	00:39:34	UWE-4E	90.00	85.00	8611139	435600000	435600000	10	15	2160	7	

Figure 6.33: Visualization of the ground station parameters

A value monitor is created in the Model View for one or multiple selected values. A monitor supports the visualization of numbers, text and images (binary model values with image extensions in their names). One or multiple views can be arranged depending on the particular responsibilities or needs of the operator. So, a person responsible for the ground station, can monitor GS parameters (see example in figure 6.33), whereat a formation operator is rather interested in satellite-specific values (figure 6.34).

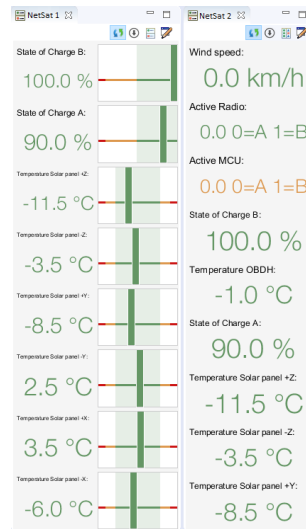


Figure 6.34: Visualization of the satellite's values

### 6.6.9 Schedule View

The Schedule View visualizes all detected TLE values in the local model knowledge. To do so it scans the entire model knowledge, finds TLE entries and creates local SGP4 propagators. Whenever a remote TLE is updated and received (subscription or manual model poll), the propagators are recreated. The name of the TLE object is the name of the parent model value – see example in figure 6.35. If multiple entries are detected with the same name, then the one with the most recent TLE value is preferred. Below the overpass rows, this view also shows all currently available *jobs* and *tasks* from the *Scheduler View*. The depth of the visualized tasks can be changed in the menu. By double-clicking on an active task, its state can be toggled between *Paused* and *Active*. In addition, if the task has some execution time constraints, the period can be resized and moved with mouse buttons.

The Schedule menu (figure 6.36) can be used to:

- Horizontally align satellite and task entries in the view
- Show tasks only up to some specific depth
- Generate future overpass report
- Change current satellite selection, i.e. add further TLEs from the known models

Name	Value	Type	Time	ID	REC
> NetSat-1				2268	
> NetSat-2				30509	
NetSat-3				49905	
TLE	1 46505U 20068V 20289.936382...	char[1...	2020/10/15 22:34:31.364	42467	●
Uplink Strength	100%	s32	2020/10/14 14:29:53.935	7073	●
NORAD ID	46505	s32	2020/10/08 10:57:56.181	38609	●
Priority (0=min)	9	s32	2020/10/07 19:32:30.432	42522	●
NetSat-4				24370	
TLE	1 46504U 20068U 20289.935949...	char[1...	2020/10/15 22:34:31.367	48484	●
Uplink Strength	100%	s32	2020/10/14 14:29:53.937	11362	●
NORAD ID	46504	s32	2020/10/09 14:56:07.333	2032	●
Priority (0=min)	9	s32	2020/10/07 19:32:32.381	34873	●
> Parking				42189	
> NetSat-EM				59992	
Default tracking fro	-3.0°	double	2020/10/07 19:32:46.182	39821	●

Figure 6.35: Auto-detected TLEs by the Schedule View

- Change the propagation period in days (past and future)
- Show durations of current overpass and duration until next overpass in the entries headers
- Hide inactive or deactivated tasks

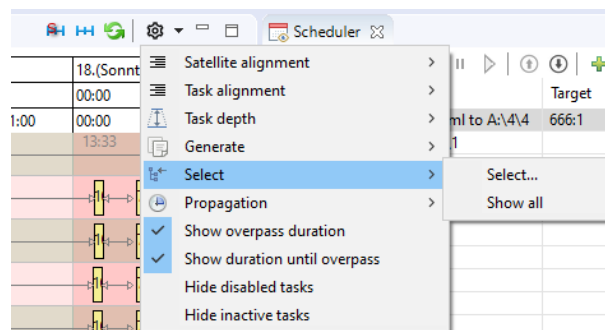


Figure 6.36: Schedule View: configuration menu

### 6.6.10 Echo View

The Echo View uses the Echo service to perform communication experiments. It is configured with: desired packet rate, payload size and the total number of packets. During the experiment all echo answers are detected and the *round-trip* time is measured, whereas both values are additionally visualized in two separate graphs. Eventually, a comma-separated report is generated, which can be used for post-processing. The report can contain additional freely selectable model values. For example, every report row can additionally contain the orientation of the antenna, radio frequencies, wind speed and distance

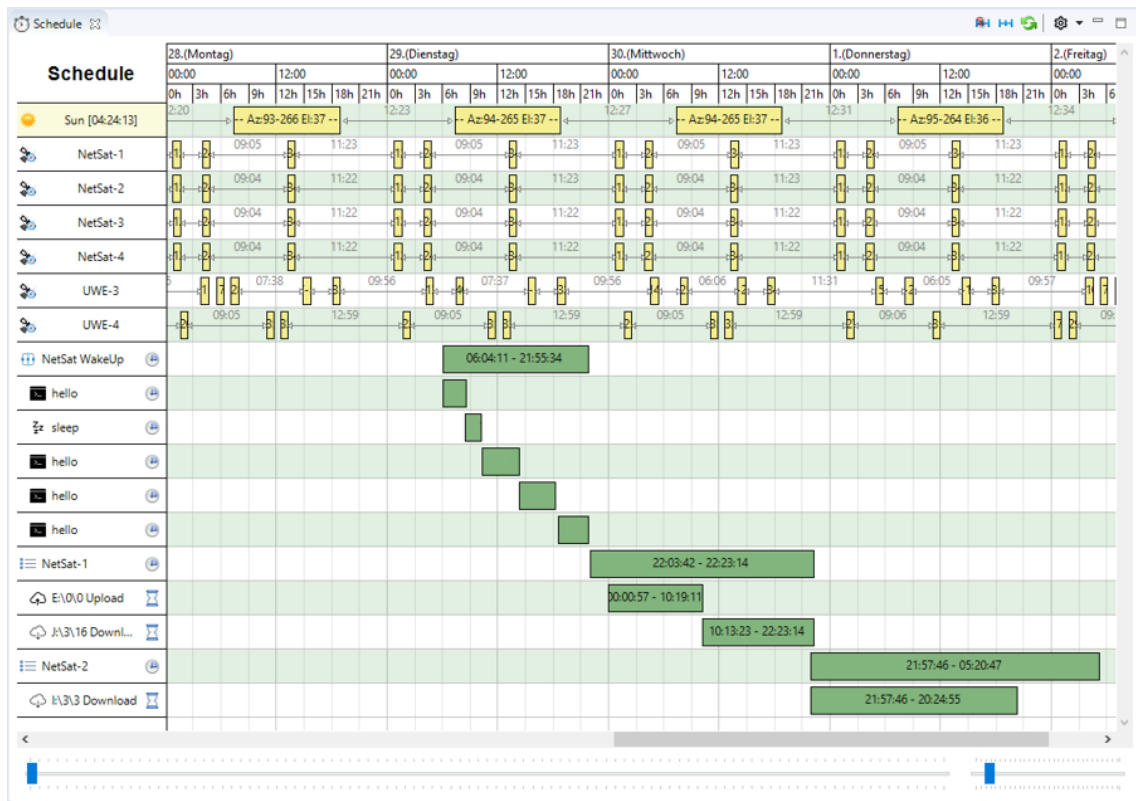


Figure 6.37: Visualization of overpasses and auto-operations tasks

to the satellite during the answer reception. These values can then be used to detect correlations between the link quality and the ground station parameters.

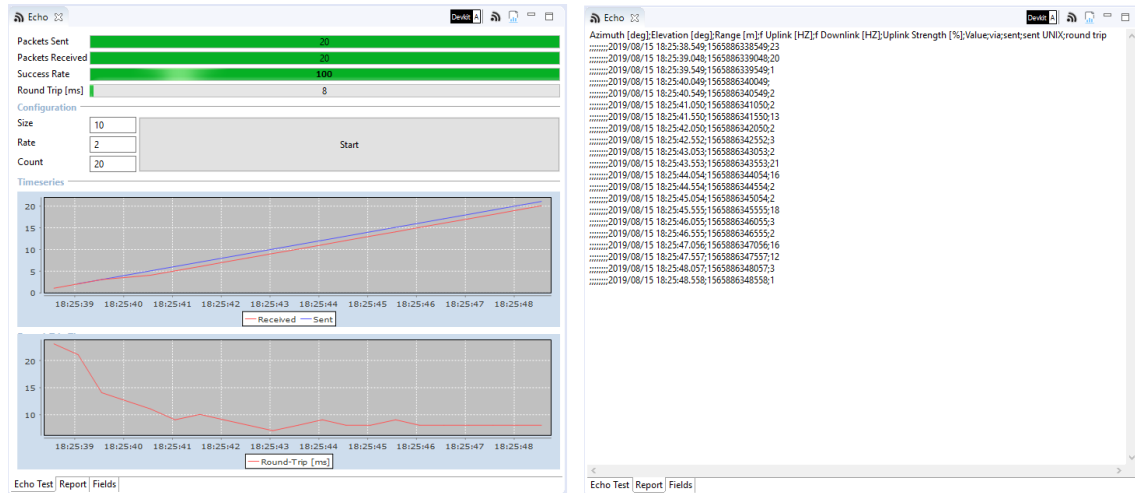


Figure 6.38: Echo View with determined round-trip times and generated report in the second view's tab

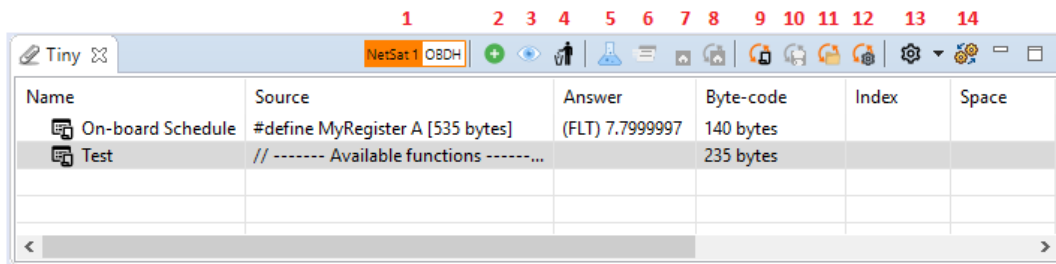
### 6.6.11 Tiny View

This view provides support for writing, testing, compiling, uploading as well as for monitoring and control of remotely running tiny scripts. For simple *one-liner* scripts the Command view is a simpler interface – Tiny View is rather designed to support more complex scripting tasks. All functions are accessed in the toolbar (see figure 6.39, from left to right):

1. Node-selector
2. *Create new script*. By clicking on this a *New Script Dialog* appears (first dialog in figure 6.40)
3. *Open Tiny IDE* for the selected script (figure 6.41)
4. *Delete selected script* (locally)
5. *Run script locally*
6. *Run script remotely*. Is only active if the script is small enough to fit in a single Compass packets (e.g. 200 B)
7. *Upload without header*. Uploads the script as-is to the remote file system
8. *Upload with header*. Same as above, but the file contains further execution parameters
9. *Create remote thread*. Generates a Script uplink with selected thread preferences (second dialog in figure 6.40)
10. *Save remote thread*. Stores selected remotely existent thread to a file (third dialog in figure 6.40)
11. *Load remote thread* from a file (third dialog in figure 6.40)

12. *Configure existing thread* (last dialog in figure 6.40)
13. *Local Tiny settings*
14. *Request a list of remote Tiny threads.*

Tiny was extensively used to perform different in-orbit attitude control algorithms on the UWE-3 satellite. On the UWE-4 satellite Tiny is used for propulsion experiments. In NetSat, TIM/TOM and CloudCT mission the language will become even more fundamental technology for supporting dynamic cooperative behavior. Even though the Tiny GUI (Tiny View and Tiny IDE) is well usable for current in-orbit and future formation tasks, improvements are continuously elaborated to provide better support in more complex Tiny-based solutions.



Name	Source	Answer	Byte-code	Index	Space
On-board Schedule	#define MyRegister A [535 bytes]	(FLT) 7.7999997	140 bytes		
Test	// ----- Available functions -----...		235 bytes		

Figure 6.39: Tiny View

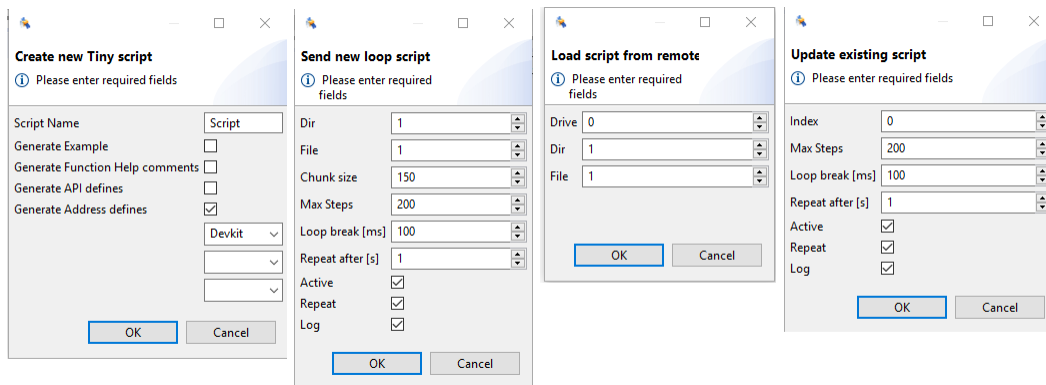


Figure 6.40: Tiny View dialogs



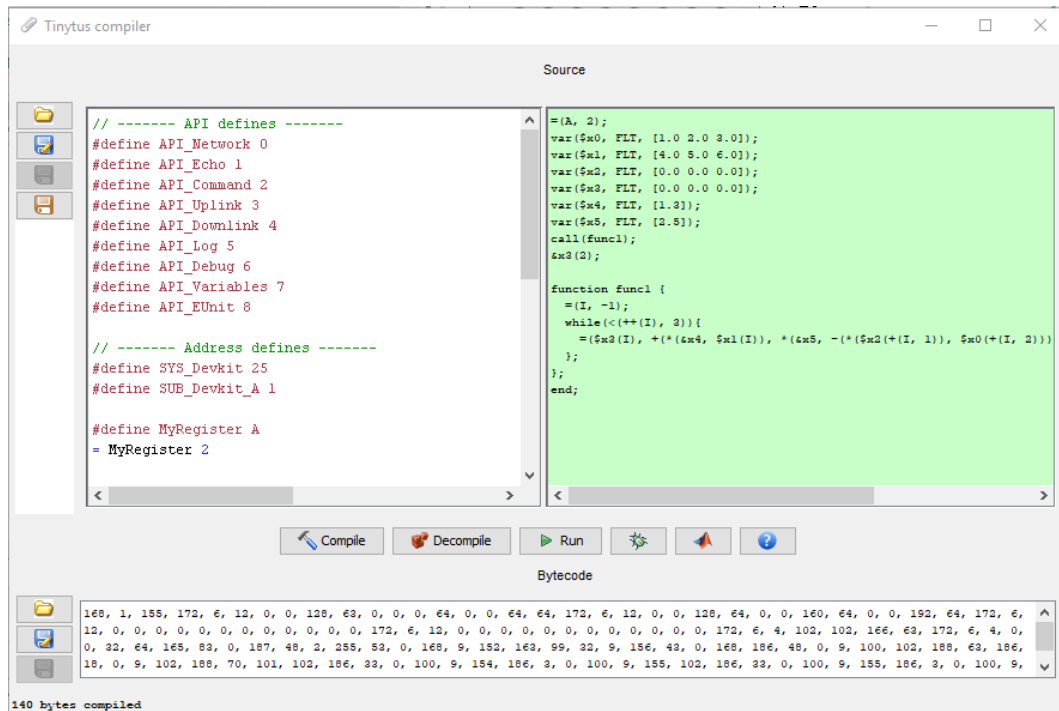


Figure 6.41: Opened Tiny compiler

## 6.7 Auto-Operations

As the last implementation of this thesis, a novel auto-operation system was designed, which drastically simplifies the creation of operations jobs for multiple satellites of the NetSat formation mission. The entire functionality is offered by the *Scheduler View* of the Compass Operations front-end. The approach is: instead of manually creating operation scripts or use some building task blocks the operation schedule is mostly *recorded*.

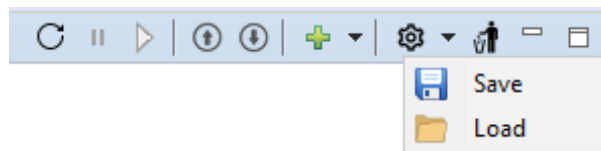
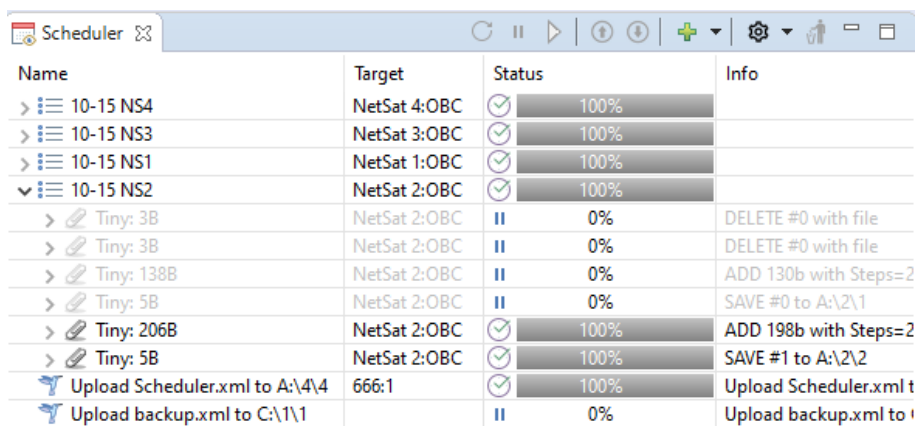


Figure 6.42: Scheduler View buttons

The Scheduler executes enabled and active tasks from top to down (one exception to that is the *ParallelJob*). A task is considered active, if the desired target node is currently available. The tasks can be grouped in jobs and sub-jobs. The view provides following buttons (figure 6.42, from left to right):

- *Reset* selected tasks, i.e. stop execution, reset states and bring it to Paused mode

- *Pause* selected tasks
- *Start/continue* selected tasks
- *Move* task up/down to change its priority
- *Add* new job or task at the scheduler's root
- *Settings*: save/load schedule to/from a XML file
- *Delete* selected tasks



Name	Target	Status	Info
> 10-15 NS4	NetSat 4:OBC	100%	
> 10-15 NS3	NetSat 3:OBC	100%	
> 10-15 NS1	NetSat 1:OBC	100%	
▼ 10-15 NS2	NetSat 2:OBC	100%	
> Tiny: 3B	NetSat 2:OBC	0%	DELETE #0 with file
> Tiny: 3B	NetSat 2:OBC	0%	DELETE #0 with file
> Tiny: 138B	NetSat 2:OBC	0%	ADD 130b with Steps=2
> Tiny: 5B	NetSat 2:OBC	0%	SAVE #0 to A:\2\1
> Tiny: 206B	NetSat 2:OBC	100%	ADD 198b with Steps=2
> Tiny: 5B	NetSat 2:OBC	100%	SAVE #1 to A:\2\2
Upload Scheduler.xml to A:\4\4	666:1	100%	Upload Scheduler.xml t
Upload backup.xml to C:\1\1		0%	Upload backup.xml to t

Figure 6.43: Scheduler View used for auto operations

### 6.7.1 Task Creation

At the time of writing (November 2020) supported jobs and tasks are:

- *Job*: contains several tasks and sub-jobs, which are executed sequentially top-down. It supports task recording function, which detects user communication and automatically creates *PacketTasks* and detected answers
- *ParallelJob*: same as job but all included tasks are executed in parallel
- *PacketTask*: send a packet. In most cases *PacketTasks* are created by the task recording function of a job.
- *ScriptTask*: locally run JavaScript
- *SleepTask*: sleep given time of milliseconds
- *TinyUpload*: performs robust (remote node can reboot inbetween) file upload by just using the remote TinyAPI

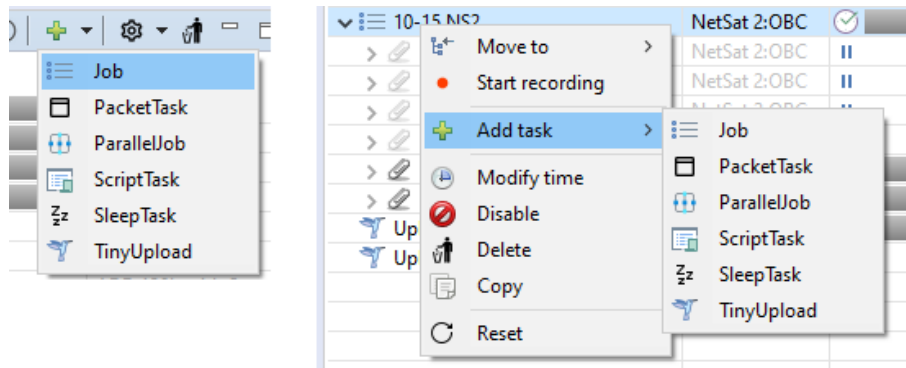


Figure 6.44: Scheduler View: task creation in the scheduler's root (left) and inside an existing job

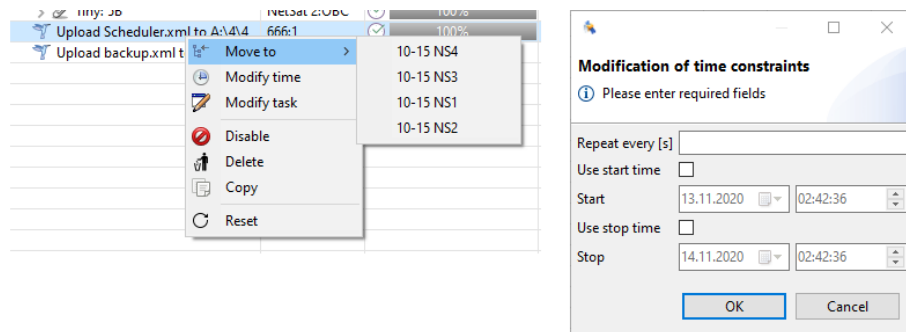


Figure 6.45: Scheduler View: task modification options

With the mouse activated context-menu, one or multiple selected tasks and jobs can be modified (figure 6.45): *move* to another job, *copy*, *disable*, *delete* and *reset* current task state. With *modify time*, re-activation of the selected task can be enabled as well as the execution time can be limited either in one or in both directions. Some tasks support finish constraints, which can be used to describe the finish-condition of a task. Currently only *Answer Constraints* can be added to *PacketTasks* to decide whether the task execution is considered finished/successful or not.

## 6.7.2 Task recording

A job can hold multiple tasks, which are either created manually, copied from another job or are generated with the *recording function*. The task recording can be activated by right-clicking on the job and selecting *start record*. After the recording has been started, all user actions in the Compass Operations front-end are recorded – more specifically the outgoing traffic is stored in the job as separate *PacketTasks* along with the detected answers, which are added to these tasks as disabled *Answer Constraints*. After the recording

these constraints can either be activated, deleted or replaced by a new constraint. Eventually the recording is stopped and tasks are modified where appropriate (change target address, rearrange, etc). Since the NetSat launch in September 2020, the vast majority of operations are performed with the auto-scheduler – more details are shown in Chapter 8.

Job	Command	Target	Status	Progress	Description
Job	Command: hello	GS:UHF Uni	II	0%	hello
	AnswerConstraint				answer equals 'Hi'
	GDS: 7B	GS:UHF Uni	II	0%	Get local Sat/NetSat-3/TLE (max
	AnswerConstraint				answer equals '1F E3 A5 BF 8C Ci
	GDS: 7B	GS:UHF Uni	II	0%	Get local Sat/NetSat-3/Uplink Str
	AnswerConstraint				answer equals '1F A1 1B 61 8F 99

Figure 6.46: Scheduler View: recorded tasks

An existing task can be modified by right clicking on it and select *modify*. Alternatively, the corresponding info-field of the task in the Scheduler View can be double-clicked to open the modify dialog. In the dialog *repeat time* and *packet bytes* can be selected. Repeat time describes the delay between consecutive transmissions if the task is constrained (Answer Constraint) and there was no match. By clicking on the *status field* in the table view, the task state is toggled between *paused* and *active*. The target of the packet can be changed by clicking on the *Target field* in the table view. The entered value is checked and shown as a legal address – so it is possible to add numeric address and see its textual representation after the entry.

### 6.7.3 File Links

New file uploads and downloads cannot be created directly in the Scheduler View – instead they are created in the corresponding File View. However, all new file links appear in the root of the Scheduler and can be moved to any job.

## 7 | Testing and Live System Experience

Due to the far-reaching influence of this work on all mission-relevant software and hardware components and limitations described in section 2.8, the integration of the implemented components in the ZfT's live system was not performed at once. Instead, all approaches were designed, implemented, tested and finally integrated in the live system in multiple *milestones*. Since this work was aimed to support nanosatellite formations, the thesis realization had to be finished *before* the NetSat launch. Thus, all of the formation-relevant component tests had to be performed individually using already existing hardware and in-orbit satellites. All *implementation milestones* were realized in this order (see Chapter 3 for more details):

- Design unification approach for space or ground system functionalities (MTBA).
- Re-implementation of the *ground station software*.
- Implement *Tiny language*.
- Design *Compass protocol* and *Compass services*.
- Start with *Compass Operations front-end* implementation.
- Implementation of middleware (*CompassNode*) for ground nodes
- Implementation of middleware (*Compass OS*) for space/embedded nodes.
- Include Compass middleware in auxiliary systems: Matlab instances, Orekit simulations, motion simulators, and satellite development kits.
- Use the completed Compass ecosystem for UWE-4 and NetSat development, verification, software testing, LEOP operation simulation.
- Combine all space and ground systems to one super-mission Compass network.
- Implementation of new *Auto Operator* to improve the outcome of simultaneously operated satellites.
- Gain in-orbit experience with UWE-3, UWE-4 and NetSat.

Throughout the work several updates and improvements were made, based on the practical experience and *colleagues' feedback*. So, the auto-operations module was completely re-designed based on the experience with UWE-3 and UWE-4 operations to drastically simplify the schedule creation.

The *agile software development* was the driving philosophy (figure 7.1) – it implies very tight collaboration with all team members and comparable early integration of developed software milestones. The update procedure was simplified, since all implementation changes were applied to only three software components:

- *Compass OS (C)*. Used by all satellite subsystems and embedded hardware components, such as e.g. satellite development kits and turntable low-level controllers.

- *CompassNode* (Java). Used by all Java-based components: Matlab, Orekit, mission server, GS server, Compass Operations front-end and subsystem emulators.
- *Compass Operations front-end*. Used by all ZfT team members to access the mission network and control components for which the person is responsible.

All components were stored in a dedicated *versioning system* server. All updates were automatically applied to all projects that rely on the mentioned artifacts. Using this strategy most of the implemented solutions could already be in practical use for years by the end of this thesis.

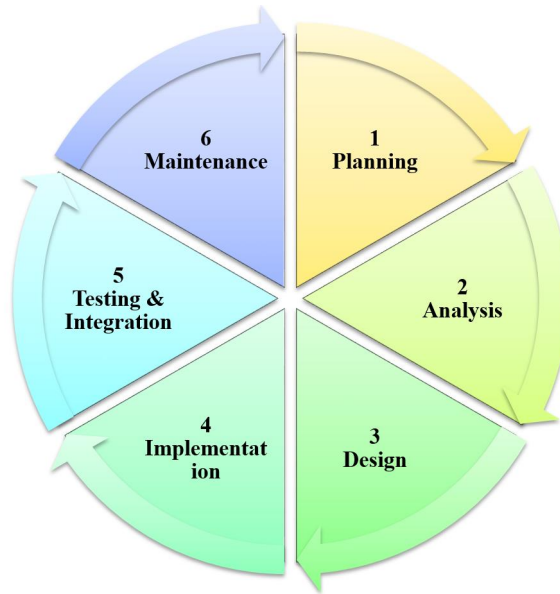


Figure 7.1: Agile Software Development

To validate the usability of the achievements for the ongoing nanosatellite formation missions, the entire path from the operator towards the desired satellite as well as the ability for cooperation between multiple nodes had to be tested. Most of the implemented technologies were tested during the agile development rounds and during further undertakings that provided proof of the applied concepts to be usable in formations (e.g. tests described in 7.2, 7.3).

In the following sections, several procedures and experiences will be shown that are transferable to specific areas of a formation mission and in total cover all formation relevant aims:

- *Protocol Performance*. Test low-level performance of the Compass implementation.
- *TOM Scenario*. Cooperation of multiple satellite subsystems.
- *UWE-4 Sensor Calibration*. Cooperation of distributed subsystems.
- *In-Orbit Experience with Tiny*. In-orbit usability of Tiny.
- *UHF Ground Station and Mission Server*. Robustness of the mission network. Ability to track and auto-operate multiple satellites.

Test	Target	Type	Speed
1	NetSat:OBC	Serial	slow
2	Local TCP (C)	TCP	fast
3	Local TCP (Java)	TCP	fast
4	ZfT:Gateway	LAN	fast
5	GS:Server	WAN	fast
6	UWE-4 EM:OBC	Radio	slowest

Table 7.1: Tests with different channel types and the theoretical speed

- *Multi-satellite operations.* Use Compass Operations front-end to operate multiple satellites.

## 7.1 Protocol Performance

The performance of the Compass protocol was tested with the Echo View (Echo service). The aim of this test was to determine round-trip times for different channel types and thereby identify the performance of the Compass stack with respect to the packet creation and *respond-ability*. More specifically, the round trip time was measured between the echo packet transmission and the answer reception. For this experiment several different connection types were used to identify the bottlenecks with respect to the packet rate (see table 7.1). To ensure the comparability of the results, all tests were performed with a packet rate of 500 ms and 10 B of random generated payload. For all tests a local instance of the Compass Operations front-end was used. The results are shown in figures 7.2 and 7.3.

### 7.1.1 Serial communication

In this test the OBC of the NetSat mission was connected locally via the serial cable (9600 baud/s). Thus, the test shows the performance of the embedded Compass implementation. The round trip time is best as compared to all other channels: *2ms* – even with the low baud-rate, this communication type provides the highest *respond-ability*.

### 7.1.2 Local TCP

Here the performance of the TCP channel was tested *without* physical lines between the nodes. In the first local test, a second C-based node was started on the *same* machine. In the second test, the target node was replaced by a Java-based node. The communication was performed in both cases using the local TCP stack (localhost → localhost). The measured mean round-trip-time in both cases was similar: *4ms* for the Java and *6ms* for the C target.

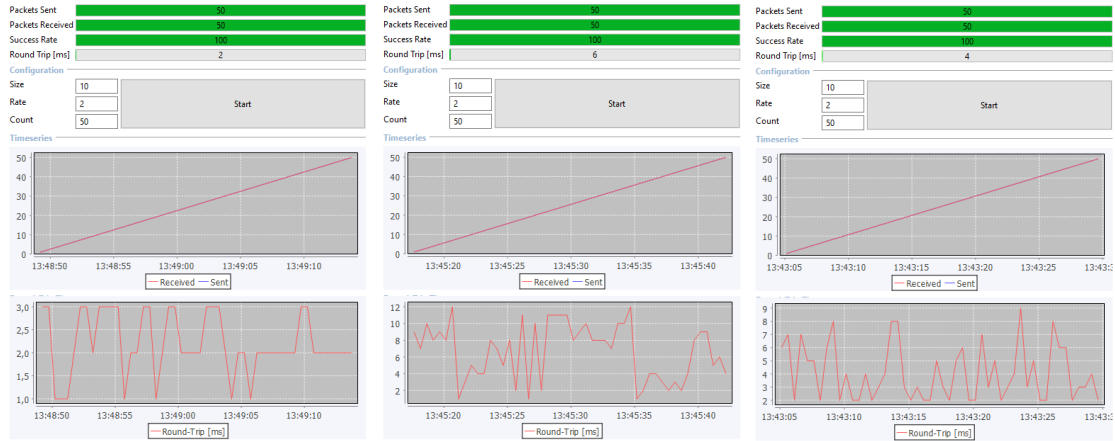


Figure 7.2: Test results: serial (right), local TCP/C (middle), local TCP/Java (left)

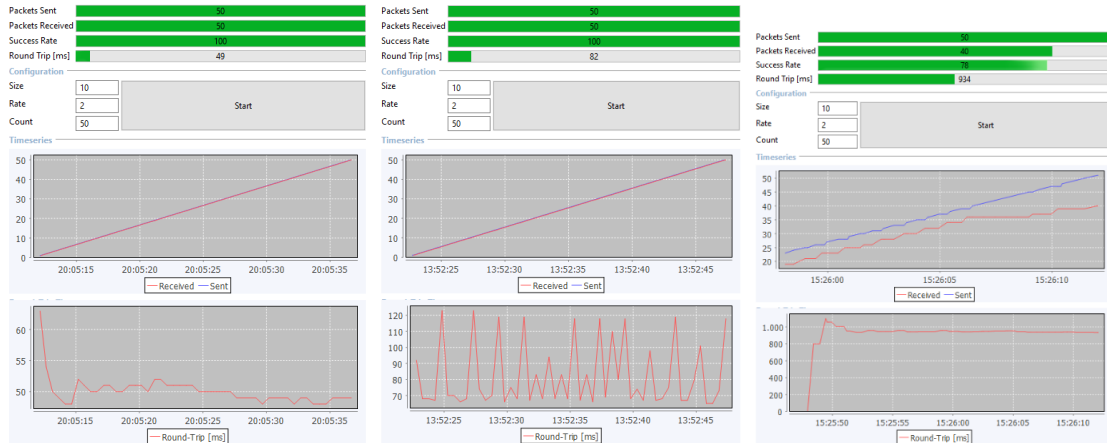


Figure 7.3: Test results: LAN (right), WAN (middle), Radio (left)



The larger trip-time, as compared to the serial communication, results from the usage of the operating system's TCP stack implementation. The performance of the Compass implementation itself – both Java and C – cannot be the source of the additional delay, since Compass does not distinguish between different channel types (channel abstraction) and both entirely different implemented C and Java versions perform very similar.

### 7.1.3 Remote TCP

With this test the performance of remote TCP channels was examined. In contrast to the local TCP test, both nodes were either connected via local area network (first test, LAN) or via Internet (second test, WAN).

In the first test the receiver was a full-fledged Compass-gateway Debian server, the measured mean round-trip time was  $49ms$ . It can be seen that the first answer takes notably longer – also in the second remote test, which is most probably the result of the TCP/IP build-up process.

In the second test the Compass Operations front-end node (located at ZfT) was connected to the UHF ground station node (located at the University) via Internet. Here the round-trip time is much higher –  $82ms$ . Possible explanations for higher delays are:

- SSH tunnel used to built-up secure TCP/IP connection.
- UHF GS node is currently powered by a Raspberry PI 3 board.
- Longer distance between the nodes with several servers in between.

Nonetheless, this configurations is currently present in the ZfT's mission network, such that the results will be considered in future decisions.

### 7.1.4 Radio Link

In this test the performance of the communication between the UHF GS server and the UWE-4:OBC was measured. The ground part of the hardware link is described in more detail in section 6.1. The round trip-time is worst among all performed communication tests ( $934ms$ ) for several reasons:

- The terminal node controller (TNC) adds a  $300ms$  *TX-delay* before send, so that the transceiver (ICOM 910H) and amplifier (BECO) have enough time to prepare the transmission.
- Continuous transmission is impossible in this test, as the radio hardware must be able to receive the answer in the same frequency band.
- The satellite's radio (tested with Astrodev Lithium-1) is considered a *black-box* in this test, as only byte-streams can be accessed via its serial interface – i.e. no further information is available about the internal processes of the transmitter

## 7.2 TOM Scenario

In the *Telematics Earth Observation Mission* (TOM), a core of 3 nanosatellites will perform coordinated photogrammetric observations from different viewing angles [Kle+19].

Test	Target	Type	mean RTT
1	NetSat:OBC	Serial	2 ms
2	Local TCP (C)	TCP	6 ms
3	Local TCP (Java)	TCP	4 ms
4	ZfT:Gateway	LAN	49 ms
5	GS:Server	WAN	82 ms
6	UWE-4 EM:OBC	Radio	934 ms

Table 7.2: Round-trip-time (RTT) results of the communication channels

One of the challenges is to achieve precise *relative orientation* of multiple satellites using the camera. All satellites concurrently take a picture of the Earth surface, detect *features* and exchange them to determine the relative angular offset. *Maros Hladky* worked on the image feature detection part and performed several in-the-loop tests[Hla18]. His software is based on the Compass OS and was running on an *Odroid* single-board computer, the performance of which is comparable to the Raspberry-based *Computing Board* used in NetSat satellite. The basic idea of the imaging subsystem is:

1. read-in camera image
2. detect appropriate features with feature-detection algorithms provided by the *Open CV* library
3. compare detected features with features from the master satellite and calculate offset
4. set control values in the attitude control subsystem depending based on the offset



Figure 7.4: 3D model of the Marienberg Fortress in Würzburg being moved on top of the *Stäbli* mobile platform

During his work two prominent tests were performed. In the first test the imaging subsystem was mounted in the ZfT's high-precision motion simulators, which substituted the attitude control subsystem by receiving all generated attitude control values followed by a corresponding axis movement. In the second test the imaging subsystem was fixed and was pointing to a moving platform with a 3D printed model of the *Marienberg Fortress* (figure 7.4). Both tests proved the usability of Compass for machine-to-machine communication and cooperative behaviour. With this project Compass was used for entirely different systems in order to build-up a complex interactive network.

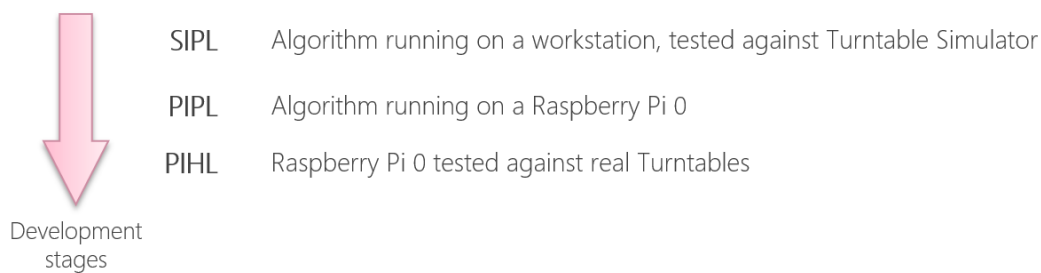


Figure 7.5: Development stages as formalized by the *modified* MDB approach from section 3.1.6

The development process was driven by the proposed *modified MDB* (Model-Based-Development) approach as described in the section 3.1.6. That is, depending on the maturity of the algorithm, it was executed either on a workstation or on hardware that is compatible to the subsystem design. The loop was either closed by using a turntable control simulator or by real turntables. Irrespective of the constellation, the communication and mutual control processes (service-based control) remained same, i.e. every node in the network could be replaced live by another implementation stage (simulation  $\rightarrow$  software  $\rightarrow$  processor  $\rightarrow$  final hardware).

## 7.3 UWE-4 Sensor Calibration

Before the UWE-4 nanosatellite was handed over to the launch provider, several tests and calibration procedures were performed with ZfT's high precision motion simulators. The motion test facility consists of two turntable actuators (see figure 7.6), low-level hardware control and two higher-level control nodes, each powered by an Atmel SAM microcontroller with Compass OS running on top. As soon as the turntables are powered on, both controller nodes become accessible in the mission network as:

- **TTU:main:** turntable with U-shaped outer frame (figure 7.6 in the foreground).
- **TTC:main:** turntable with C-shaped outer frame (figure 7.6 in the background).

In this configuration both motion simulators can be utilized for extensive test procedures, e.g. as a part of a forward-thinking rapid satellite assembly [Mar+18].



Figure 7.6: “U-shaped” (front) and “C-shaped” (back) motion simulators

During the calibration of the sun-sensors, the fully assembled UWE-4 flight model was mounted in the U-shaped turntable as shown in the figure 7.9. The ZfT’s sun simulator, which almost exactly replicates the sun’s spectrum, was placed in front of the turntable with light being pointed to its inner axis. The calibration process itself was delegated by a Matlab node (see figure 7.7) successively for all six panels:

- Set desired orientation of the satellite by changing the control values of the turntable (TTU node).
- Wait until the desired attitude is adjusted by continuously reading the current turntable orientation values.
- Read-in the sun sensor values from the currently selected panel (e.g. UWE-4:Panel +X) and add it into the list of real-vs-measured values.
- Repeat until enough measure points are collected for the calculation of the calibration matrix.

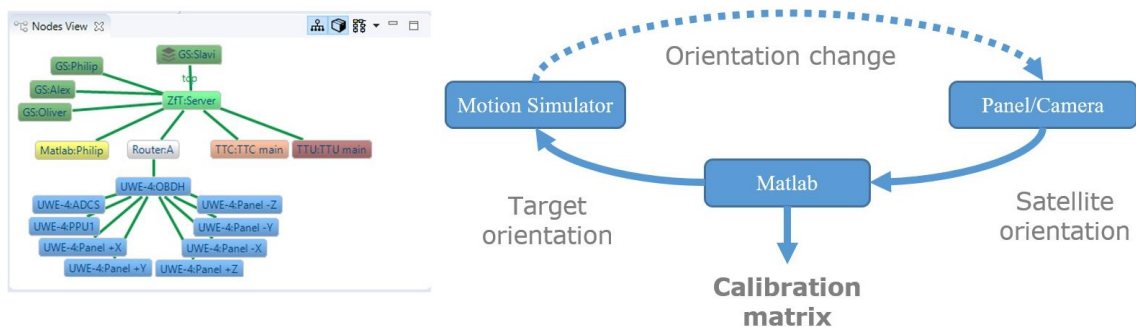


Figure 7.7: Communication loop during the sensor calibration

The entire calibration procedure was performed autonomously by three cooperating nodes, whereat the communication was based solely on the Compass model service. Even-

tually, a calibration matrix for all six sun-sensors was determined and stored in the correspondent panel. Besides the Matlab instance, all attendant team members utilized the same version of the Compass Operations front-end to observe the node of their particular interest: satellite subsystems, communication process and turntable values. Though, different view arrangements were made depending on the specific needs, e.g. additional visualization and control views for the motion simulators (figure 7.8).

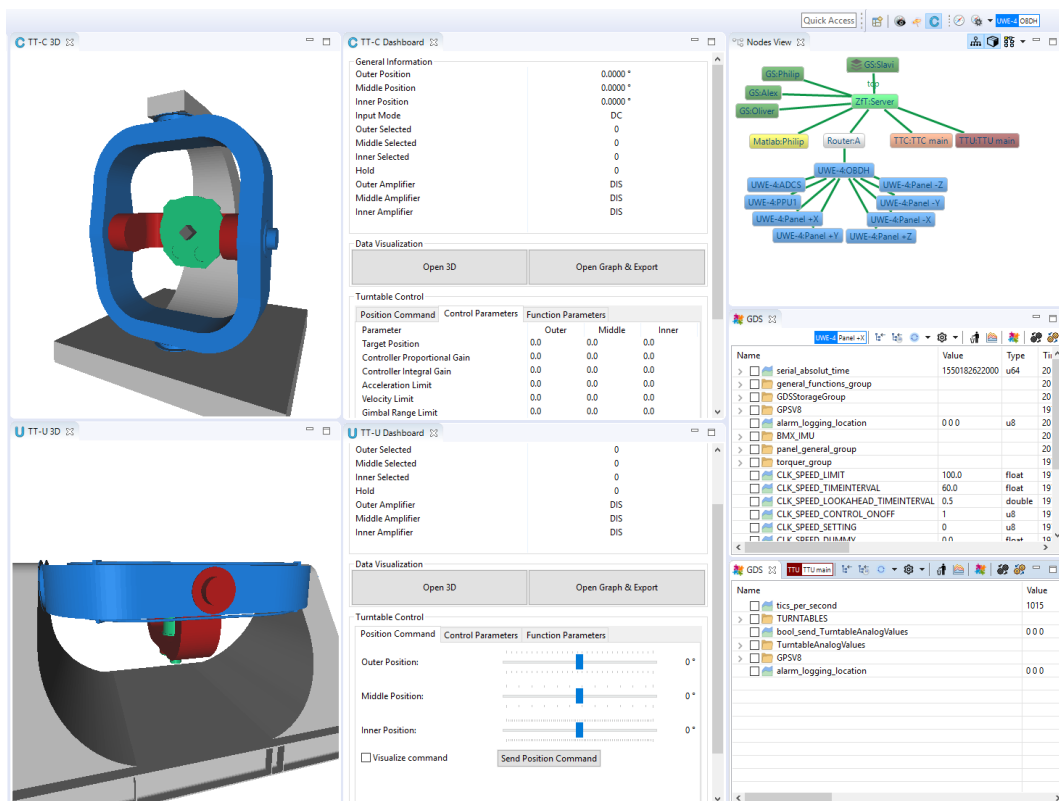


Figure 7.8: Turntable visualisation in the Compass Operations front-end with the Nodes View showing the mission network during the calibration. The 3D visualization of the turntables was developed by Oliver Ruf

In contrast to former calibration undertakings (e.g. with UWE-3), the communication *and* the mutual control was performed uniformly, thus drastically reducing the complexity and the required preparations as well as eliminating the need of “glue-code”. The process has proven the correctness of all Compass implementations (C, Java, GUI), distributed network, decentral communication process and high-level functionality of Compass services. Moreover it has proven the eligibility of Compass for the *machine-to-machine* communication and the cooperative behaviour.



Figure 7.9: UWE team performing pre-flight sensor calibration of the UWE-4 nanosatellite

## 7.4 In-Orbit Dynamic Code Execution

The Tiny language was introduced with this thesis as a main driver for the in-orbit dynamic code execution. It was designed and implemented to *run permanently* on extremely limited microcontrollers – such as the OBC in UWE and NetSat missions. The projected usage of the Tiny is:

- in-orbit execution of experiments, which can be modified on demand
- execute constraint-based actions using the model knowledge of the local and remote systems
- on-board scheduler execution

Due to the importance of this component, it was decided to design, implement and evaluate Tiny as fast as possible *before* the remaining goals of this thesis become finished (transition of all mission-relevant systems towards the new Compass-enabled approach). Thus, the implementation of Tiny was started with highest priority and could be successfully tested in-space for the first time on board the UWE-3 satellite. The following content of this section was published in the *66th International Astronautical Congress* in Jerusalem [DB15] and **is placed here as-is** with permission of *Philip Bangert* as the co-author of the publication.

First in-orbit experience with Tiny was gained after a software update of the OBDH in June 2014. In the first introduction of the scripting language on board the satellite only a few interfaces and functions have been made available to the scripts in order to ensure safe first tests.

Having shown its efficiency and reliability during this time, in a consequent update the script's functionality on board the OBDH was highly increased. Especially write and read access to the mass memory together with the string based access to the satellite's commanding interface have opened up unprecedented possibilities for advanced script based satellite operations. These features for instance allow switching between uploaded Tiny scripts based on events, time, or the satellite's state.

While the deployment on the satellite's OBDH rendered new operation modes possible, the usage in the context of flexible attitude control experiments could only be achieved when the scripts are executed on the ADCS. Therefore, in February 2015 a software update for the entire satellite (OBDH, ADCS, and six side-panels) was uploaded and installed. Since then, Tiny plays an important role in experimenting with attitude control algorithms, which will be further described in the following section.

The UWE-3 ADCS features an attitude determination based on magnetic field sensors, sun-sensors and gyroscopes, that has previously been characterized extensively as described in [BBS14a], [BBS14b], and [Bus+15]. Attitude control is achieved using magnetic torquers primarily and a single-axis reaction wheel for fast slew manoeuvres.

While the attitude determination's accuracy could be demonstrated, the attitude control's performance has been heavily disturbed by an un-modelled residual magnetic dipole moment in the order of the satellite's magnetic torquers' strength. Therefore, new attitude control algorithms needed to be developed and were updated in February 2015 to the satellite. Due to the challenge to control the satellite with limited actuation possibilities, a flexible software architecture was in the focus in order to adapt the control algorithms based on experiment results.

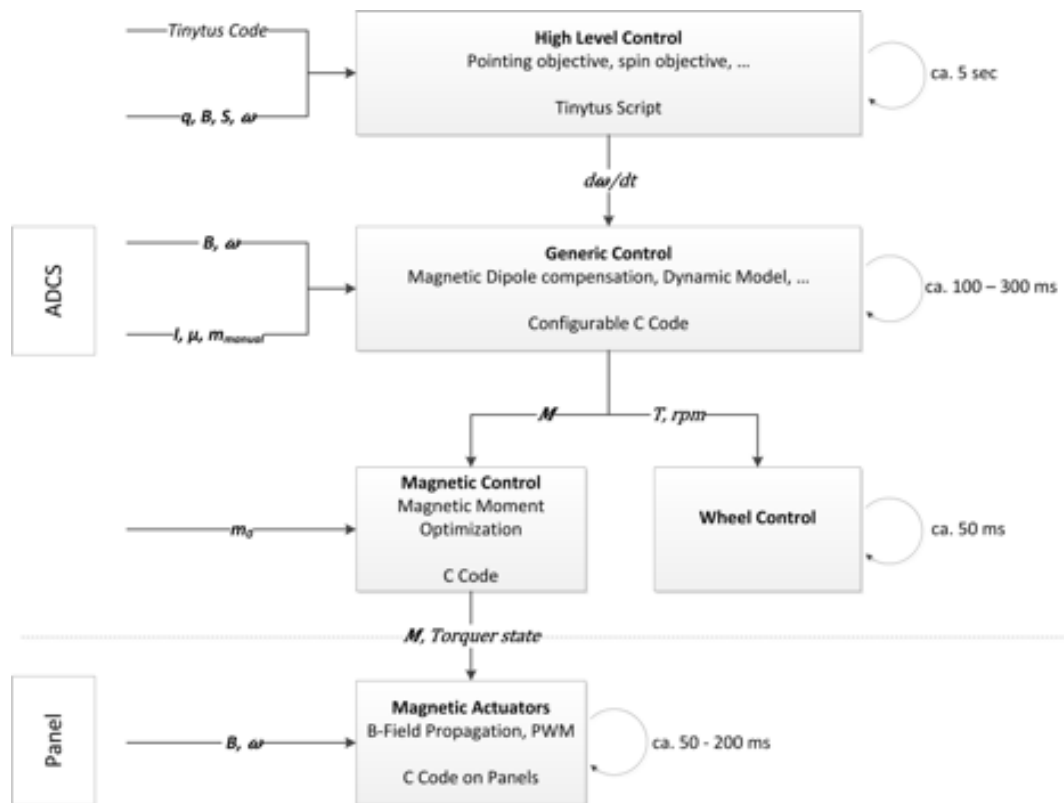


Figure 7.10: Attitude Control architecture implemented on the ADCS and the panels with appropriate timescales. Tiny is employed as flexible high-level controller. Image used with permission from Philip Bangert



Attitude control is now organized in a multi-layered architecture in order to account for the different time-scales involved in the task as shown in a schematic overview in Figure 3. In this architecture Tiny is used as high-level controller that ensures the accomplishment of a given objective by providing input in form of a desired angular acceleration to the underlying generic control law. The generic controller has dedicated knowledge about the satellite's dynamics and its residual dipole moment. Therefore, it computes the required torque to cancel unwanted dynamical behaviour and in order to achieve the angular acceleration given as input by Tiny. On a lower level, the required torque is split into a magnetic control and reaction wheel control and distributed to the according actuators.

Using this architecture, plenty of different control logics can be experimented with, such as pointing controllers, spin-stabilized, and spin-pointing algorithms. The uncomplicated exchange and safe execution of the high-level Tiny controller enables that even students can implement their controllers and experiment with the satellite.

For this purpose, Tiny scripts of up to 1kB can be uploaded into the satellite's mass memory and are then transferred to the ADCS' RAM. Each of these steps is ensured using checksums and scripts can only be executed when the checksum matches the according command. The interpreter is configured with a start time, duration, task update period, and maximum number of allowed interpreting steps. The task update period defines at which rate the interpreter is given time to operate. This further facilitates the script implementation, since precisely timed execution does not have to be implemented in Tiny but is ensured by the surrounding program.

The ADCS implementation currently features about 50 functions provided to Tiny scripts. This includes timing functionalities, basic attitude determination adapters such as reading the current quaternion from the Kalman Filter, sensor sampling and actuator activation functions, and useful mathematical algorithms (e.g. to normalize vectors/quaternions, calculate direction cosine matrices from quaternions, inversion of quaternions and quaternion products). Furthermore, all necessary adapters in order to feed the generic controller and to set the control update rates, as well as dedicated functions to record Tiny variables are implemented.

The usage of Tiny in the context of the ADCS has shown that variable declaration in the code section and specific data types such as vectors are useful extensions. In the future Tiny versions the support of code optimization before compilation will allow even more complicated scripts, for instance to make Tiny implementations of attitude determination algorithms possible. Already now it is evident, that the introduction of the highly efficient scripting language Tiny has made flexible and fast execution of attitude control experiments possible and will further be intensified in the future.

## 7.5 UHF Ground Stations

Since 2017 the ground station server is powered by a single Raspberry Pi – currently Pi 3 Model B+ with 1 GB of RAM and four 1.4 GHz cores. There are several reasons, why it was decided to do a long term test on that platform:

- The ability to have multiple low-cost spare units, which can be replaced within minutes.
- Fast and low-cost development platform for colleagues and students – a copy of the entire GS server software is obtained by copying the SD-card.

- Raise the awareness of resource-humble development.

With all the numerous tasks (tracking, database access, web server, handling of space links and hardware control) the CPU load of the server is between 4 and 20 percent, such that there was no need yet to replace the hardware with a full-fledged computer.

The ground station server acts as a ground relay and is a gateway between all ground nodes and the space nodes. The network remained stable during the last years and many improvements were implemented to optimize the connectivity: “tracking” support for satellite EM models, traffic based satellite tracking, etc. Table 7.3 shows all recorded ground-space Compass traffic during the UWE-3, UWE-4 and NetSat missions. A more detailed visualization is shown in Appendix 9.6. The comparable high uplink data traffic in UWE-3 is due to several in-orbit software updates that were also performed to test new approaches developed in this thesis: Tiny interpreter and Tiny-based experiment execution.

Beginning with UWE-4 the space-segment was entirely based on Compass-enabled nodes. During the in-orbit phase of the UWE-4 mission the ground segment was performing fully autonomous, such that a single Operator could conduct in-orbit experiments. Within only one and a half years more downlink traffic was received in UWE-4 as in more than six years in the UWE-3 mission (58 vs. 46 MB). At the same time much less uplink traffic was required to achieve mission goals (7 vs. 49 MB). That is, with the Compass auto-operations and with the Tiny-based in-orbit experiment execution the mission outcome (received in-orbit data) could be significantly improved.

With improved recording-based auto-operation capabilities that were introduced before the NetSat launch, the data outcome of the NetSat mission will be even higher. In only one and a half months of NetSat operations 8.4 MB of downlink traffic was processed – it makes 67 MB (versus 38 MB in UWE-4) per year. This is due to the fact, that the vast majority of operations are now performed with recorded operations schedules, thus eliminating human-induced breaks between the transmissions. Many operation tasks that were formerly initiated from ground (manually or by the auto-operator), were performed in-space with Tiny-based task scheduling scripts – thus reducing the required uplink communication even more.

Mission	Launch	Down	Up	Down, MB	Up, MB	Ext
UWE-3	11/2013	235'731	576'187	46	48.6	215'568
UWE-4	12/2018	323'304	240'760	57.7	7	229'052
NetSat	09/2020	80'876	77'983	8.4	3.3	
NetSat-1		15'571	18'769	1.8	0.7	
NetSat-2		38'792	23'524	3.3	0.9	
NetSat-3		17'327	18'205	2.4	1.0	
NetSat-4		9'186	17'485	0.9	0.8	
Total		639'911	894'930	112.1	58,9	444'620

Table 7.3: Ground Station Server: stored ground-space traffic for all three missions (2020/11/13). *Ext* denotes packets being received from remote stations

With the Compass dynamic routing, the server automatically propagates its ability to communicate with currently reachable satellites. The communication between the Op-

erations nodes and the desired satellites were handled entirely by Compass – also when the satellite node *jumps* from server to server as a result of a successive overflight over multiple ground stations. This is a foundation stone for a simple and *decentralized* ground station network – all reachable satellites within a larger mission network (as will be the case in TIM/TOM mission) pop-up in the network view behind the corresponding ground station nodes.

## 7.6 Multi Satellite Operations

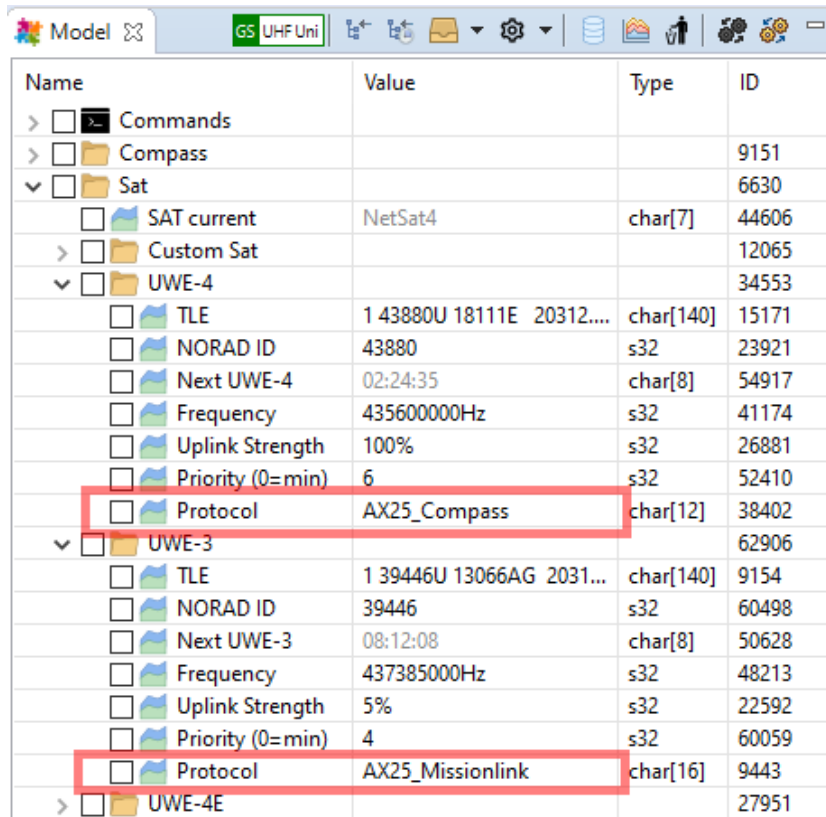
After the successful launch of the UWE-4 CubeSat in December 2018, yet multiple concurrently controllable ZfT satellites were in orbit (UWE-3 and UWE-4). So, this constellation could be used to verify the ability of ground systems to support multiple satellites – an important step before NetSat formation launch. Before the launch the Ground Station and the Mission Server were modified to support a large number of satellites with respect to:

- Prioritized satellite tracking.
- Updating Compass routing entries depending on the availability of the satellites.
- Extension of the auto-operations scheduler to support multiple satellites.
- Implementation of Tunnel service to support Compass-based communication with non-Compass satellites (UWE-3).

As described in detail in section 6.6, the graphical user interface of the Compass Operations front-end was designed in such a way, as to enable straightforward operation of multiple satellites. That is, all available views can be targeted to the desired satellite node, whereat multiple instances of the same view type can be used – for example to simultaneously visualize the model of several nodes.

Since the software of the UWE-3 satellite was developed in the pre-Compass era, it could only be operated with the *Missionlink* protocol. A *Missionlink* ↔ *Compass* bridge was implemented in the Ground Station server, which automatically converts Missionlink packets to Compass packets and vice-versa. The Ground Station server supports multiple protocol conversions, which can be selected in the *Protocol*-setting of the corresponding satellite (figure 7.11) – please approach section 6.3 for further details. Eventually the UWE-3 satellite could be accessed in the mission network in the same way as any other Compass-enabled node.

Following the *decentralized mission control* approach, the operations were performed with multiple distributed instances of the Compass Operations front-end: ZfT, University of Würzburg (Chair 7) and home-office. Every instance is configured individually with views depending on the specific operator’s interests. For example, UWE-4 team members use the Compass Operations front-end to access the satellite’s model and to create schedules for long lasting auto-operation activities (file links). In contrast to that, a person who is responsible for the ground segment uses the software to keep track of the entire mission network, view all satellite-related traffic, supervise the healthiness of all servers (Ground Station server and Mission server in particular). In addition to desktop workstations, wall-mounted smart TVs with integrated *remote-desktop* function were installed in



Name	Value	Type	ID
> <input type="checkbox"/> Commands			
> <input type="checkbox"/> Compass			9151
▼ <input type="checkbox"/> Sat			6630
<input type="checkbox"/> SAT current	NetSat4	char[7]	44606
> <input type="checkbox"/> Custom Sat			12065
▼ <input type="checkbox"/> UWE-4			34553
<input type="checkbox"/> TLE	1 43880U 18111E 20312....	char[140]	15171
<input type="checkbox"/> NORAD ID	43880	s32	23921
<input type="checkbox"/> Next UWE-4	02:24:35	char[8]	54917
<input type="checkbox"/> Frequency	435600000Hz	s32	41174
<input type="checkbox"/> Uplink Strength	100%	s32	26881
<input type="checkbox"/> Priority (0=min)	6	s32	52410
<input type="checkbox"/> Protocol	AX25_Compass	char[12]	38402
▼ <input type="checkbox"/> UWE-3			62906
<input type="checkbox"/> TLE	1 39446U 13066AG 2031...	char[140]	9154
<input type="checkbox"/> NORAD ID	39446	s32	60498
<input type="checkbox"/> Next UWE-3	08:12:08	char[8]	50628
<input type="checkbox"/> Frequency	437385000Hz	s32	48213
<input type="checkbox"/> Uplink Strength	5%	s32	22592
<input type="checkbox"/> Priority (0=min)	4	s32	60059
<input type="checkbox"/> Protocol	AX25_Missionlink	char[16]	9443
> <input type="checkbox"/> UWE-4E			27951

Figure 7.11: Protocol settings for satellite modules in the Ground Station server software (section 6.3)



Figure 7.12: Wall-mounted monitor in all ZfT's offices as a part of the distributed mission control approach

all ZfT's offices. The TVs are connected to separate virtual machines running Compass Operations – as exemplary shown in picture 7.12.

Even though a separate *mission control center* room is currently in progress, the distributed approach remained the main control tool, as all team members continuously remain in the *satellite operation loop*. From experience with the UWE-3, UWE-4 and NetSat missions, which were conducted by small teams – whereat every team member had multiple duties, a distributed mission control system has significantly increased the outcome and the quality of the satellite operations.

With two active satellites in orbit, multi-satellite operations could be performed:

- Arrange views in the Compass Operations front-end to access multiple satellites.
- Monitor the status/healthiness of multiple satellites (Model View, Monitor Views).
- Perform fast command execution.
- Manual file downlink and uplink.
- Create auto-operation tasks for multiple satellites.

Only one UHF Ground Station was used for uplink, thus if multiple satellites were passing at the same time, the one with higher priority was selected for tracking and became available in the mission network via the GS node. The auto-operation task in Compass Operations scheduler became active, as soon as the desired satellite appeared in the mission network. In that configuration only one satellite could be operated *at the same time* – both manually or via scheduler. In future multiple UHF ground station nodes will be utilized, such that theoretically multiple satellites can be operated at the same time.

In the NetSat formation mission multiple satellites use the same frequency and are initially close enough to each other to “fit” into the beam of the Ground Station UHF antenna (21 elements, 18.2 dBi, ~21°-23° angular beam width). Thus, after NetSat launch multiple satellites can be served with broadcast packets using only one directed antenna. The configuration of all NetSat satellites in the GS server has the same priority, thus all of them can be made accessible in the mission network at the same time.



## 8 | NetSat Experience

All four NetSat 3U satellites have been successfully launched in 28 September 2020 by Sojus rocket from Kosmodrom Plessezk, Russia. Before launch all approaches and components developed in this thesis were tested in-depth during the precursor mission, such that most of the implementations, the mission network and all participating Compass nodes were already running for years in the ZfT’s live system.

### 8.1 External Tests and Verification

The assembly of all four NetSat CubeSats was finished in May 2020. Besides the verification tests performed at ZfT, the engineering and flight models were also tested at external facilities:

- Temperature regulated *vacuum chamber* tests: *iABG Industrieanlagen Betriebsgesellschaft mbH* in Munich
- *Acceptance shaker test*: *EXO Launch* in Berlin
- *Propulsion system tests*: *IRS*, University of Stuttgart.

During most of the tests the satellites were connected with serial cables to the Compass Operations front-end. The connection was used to access the internal values of all subsystems to monitor temperatures, power consumption and possible communication problems in the satellite (e.g. due to high currents during the active propulsion).

With Compass the test execution and outcome was dramatically enriched as compared to the pre-Compass era. So, the entire traffic between the satellites and the front-end nodes was continuously recorded using the build-in record function of the Compass Operations software (described in section 6.6.3). The recordings can easily be imported in the *Packet View* to reproduce the visualization of the satellite’s state at some specific time. Furthermore, the *auto post-process* function, which can also be activated in the Packet View, can be pointed to a recording to generate a traffic report: summary, log file with all detected Log service entries and separate CSV files for every model element containing all data changes (see 8.1).

Another advancement of Compass was the ability of test conductors to additionally connect their Operations front-ends via mobile connection with the “home network” . Due to organizational reasons (and the COVID-19 situation) not all experts were able to support the test procedure on site – but they were able to access remotely connected satellites and supervise all subsystems during the test. Figure 8.2 shows the Operations front-end of an expert at ZfT accessing the NetSat EM model at IRS. In this example

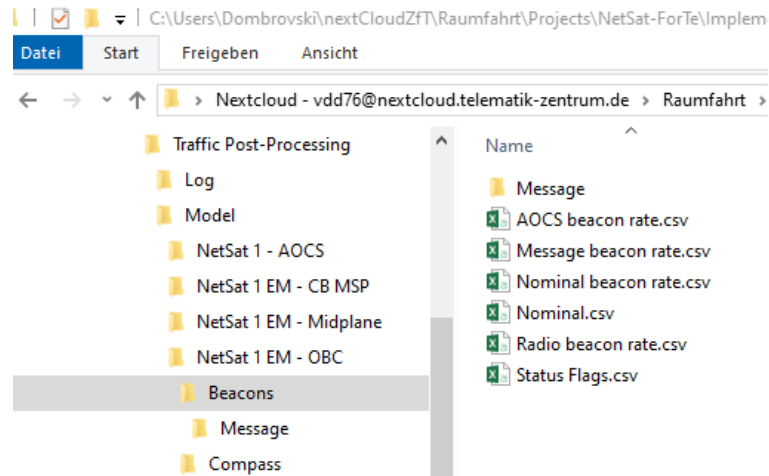


Figure 8.1: Browsing the report directory (**Traffic Post-Processing**) created by the automatic post-processing function of the Compass Operations front-end

the **Fire:1** Compass gateway (used to offer access to external nodes) was temporary disconnected from the remaining mission network for security reasons.

## 8.2 Pre-flight LEOP exercises

Before the satellites were handed over to the launch provider, numerous tests were performed on component, subsystem and system level. Eventually *LEOP simulation* was conducted multiple times, that is all flight models were brought to the final *flight mode* (no wires, antennas rolled-up) placed on separate desks – as shown in figure 8.3 – and the *remove-before-flight* pins simultaneously removed. The testing team was composed of: one operator per satellite, one reporter and one operations supervisor. The operators and other observing team members were connected to the mission network with separate Compass Operations front-ends, i.e. all front-end nodes were connected to the Mission server, which in turn had a connection to the UHF ground station node.

The deployment of all antennas started after the mandatory 30 minutes. From then on the satellites were waiting for the arrival of the first ground packet via the radio channel. After the satellites have received a broadcast command from an operator (simple **Hello** command), the downlink on the satellites was permanently activated leading to the transmission of network and housekeeping model beacons. With every network beacon being received by the UHF ground station, the corresponding satellite appeared in the mission network and became available for operations. Figure 8.4 shows the whole mission network after all network beacons were received by the ground station:

- The Mission server node is shown in the middle with all front-end nodes (upper-right area) being connected to it



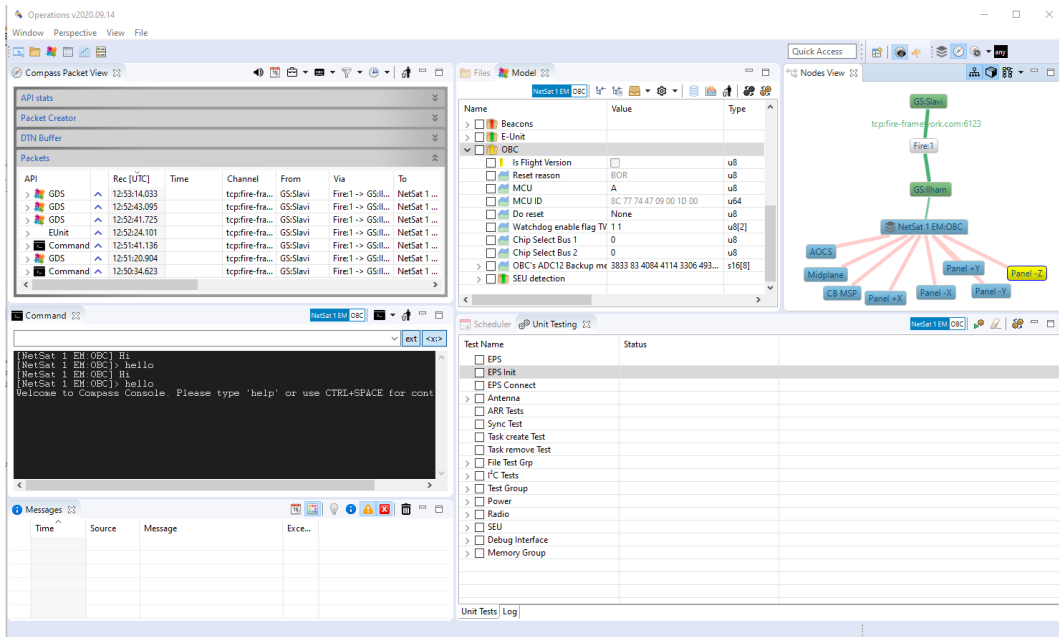


Figure 8.2: Operations front-end of an expert at ZfT accessing the NetSat EM model at IRS

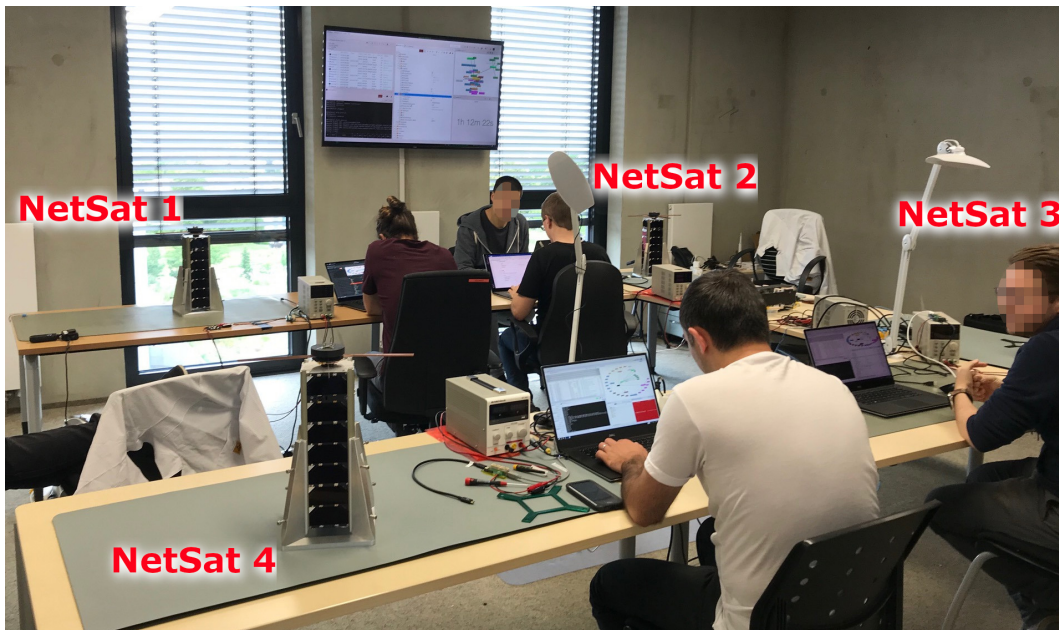


Figure 8.3: Pre-flight LEOP exercise: all four NetSat satellites shortly after the antenna deployment

- The ground station node (black, since it was blinking to signalize packet transmission) is accessible by the Mission server
- The satellite nodes are shown in the outer areas: NetSat 1 (orange), NetSat 2 (red), NetSat 3 (pink) and NetSat 4 (blue)
- Active satellite subsystems are perceptible with green lines

During the exercise the Model and the Command service were used to check the internal states of all subsystems as well as the overall satellite behavior: auto-activation of subsystems depending on the current mode, power consumption, inter-satellite communication. In addition, all file-based (Link service) tasks were tested in-depth: software image upload to the OBC for *all* subsystems with subsequent software update execution, maintenance of remote drives, download of large files from all subsystems with storage capabilities (OBC, Panels, Computing Board) and so on. By the end of the tests the feasibility of the Compass Operations front-end, ground station node, mission network and all Compass services for all imaginable tasks was ultimately consolidated and the satellites were ready for launch.

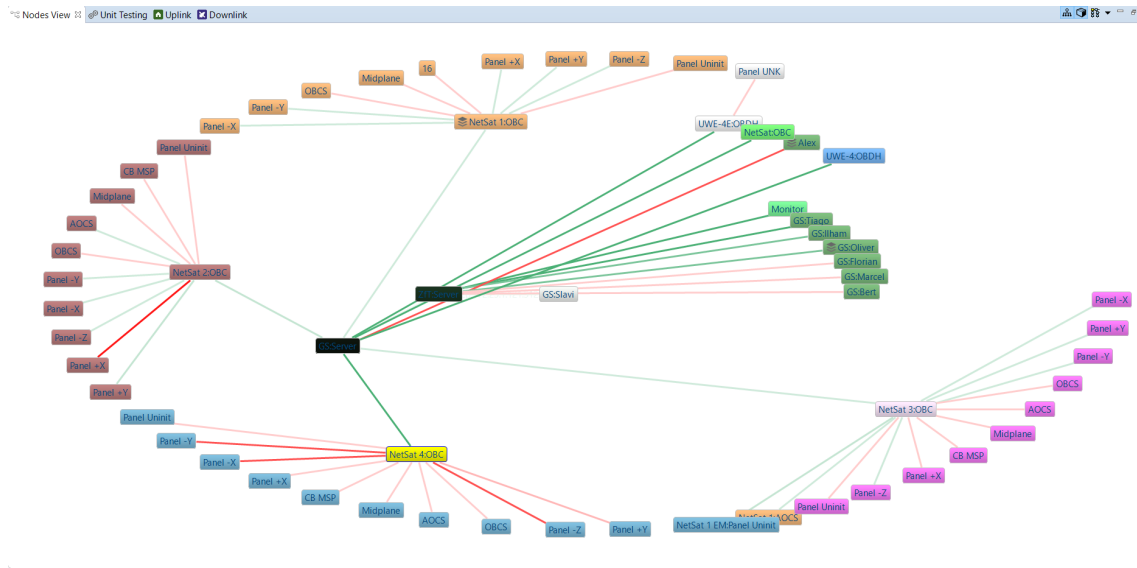


Figure 8.4: Pre-flight LEOP exercise: appearance of all four NetSat satellites in the Node View after beacon reception

### 8.3 Launch and Early Orbit Phase

The NetSat operations were started one day after the launch on 29 September 2020. At that time all satellites were very close to each other, thus the beam width of the UHF ground station antenna could enclose the entire formation. During the first operated



Figure 8.5: All four NetSat CubeSats ready for launch

overpass a Hello command was uplinked as a broadcast packet, which permanently turned on the downlink capabilities on all four satellites. Initially an unnamed TLE list of all satellites from the launch event was provided. So, during the first week all feasible TLE combinations were tried to properly map the NORAD-IDs with the corresponding NetSat. To improve the downlink capabilities, an additional SDR radio node was installed and connected as remote Compass-enabled radio node to the ground station server. As has been shown by Tim Horst in [Hor+20], the recorded RF traffic during the first days could be used to identify the Doppler shift of the downstream, which in turn was used to find a proper mapping between the provided TLEs and the satellites.

As has been described in more detail in section 6.3, the implemented antenna tracking and radio packet encoding of the Compass ground station server was made *traffic-sensitive*. During the formation overpass all satellites were handled with same priority and the first ascending one was *initially* selected for antenna tracking. By processing the outgoing Compass packets, the GS software detects required hardware parameters *for every packet*: antenna orientation, uplink frequency, Doppler shift and protocol configuration. Presupposed the satellites are close enough, all hardware parameters can be changed fast without problems, thus allowing disordered operations of multiple satellites at the same time.

Two weeks after launch, the NetSat formation became wider and it became necessary for the ground station server to move the antenna between the single satellites, thus increasingly baffling *disordered* satellite operations. Since the vast majority of the operations in NetSat are performed with auto-operations capabilities of the Compass Operations front-end, all required tasks are grouped in satellite-specific jobs, i.e. during the overpass first all tasks for NetSat-1 are executed, then for NetSat-2 and so forth.

Unfortunately on 27 October 2020 the contact to NetSat-1 was lost and could not be recovered yet. Since the software on all four satellites is identical and the remaining

satellites are still accessible and perform well, most probable explanation is a hardware failure.

## 8.4 Auto-Operations

As soon as the orbital elements for all formation satellites were finalized and verified, the operations team could start with *ordered* operations. The auto-operations scheduler is accessible in the *Scheduler View* of the Compass Operations front-end. In general any node of the mission network can be auto-operated – though to date it was only actively used for space nodes. During the NetSat operations following *modus operandi* has proven itself:

1. Activate *NetSat EM model* in the ZfT's lab, such that it becomes accessible by the UHF ground station
2. Create new Job in the Scheduler View
3. Select the new Job and activate recording
4. Perform all desired operations with NetSat EM as target, e.g. send commands in the Command View, change its model values in the Model View, create Tiny threads and so on.
5. Stop recording and change the target of the Job from NetSat EM to the desired satellite – e.g. to NetSat-1
6. If the same operations are required for other satellites, copy the Job and rename the target or create a new Job and copy single tasks.
7. Optionally change the time constraints of the Job execution in the *Schedule*
8. Store schedule to a XML file on a common network drive for possible later use and retraceability.

The recording function does not require the target node to be available. Using the engineering model during the recordings has the advantage, that also the satellite answers are recorded and inserted as *Answer Constraints* into the corresponding tasks. During the recording all auto-generated tasks appear live in the corresponding Job, for which the recording was activated. As soon as the satellites that are specified in the *Target* field of the jobs/tasks become available, the scheduler activates the tasks one-by-one. A task is considered finished, if it does not have any constraints or the defined constraints are fulfilled, e.g. an answer was received and was matched by the answer constraint. As soon as a task is finished, the scheduler activates the next one and so forth.

A simple example of recorded tasks is shown in figure 8.6. For NetSat-4, for example, the remote Tiny thread with ID=0 will be replaced with a new one, followed by a download of a J:\1\1 log file and eventually the log file will be deleted.

Over time a set of former operation jobs was aggregated, such that many higher-level tasks (i.e. tasks with multiple sub-tasks) could be copied in the *Scheduler View* into a new schedule. So, for example, tasks exist for: GPS experiment execution, download all current log files and delete them afterwards, activate Linux-based Computing Board and perform in-orbit processing, etc. To speed up schedule creation, operators leave processed jobs in the schedule and deactivated them instead of deleting them. This way new operation jobs can be populated fast with tasks copied from the deactivated ones.

In theory multiple Compass front-ends can perform auto-operations at the same time. Nevertheless, the operations team decided to use one discrete front-end for auto-operating NetSat satellites.

## 8.5 Dynamic Code Execution with Tiny

In the realization roadmap of this thesis, the Tiny interpreter was the first implemented component. The decision was motivated by the initially poor performance of the UWE-3 mission. As a consequence of this, the interpreter became an enabling technology for three missions. Tiny was used in UWE-3 and UWE-4 to perform in-orbit experiments, thus leading to achievements of mission goals. *Philip Bangert* describes in his PhD thesis, how Tiny was used to design and execute several attitude determination and control algorithms on board the UWE-3 1U nanosatellite[Ban20]. In the follow-up UWE-4 1U nanosatellite mission, *Alexander Kramer* used Tiny to successfully perform NanoFEED propulsion experiments[KBS20].

The main advantage of Tiny is – as the name implies – its tiny size in the program memory and very compact byte code used for the execution. It was specifically designed to enable its execution on very low-powered 16 bit microcontrollers – such as the OBC on all UWE and NetSat satellites. Since the OBC is the only subsystem that is guaranteed to run permanently, it has been selected as a primary host platform for Tiny interpreter. The interpreter supports parallel execution of multiple scripts in a sand-box on a *single-threaded* system, without affecting the system’s respond-ability. The functionality of the Tiny interpreter is accessed via the *Compass Tiny service*.

On NetSat the interpreter is available on all subsystems – except the *Thruster Control* subsystem. On the OBC it has been configured to support up to 5 *persistent* Tiny threads, i.e. up to 5 scripts can be executed in parallel. Persistent scripts are automatically loaded on boot-up, which turned out to be an essential feature during the NetSat’s LEOP phase.

Besides its baseline features, listed in section 5.5.1, Tiny was equipped with *external functions* used to access local and remote model values, generate user-defined Compass packets and to perform local file operations (see table 8.1). Due to the drive abstraction of the Compass OS, Tiny can be used to write data to remote files, which is of particular interest for the AOCS system (since it does not have own file storage).

During the early operations, several undesired behaviors were detected on all NetSats that either were not observed or were overlooked during the test phase on-ground. One solution was to perform in-orbit software updates on all four satellites, which in turn would require long-lasting file uplinks to all four satellites using only one UHF ground station. Since most of the OBC’s capabilities are controllable via its model (e.g. by using the Compass Model service) and Tiny scripts can be used to control local and remote models, a preliminary *bug-fixing* script (figure 13 in the Appendix) could be designed and uploaded to all four satellites. Since the size of the compiled script was only 97 bytes, it could be delivered via Tiny service with only one packet and configured as persistent with a second one. The scripts will be removed as soon as the first on-orbit software update is performed.

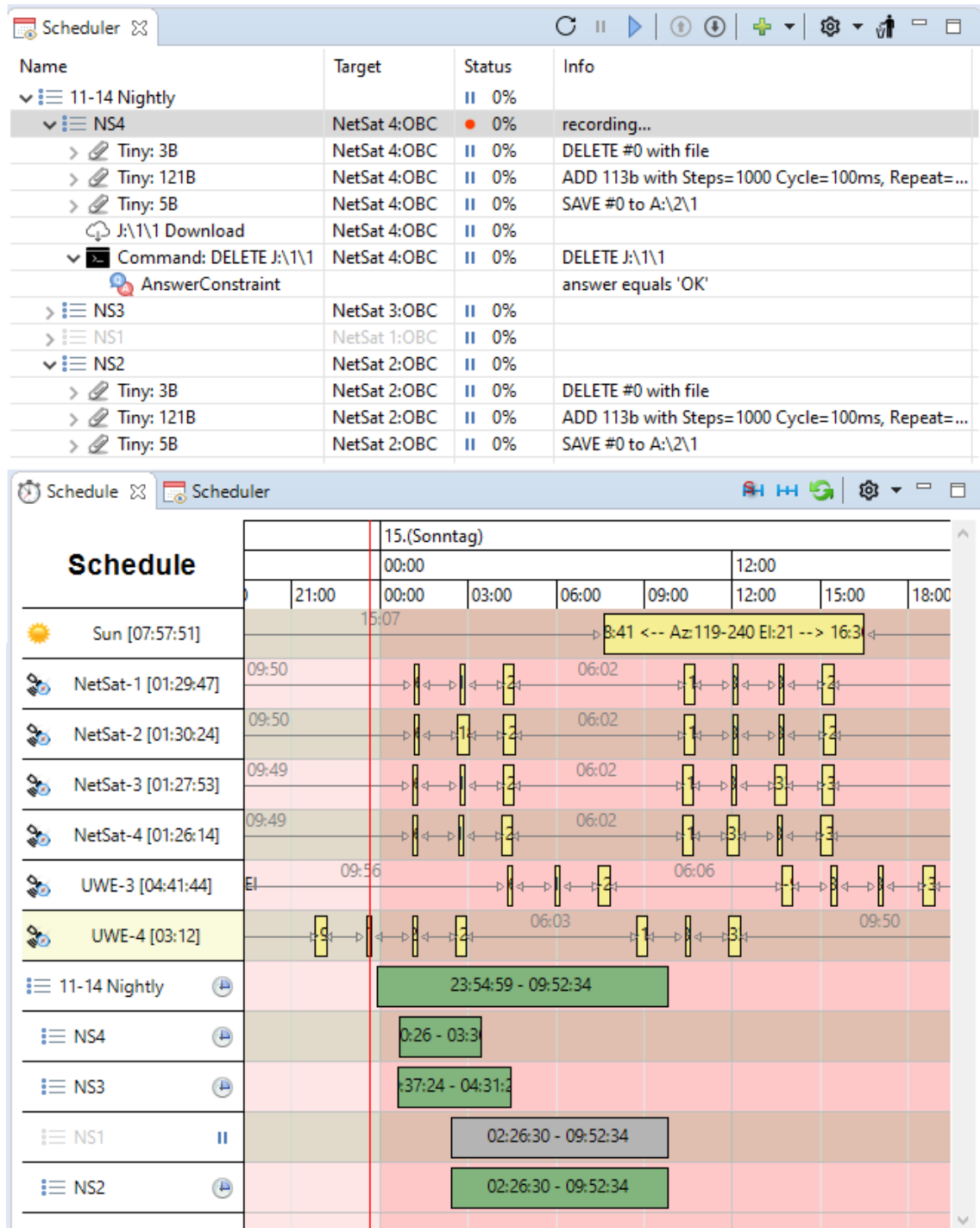


Figure 8.6: Top: Real-life example of auto-operations recorded for all NetSat satellites (NS4 job is shown in recording mode). Bottom: Visualization of all overpasses and auto-operations tasks

Function	Description
<code>gds_get</code>	read local model value
<code>gds_set</code>	set local model value
<code>gds_getRemote</code>	read remote model value
<code>gds_setRemote</code>	set remote model value
<code>send</code>	send compass packet to a target
<code>command</code>	execute command locally or remotely
<code>file_delete</code>	delete file
<code>file_write</code>	write to a file
<code>file_read</code>	read from local file

Table 8.1: Tiny interpreter: available external functions on all UWE-4 and NetSat subsystems

In NetSat, the Tiny interpreter is also used to execute constraint-based schedules. During the LEOP phase multiple scripts were designed to switch on and off multiple subsystems during several orbits and periodically create reports, e.g. results of the I2C bus scanning or power consumption of subsystems during the specific time periods. The logs were then automatically downloaded by the auto-operator and were used to evaluate the overall system performance. It became evident that Tiny is increasingly used to execute actions that are otherwise conducted from ground. To date (November 2020) several ZfT experts are designing Tiny scripts to:

- evaluate the performance of thrusters
- improve the performance of the AOCS subsystem
- execute GPS experiments (e.g. measurements during different satellite orientations)
- establish cooperative behavior between the always-running Tiny interpreter and the on-demand switchable Linux-based Computing Board (e.g. to test the Goal-based Operations approach – as has been proposed by *Tiago Nogueira* in [NFS17]).
- perform in-orbit experiments to support on-going nanosatellite missions

So, apart from the mission goal achievement, NetSat will serve as a multi-satellite testing platform for software approaches.

## 8.6 ISL

On the satellite level (intra-satellite) the cooperating behavior of several subsystems became a standard process. With the Compass OS the complexity of the satellite software development was dramatically simplified, such that most cooperative tasks could be implemented based on the Compass Model service (*model-based communication*). Some examples for model-based cooperative processes are:

- all panels periodically promote their temperature values to the OBC
- panels with activated GPS periodically send current measurements to the AOCS
- the AOCS accesses all panels to read current magnetic field and sun-vector measurements and to control magnetorquers



Due to Compass, from the software point of view, the only difference between the intra-satellite and inter-satellite model-based communication is the system-address used in the corresponding calls. In NetSat AOCS is the only subsystem that is intended to use inter-satellite (model-based) communication to perform the formation control. However, the formation control could not be activated yet, as the requirements could not be fulfilled (yet). So, for example, thrusters, GPS sensors, sun-sensors and gyros are either:

- newly developed
- used for the first time in-orbit (by ZfT)
- were never used in that depth

Thus, all these components need first to be characterized *in-space* before a satellite can autonomously control its orbit. As soon as every satellite is able to autonomously or semi-autonomously change its orbit, the cooperative formation control can be activated.

The only difference between packets from different Compass services is the packet payload. Thus to prove the eligibility of the Compass protocol to carry out inter-satellite communication of cooperating processes, it is sufficient to show that the packets from any service could be successfully transmitted between the satellites. For this task packets from the Network service, more specifically network beacons, were selected to show proof. All satellites periodically (every minute) broadcast network beacons, which contain the current network knowledge of the sender. If one satellite receives a network beacon from another satellite, it automatically updates its internal routing table with entries included in the beacon. During the overpass network beacons from satellites are also received by the UHF ground station and stored in the traffic database. Since the Compass network is self-configuring, sometimes when only one satellite was accessible by the UHF ground station (e.g. during the ascension phase of the first satellite of the formation) other satellites were shown in the mission network “behind” the active one. So, with the first network beacon the active satellite informed the ground station, that it either had a currently active link to another formation nodes or it had it in the past. The example in figure 14 in the Appendix shows an active connection to NetSat-4 via NetSat-2 during the active tracking of the latter. The satellites were close enough for ISL but distant enough to not fit into the UHF antenna beam at the same time.

The self-configuration of the Compass network relies on network beacons, i.e. if the downlink is weak and no network packets were received from the satellites, they do not appear in the mission network. For this reason the Compass ground station server supports the auto-creation of active routing entries for all currently passing satellites. That is, as soon as a Compass-enabled satellite is *theoretically* reachable by a ground station, it appears in the network as an active (= accessible) node. Due to occasional communication issues with the UHF ground station, this functionality was activated. The downside of this approach is the de facto deactivation of ISL forwarding for *ground*  $\rightarrow$  *satellite* packets. Nonetheless, network beacons that have been received and stored in the traffic database could be used for post-processing to visualize ISL accessibility of the formation nodes.

Over 4500 received network beacons were processed to detect ISL activities. An example of a satellite network beacon is shown in figure 8.7. The results are shown in figure 8.9 and the shape of the formation at different time points is depicted in figure 8.8. Every graph shows whether the corresponding satellite has *directly* received packets from other satellites within the last 60 seconds before the overpass. Multi-hop availability is



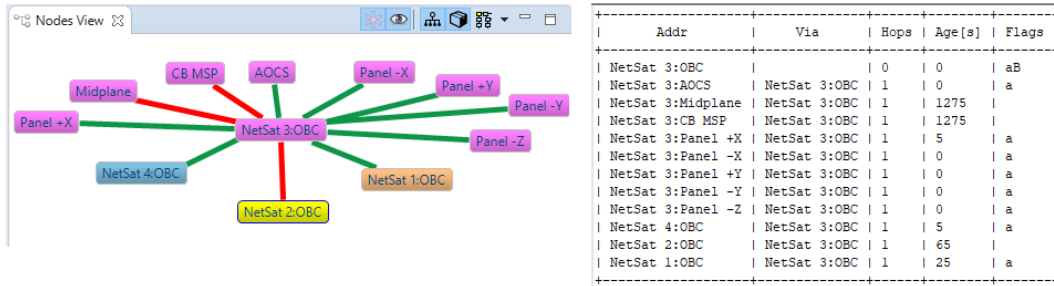


Figure 8.7: Left: Visualization of NetSat-3 network knowledge in the Operations front-end by using the received network beacon (2020/10/10 09:59). Right: Contents of the same beacon converted to human-readable form

not shown in the graph for the sake of simplicity. Since the satellite network beacons were only received by the ground station during the overpasses, the data available for the post-processing is limited to time ranges shortly before and during the satellites became accessible from ground. Nonetheless, the derived data samples show the general ISL ability at different time points and can be used for comparison. The graphs do not contain *false positives* but may contain *false negatives*: if no ISL was performed between two particular satellites during the last 60 s before the overpass (i.e. before a network beacon was received by the ground station), it is still possible that packets were successfully transmitted between these satellites shortly before. The following can be derived from the graphs (please use figure 8.8 to compare the shape of the formation at different time points):

- Until 19-21 October almost all satellites could directly communicate with each other.
- From 20 October NetSat-1 was not received by other satellites anymore – nor was it received by the ground station.
- From 8 November NetSat-2 has not received packets from other satellites – but is still received sporadically by NetSat-3 and NetSat-4. At this date the distance between NetSat-2 and the nearest active satellite (NetSat-3) surpassed the 1000 km mark.
- NetSat-3 and NetSat-4 have both to date working ISL communication to each other (~790 km distance at the time of writing).

## 8.7 Outlook

With the successful NetSat launch the final goal of this thesis was achieved – an in-orbit demonstration of the Compass protocol and the middleware being suitable for extremely demanding mission environment:

- *limited radio link*: single 9k6 half-duplex radio link to monitor and control over 60 individually accessible satellite subsystems
- *limited hardware resources*: fully equipped Compass embedded software is capable of running on subsystems with extremely limited 16 bit microcontrollers, thus minimizing memory area for potential SEU effects

- *limited workforce*: nominal operations, such as TT&C, record auto-operations, monitor ground station, etc., can be performed by one operator

Nonetheless, the NetSat mission is only in its beginning and many fascinating developments and achievements will be presented in future. Fortunately the technical outcome of this thesis has provided a basis that is now used by other scientists at ZfT to realize their research. So *Julian Scharnagl* and *Panayiotis Kremmydas* are elaborating formation control algorithms that will be executed on NetSat. *Anna Aumann* is working on the improvements of the attitude determination and control system. Together with *Eric Jäger* she is developing formation control software on the AOCS subsystem. *Roland Haber* will execute ISL experiments and elaborate a more in-depth analysis compared to the one performed in this thesis.

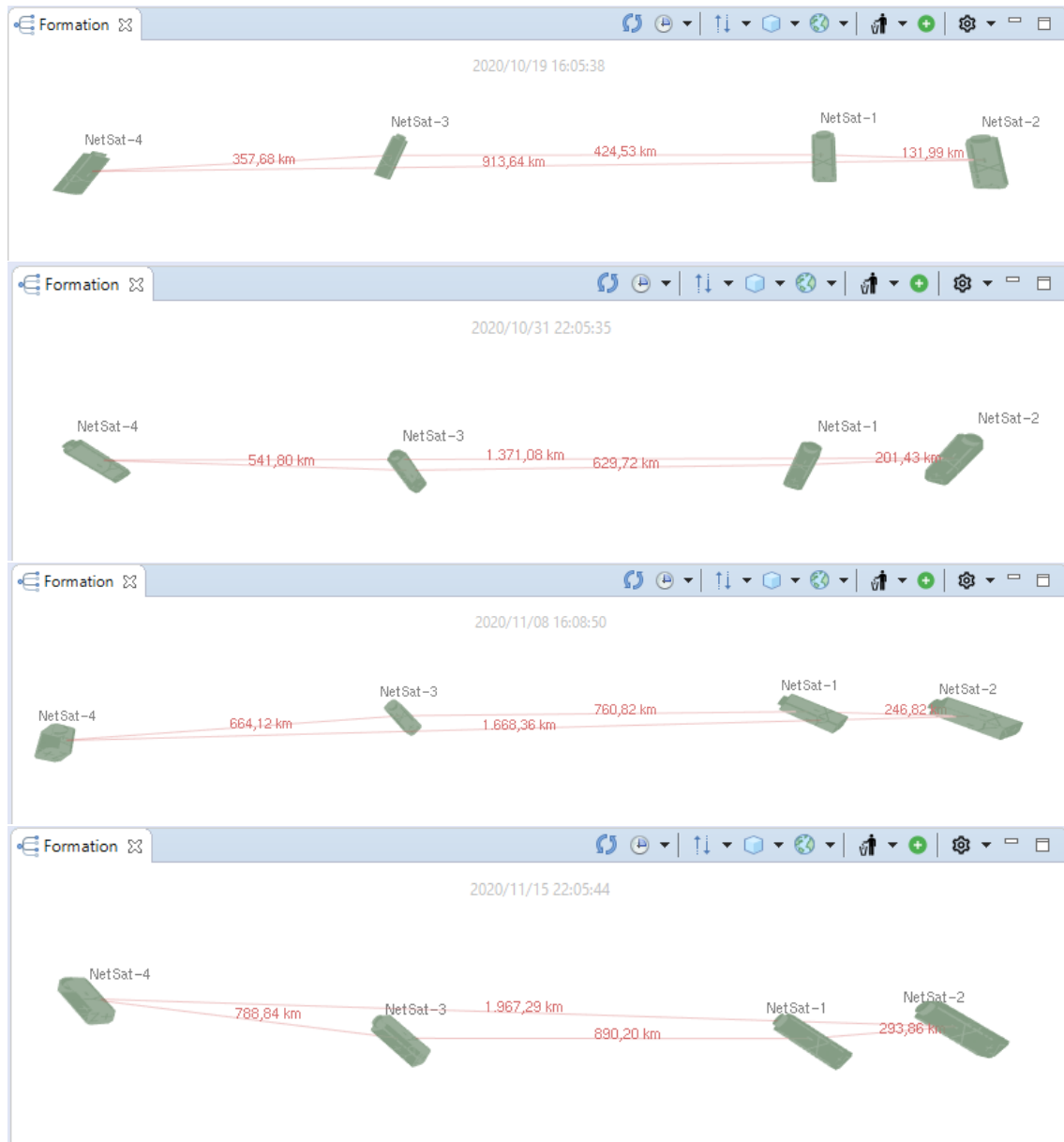


Figure 8.8: NetSat formation at four different time points: 19 October 2020, 31 October 2020, 8 November 2020 and 15 November 2020. Visualized in the *Formation View* of the Compass front-end

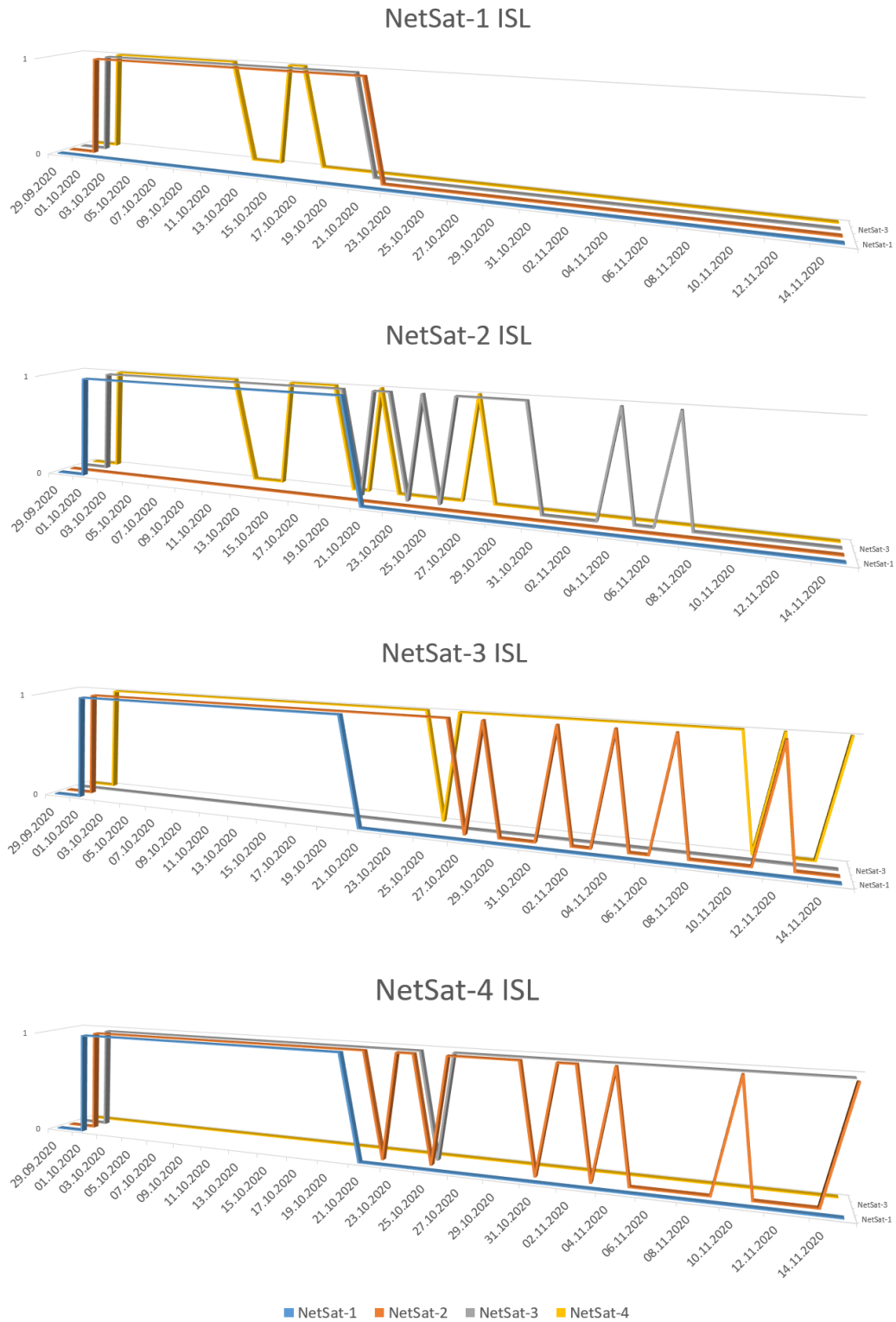


Figure 8.9: Detected ISL activity from the point of view of the NetSat-1, NetSat-2, NetSat-3 and NetSat-4 respectively

## 9 | Conclusions

The proposed approaches and the corresponding implementations of this thesis have been tested in depth in multiple CubeSat missions on board the: 1U UWE-3, 1U UWE-4 and four 3U NetSat satellites. The complexity of all ZfT's current and future satellite missions has been dramatically reduced with respect to the interfacing, software implementation, realization of cooperative processes, testing and operations. With standard interfaces all team members can focus on their specific tasks, thus considerably reducing overhead. All proposed and tested technologies are suitable for the fulfilment of common nanosatellite formation objectives, such as formation control, in-the-loop testing and forward-looking operation approaches. Due to the permanent switching between theory and practice, all mentioned methods and implementations are now part of the live-system at ZfT and are actively used for all satellite-related activities.

### 9.1 Ground Segment

This work has shown, that a common Compass protocol and a small set of standard services is sufficient to generalize the functionality of all space and ground systems. This approach not only greatly simplifies the interfaces between multiple systems, it also allows a single set of GUI components in a front-end to be used both to control satellites and to control ground systems. At ZfT both *ground stations* and *mission servers* are actively used to support operations of multiple in-orbit satellites and have been prepared for further satellites in the ongoing missions. Due to the applied model-based approach, these ground systems are now monitored, controlled and operated in the same manner as the satellites. For example, the Model service is used to access both ground stations to:

- view the live image of the antenna
- view current antenna orientation, transceiver parameters, etc.
- change the tracking priority of the satellites
- change satellite protocol chain for test purposes
- point antenna to an engineering model on ground for RF tests

Both ZfT's high-precision *motion control simulators* (turntables) are running on top of the Compass middleware and are actively used for flight model testing and sensor calibration, and to test cooperative image acquisition and feature detection with two 3U CubeSats. Each CubeSat can access the turntable, into which it is mounted, to change its own orientation. That is, the model-based approach allows a turntable to take over the responsibility of the satellite's attitude control system.

The Compass middleware is also used as basis for the ZfT's formation simulators. A simulation is consisting of multiple cooperating Compass nodes, with each node being responsible for a particular simulation task. All simulation nodes are accessed in the same way as other Compass-enabled systems – via standard services. That is, a simulation node can substitute a real satellite subsystem for which it was designed, and vice versa.

Regarding the software complexity – the software of all ground nodes at ZfT and University of Würzburg, Chair VII Robotics and Telematics, is based on the Compass middleware library (CompassNode), which has dramatically simplified the implementation process and the ability to add new ground systems, such as new SDR radios, future S-band ground station, etc.

A comparable middleware for most ground systems is the *OHB's RAMSES* (Rocket and Multi Satellite EMCS Software), which was utilized in the Prisma mission. However, it was designed solely for ground systems and relies on a proprietary UDP protocol – and is therefore limited to IP-activated devices. In contrast to RAMSES, the Compass protocol supports all space and ground systems and provides dynamic and decentral network capabilities *without* further network protocols on lower OSI layers.

The developed *Simple Downlink Share Convention* was the first successful attempt to standardize the forwarding of satellite packets from the radio amateur community towards the satellite providers. This interface was initially tested with UWE-3 and is now an interface of the reputable SatNOGS platform. SiDS hat considerable improved the output of the UWE-3, UWE-4 and NetSat missions and will definitely be appreciated in ongoing and future nanosatellite missions.

## 9.2 Multi-Satellite Operations

At ZfT and the University of Würzburg, Chair VII Robotics and Telematics, the software of all ground stations, mission servers and the Operations front-end is designed to *support multiple satellites*. A ground station is configured (via Model service) to track multiple satellites, whereat every satellite entry contains: NORAD-ID, protocol chain, Compass address, formation identifier and priority. It can autonomously track multiple satellites and automatically switch tracking based on the *current uplink traffic*.

The Compass Operations front-end became a standard application at ZfT for monitoring, control and operations of all mission nodes and provides auxiliary satellite-specific views to support operations of multiple satellites and satellite formations. At ZfT the Compass Operations became *the only front-end* used to operate all mission-relevant systems:

- ground stations
- mission servers
- high precision motion simulators
- flat-sat tests and operations
- orbit simulations
- all satellite subsystems, independent of their location: flat-sat, engineering model, connected via serial line or via RF communication (ground and space)

In contrast to *ESA's SCOS 2000*, the Compass front-end was designed in a more universal way and therefore supports operations of all space and ground systems. Furthermore,

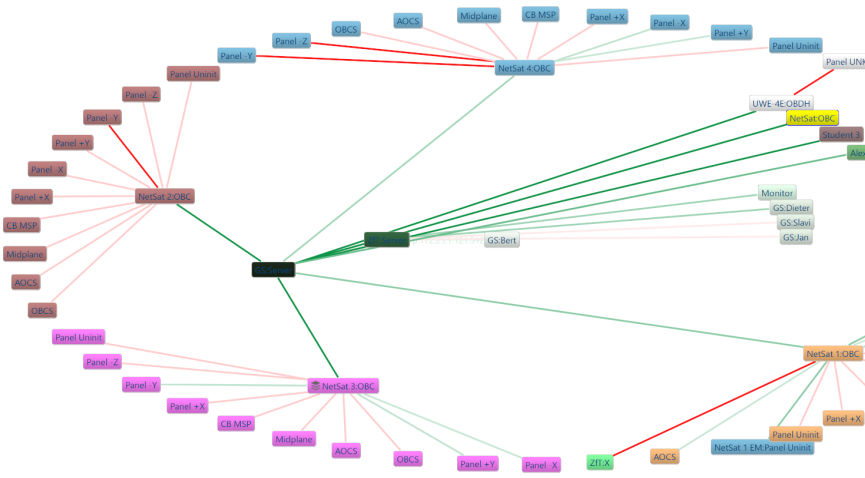


Figure 9.1: Example of the dynamic mission network, shown in the Operations front-end, during the NetSat overpass.

the front-end goes far beyond and offers advanced GUI components, which automatically detect remote model entries and visualize them accordingly (*model-based GUI*), such as:

- detected TLE model values (e.g. from ground stations) are automatically shown in the *Schedule View*
- detected latitude, longitude and altitude values are automatically shown in the *Earth View*
- detected ECI values (from satellites) are automatically shown in the *Formation View*

That is, the Operations front-end automatically configures itself depending on the current model state of the entire mission (top-level digital twin of the mission).

The Compass front-end is *highly customizable at runtime*, i.e. single GUI components (*views*) can be arranged depending on the operator's responsibilities: ground station maintenance, simulation control, pre-flight tests and verification, formation analysis, AOCSS control, TT/C, operations scheduling, experiment design, etc. It offers views for all available services: commanding, file transfer, model access, unit-testing, etc. All GUI components can either be pointed individually to one or multiple target systems, or be configured altogether with the *main target address*. Some examples of the same front-end but with different view arrangements are shown in the Appendix.

After the NetSat launch in September 2020, all four satellites were operated with multiple distributed front-end instances. The majority of the operations was executed automatically by the *operations schedule*, which is part of the Compass front-end and allows *visual recording of complex operations* for one or multiple targets. During the overpass, the recorded operations were automatically executed – including the check of the desired/correct satellite responses.

To prepare operations, numerous front-end GUI components were used to: design and control remote Tiny scripts, perform file transfer (both uplink and downlink), read and change remote model values and execute commands. Over time, more and more operation tasks were moved from ground into space, i.e. instead of running recorded operations from ground, they were uploaded as generated Tiny scripts to all four satellites. For example,

the OBC's Tiny-based scheduler was conducting the acquisition of Earth images at specific time points (see example in 9.2). The recorded operations approach has greatly improved the operations procedures and enabled the ZfT's team to control four satellites using only one half-duplex UHF link – also for image downloads. However, if more than one image download per day is desired, faster downlink channels must be considered (S-band and above).

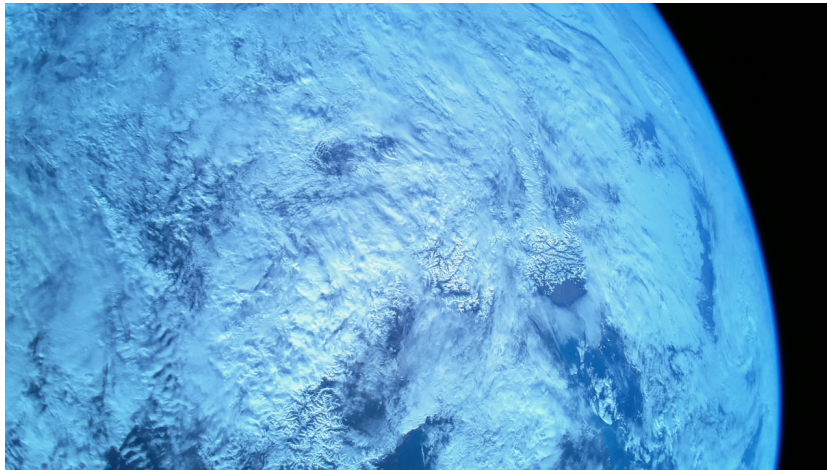


Figure 9.2: Image showing snowy Alps. Taken by NetSat-3 on 13 December 2020

### 9.3 Space Segment and In-Orbit Autonomy

The embedded Compass middleware implementation has been tested in orbit with the UWE-4 and NetSat on *almost 60 separately* accessible nodes and is now the base for all currently available in-house satellite subsystems at ZfT.

All Compass services were designed to support very slow ground-space links with highly asymmetric packet loss ratios in the up- and down directions. During the UWE-4 and NetSat operations the uplink was usually much worse than the downlink. In this case, the File service could achieve better download throughput as it would for example be possible with CCSDS CFDP service, which requires separate session start and stop packets. At the cost of slightly larger service overhead, every single service packet carries all information that is required by the receiver to successfully process the packet (*session-less* communication).

The cooperative utilization of the Compass protocol, the Tiny interpreter and the storage device abstraction layer is the first available all-in-one solution for distributed computing for very low-power 16 bit microcontrollers. Currently there exist no other script interpreter with comparable capabilities and such low hardware requirements (16 bit architecture, 1 kB RAM, 9 kB ROM). Virtually any C-programmable microcontroller can now be used to offer its functionality to other nodes or be connected to an existing Compass network to gain access to already existing functionality.



The Tiny script service has enabled far-reaching autonomy in the UWE-3, UWE-4 and NetSat satellites. Due to its very low memory requirements, the service is offered by all satellite subsystems – in contrast to comparable missions, where the autonomy is either achieved by using much more powerful microcontrollers or by utilizing a separate high-power subsystem that is designed exclusively for in-orbit scheduling and executing high-level script languages (Java, Python, etc.). Thus, the script service can be used to distribute dynamic tasks to all subsystems, instead of focusing these tasks to one single subsystem.

In the *UWE-3 1U CubeSat mission* the interpreter was utilized to execute several attitude determination and control algorithms that were developed on-demand. The experiment designer, Philip Bangert, was able to react to the observed behaviour of the satellite in space and gradually improve ADCS algorithms, and eventually fulfil the mission goals[BBS14a].

In the follow-up *UWE-4 1U CubeSat mission*, the Compass middleware was for the first time installed on all satellite subsystems. All subsystems became accessible as distinct nodes in the mission network and offered all standard Compass services. As a consequence of this, Tiny scripts could be executed on several subsystems simultaneously, thus enabling distributed and dynamic code execution. Alexander Kramer has utilized Tiny service to perform numerous in-orbit experiments to characterize the NanoFEEP propulsion system and eventually demonstrate the first electric propulsion on a 1U CubeSat[KBS20].

After the launch of the first ZfT's formation mission, *NetSat* – consisting of four 3U CubeSats, in September 2020, further 52 individually accessible Compass nodes were added into space. The Tiny service was utilized to perform numerous experiments in different areas: ISL characterization, Earth image acquisition, characterization of new reaction wheels and attitude detection, etc. Many tasks that were previously executed from ground by the (auto) operators, have been shifted into space. So, many operation schedules were auto-converted to Tiny scripts in the Operations front-end and automatically uploaded to all four satellites. The *main* Tiny code was mostly running on the always-on OBC. From there, the main code has automatically deployed smaller code fragments to other subsystems for further subsystem-specific actions, such as image acquisition that has been performed by the high-power Computing Board.

Since the Tiny language was designed to produce extremely dense byte-code, the operations of all active satellites could be performed with a *single half-duplex 9600 baud UHF* connection.

## 9.4 Compass Protocol usage

To quantify the influence of the Compass protocol, the complete ground-space traffic of the UWE-3, UWE-4 and NetSat missions was compared. The results are shown in figure 9.3 (more detailed graphs are shown in the Appendix). The upper graph shows relative packet numbers used in the corresponding service for uplink and downlink in relation to the number of all recorded packets in the mission. The lower graph shows the relative data traffic in relation to the entire traffic of the mission (see table 7.3). For the sake of convenience, the *Variable service* of the UWE-3 was put in comparison to the *Model service*.

It is clearly visible, that the command traffic (the amount of packets and aggregated

data) was dramatically reduced after the transition towards the Compass-based communication. With model-based approach most of the functions became accessible in the system's model via Model service. In the NetSat mission, model-based operations became even more prominent – as can be seen in the relative data traffic of the Model service. The increased traffic of the *Log* and *Network services* shows an improved ability of the operators to reproduce the state of the mission at some specific time point in the past. The UWE-4 was configured to omit the transmission of network beacons and logs. The usage of the *Tiny service* was also improved in the NetSat mission with new auto-operations capabilities – compared to the UWE-4, the data traffic of the service was increased and at the same time the packet traffic was considerably reduced, i.e. less control packets were required to achieve similar results.

## 9.5 Future Work

Based on the now available distributed computing capabilities, different distributed formation control mechanisms will be evaluated. Since the Tiny interpreter is permanently active in the low-power mode, continuous formation control with more frequent update rates can now be established.

Since the established Compass-based system can be viewed as a mission-global middleware, many existing standards and technologies will be adopted in order to make them available in the ZfT's mission networks. So *CCSDS protocols* and *ECSS-standardized protocols* will be utilized either on top or below the Compass stack. This will be of particular importance for missions that require these standardized protocols. Such missions will still benefit from the existing dynamic mission network (routing, DTN) and uniform access to all existing Compass-enabled nodes.

The outcome of this work can be used as a guideline for new (formation) satellite missions. It provides support for satellite mission planning, as it proposes a way of *how* the overall mission basis can be built-up: structure of the space and ground segments, testing, access to functionality and protocols.

Since the basis of this work is not limited to satellite missions, the uniformity approaches will be expanded to other departments of Zentrum für Telematik: Robotics, Mobile Systems and Industry 4.0 Production and remote Maintenance. The theory behind Compass and MTBA as well as the elaborated solutions (Compass OS, CompassNode and Compass Operations front-end) are well suitable to approach many challenges that are structurally comparable to those existing in the space industry.

One of the future research aims is to enable *Goal-based operations* (GBO) to control the mission operations of a nano-satellite formation [Nog+17]. Here a goal defines what to do and not how to achieve it [DAS08]. The goal-based approach has already been tested on “big” satellites, such as the *AEGIS software* on MER and MSL rovers or *ASE* on EO-1 [Est+12]. In a GBO-activated mission the operator defines goals for the entire space segment. The breakdown of the goals into tasks and its distribution is handled by the *Planner*, which requires comparable high computation resources due to the nature of the planning optimization problem. The resulting plans and sub-plans are distributed as Tiny scripts along the formation nodes via Compass and executed by the on-board executives. With the Linux-based *Computing Board* on all NetSat satellites, GBO could be generated



Figure 9.3: Comparison of the relative packet and data traffic of different services used in UWE-3, UWE-4 and NetSat missions. The analysis was performed using the entire recorded space-ground traffic – in total over 1.5 million packets – of the corresponding missions.

in space.

The established Simple Downlink Share Convention can be extended with uplink functionality – without the need to hand over hardware control to other network members – instead a new Compass-enabled standard can be established, which enables satellite uplink by offering Internet API to the satellites in range, thus additionally converting all uplink-able external stations to Internet access points.

## 9.6 Publications

The advances of this thesis were presented in the following publications (in descending chronological order):

- Horst, T. and Kleinschrodt, A. and Freimann, A. and Jaeger, E. and Dombrovski, S. and Haber, R. “Extended Ground Station Concept and its Impact on the In-Orbit Communication with the Four-Nano-Satellite Formation NetSat”. In IEEE Radio and Wireless Week 2021, 2020[Hor+20].
- S. Dombrovski and K. Schilling. “Control of Multi-Picosatellite Systems: Tiny Scripting Language and Multi-Layer Compass Protocol”. In SpaceOps, Marseille, France, 2018 [DomSpaceOps’18].
- S. Dombrovski, O. Ruf and K. Schilling. “Uniform, Multi-Level protocol for Ground and Space Segment Operations and Testing”. In 4S Symposium, Sorrento, Italy, 2018 [DRS18].
- S. Dombrovski and K. Schilling. “In-Orbit Database and Distributed Computing based on Tiny 2 Language”. In 68th International Astronautical Congress, Adelaide, Australia, 2017 [DS17].
- T. Nogueira, S. Dombrovski, S. Busch, A. Gasparyan and K. Schilling. “Monitoring and Control of the NetSat Formation: Concepts and Tools for Operations of Multi-satellite Systems”. In 68th International Astronautical Congress, Adelaide, Australia, 2017 [Nog+17].
- O. Ruf, S. Busch, S. Dombrovski and K. Schilling, “Challenges and Novel Approaches for Testing Large Number of Small Satellites”. In 68th International Astronautical Congress, Adelaide, Australia, 2017 [Ruf+17].
- T. Nogueira, S. Dombrovski, S. Busch, K. Schilling, K. Zakšek and M. Hort. “Photogrammetric Ash Cloud Observations by Small Satellite Formations”. In IEEE Metrology, Florence, Italy, 2016 [Nog+16].
- P. Bangert, S. Dombrovski, A. Kramer and K. Schilling. “UWE-4: Advances in the Attitude and Orbit Control of a Pico-Satellite”. In Small Satellites, System and Services Symposium (4S), Valetta, Malta, 2016 [Ban+16].
- P. Bangert, S. Busch, S. Dombrovski, A. Kramer and K. Schilling. “UWE – Lessons Learned and Future Perspectives”. In 3rd IAA Conference On University Satellite Missions and Cubesat Workshop, 2015 [Ban+15].
- S. Dombrovski and P. Bangert. “Introduction of a new sandbox interpreter approach for advanced satellite operations and safe on-board code execution”. In 66th International Astronautical Congress, Jerusalem, Israel, 2015 [DB15].
- S. Dombrovski. “Introduction of a new Framework for Intuitive and Rapid Software Evolution”. In European Ground System Architecture Workshop, Darmstadt, Germany, 2015 [Dom15a].

- S. Busch, P. Bangert, S. Dombrowski and K. Schilling. “UWE-3, In-Orbit Performance and Lessons Learned of a Modular and Flexible Satellite Bus for Future Picosatellite Formations”. In *Acta Astronautica*, Volume 117, Pages 73-89, 2015 [Bus+15].
- S. Dombrowski. “Simple Downlink Share Convention v0.9”. University of Würzburg, 2015 [Dom15b]. *Publicly accessible standard*.
- S. Dombrowski and K. Schilling. “Approaches for Efficient Global Ground Station Networks for Multiple Small Satellites”. In *Second UNISEC-Global Meeting*, Tokyo, Japan, 2014 [DS14].
- S. Busch, P. Bangert, S. Dombrowski and K. Schilling. “UWE-3, In-Orbit Performance and Lessons Learned of a Modular and Flexible Satellite Bus for Future Picosatellite Formations”. In *65th International Astronautical Congress*, Toronto, Canada, 2014 [Bus+14].

Also advances that were elaborated during the undergraduate and postgraduate studies were used as inputs and knowledge base at the beginning of this thesis:

- S. Dombrowski. “UWE-3 Communication and Operation Capabilities”. MSc. Thesis in Space Master and Technology at University of Würzburg, 2012 [Dom12].
- S. Dombrowski. “Automatische Kalibrierung einer Bodenstationsantenne für Satellitenkommunikation”. BSc. Thesis in IT Science at University of Würzburg, 2010 [Dom10].



# Appendices





# Compass Node Creation

## Matlab

Listing 1: Compass Node Creation in Matlab

```
1 %% Compass
2 % CompassMatlabNode node = new CompassMatlabNode();
3 % node.setup('GS:Philip', '132.187.9.173', 'COM1', 9600);
4 % % node.setup('GS:Philip', [], [], 0); // use no server and no serial
5 % node.register('matlabCallbackGDS', 14); // 14 =
   GDS api
6 % node.sendPacket(new CompassPacket(...));
7 % node.sendPacket('UWE4:OBDH', 2, String); // send
   String
8 % node.sendPacket('UWE4:OBDH', 2, String, 'matlabCallback', 2000); // send
   String (2000 = timeout ms)
9 % node.sendPacket('UWE4:OBDH', 14, BYTE_ARRAY); // send
   byte array to GDS
10 % node.sendPacket('UWE4:OBDH', 14, BYTE_ARRAY, 'matlabCallback', 2000); // send
   byte array to GDS (2000 = timeout ms)
11 % node.unregister('matlabCallbackGDS', 14); //
   unregister single
12 % node.unregister(); //
   unregister all
13 % node.start();
14 % node.stop();
15
16 % ==== Available APIs ====
17 % API_NETWORK      0
18 % API_ECHO         1
19 % API_COMMAND      2
20 % API_UPLINK       3
21 % API_DOWNLINK     4
22 % API_LOG          5
23 % API_DEBUG        6
24 % API_VARIABLES    7
25 % API_EUNIT        8
26 % API_GMS          9
27 % API_GLS         10
28 % API_NFS         11
29 % API_FIRE        12
30 % API_TINY        13
31 % API_GDS         14
32 % API_TURNTABLE   20
33 % API_CHAT        99
34 % =====
35
36
37 %% Add Matlab jar to dynamic path
38 javaaddpath(char(java.io.File('CompassNode.jar').getAbsolutePath()));
39 eval('import de.uwe.compass.matlab.CompassMatlabNode');
40
```

---

```

41 %% Create one or multiple compass node(s)
42 node = CompassMatlabNode('Slav:i', '132.187.9.173', [], 0);
43 %      ('GS:Student 1', [], [], 0); % No
44 %      Server, no COM      ('GS:Student 1', '132.187.9.173', 'COM1', 9600); % Uni (
45 %      ZfT: '172.25.1.121') ('1:5', [], [], 0); % Use
46 %      numerical address   ('Phil:M', [], [], 0); % Use
47 %      4/1 ASCII address (recommended)
48 %node.setup('GS:Student 1', '172.25.1.121', [], 0); %
49 %      change settings of an existing node:
50 %% Register callbacks and Start Compass
51 node.register('CompassCallback', 99); % chat service
52 node.register('GDSCallback', 14); % Beta
53 node.register('NFSCallback', 11); % NFS Callback example
54 node.start();
55 %% Send messages (Address can be numerical string, e.g. '1:5')
56 node.sendPacket('GS:Slavi', 99, 'This is a chat message'); % string
57 node.sendPacket('GS:Slavi', 99, int8(zeros(10,1))); % byte array
58 node.sendPacket('GS:Slavi', 1, 'echo', 'CompassCallback', 2000);
59
60 %% Stop Compass
61 %node.stop();
62 % node.unregister('CompassCallback'); % unregister one callback - not necessary
63 % node.unregister(); % unregister all callbacks - not necessary

```

---

Listing 2: Compass Callback Creation in Matlab

---

```

1 function CompassCallback(error, node, packet)
2     c = node.getCompass(); % used to convert numbers (API, Address) to strings
3
4     from = char(packet.getFrom().toString(c));
5     api = char(c.getAPI(packet.getAPI).getNodeName());
6     value = char(packet.getPayload());
7     fprintf(['[%s] %s: %s\n', api, from, value]);
8
9     %disp(packet.getPayload());
10 end

```

---

## Java

Listing 3: Compass Node Creation in Java

---

```

1 package de.uwe.compass;
2
3 import de.dombrowski.fire.module.OnReceiveException;
4 import de.uwe.compass.packet.Address;
5 import de.uwe.compass.packet.CompassPacket;
6
7 public class Example {
8     public static void main(String[] args) throws Exception {
9         example();
10    }
11
12    public static final void example() throws Exception {
13        // Create a Node with local address "GS:Student 3", ZfT Server as Compass
14        // Gateway and no Serial connection

```

```
14     final CompassClient client = new CompassClient("GS:Student 3", "
15         172.25.1.121", null, 0);
16     client.start();
17
18     // Do stuff
19     final Compass c = client.getCompass();
20     final Address to = new Address("GS:Slavi", c);
21     final CompassPacket cp = new CompassPacket(c.getLocalAddress(), to);
22     cp.setAPI(Registry.API_TURNTABLE);
23     cp.setPayload(new byte[] {1, 2, 3, 4});
24     // Alternativ (gibt es noch nicht)
25     // cp.setPayload(TurntableAPI.createSetCommand(x, y, z));
26
27     // Send test packet to Slavi, when he is available
28     // Wait until some specific system is available
29     c.addCompassListener(new CompassAdapter() {
30         @Override
31         public void onRouteAction(RouteAction a, Address target) {
32             if(a == RouteAction.ESTABLISHED && target.equals(to)) {
33                 try {
34                     client.sendPacket(cp);
35                 } catch (OnReceiveException e) {
36
37                 }
38             }
39         }, true);
40
41     // Stop client
42     //client.stop();
43 }
44 }
```



# Front-End Examples

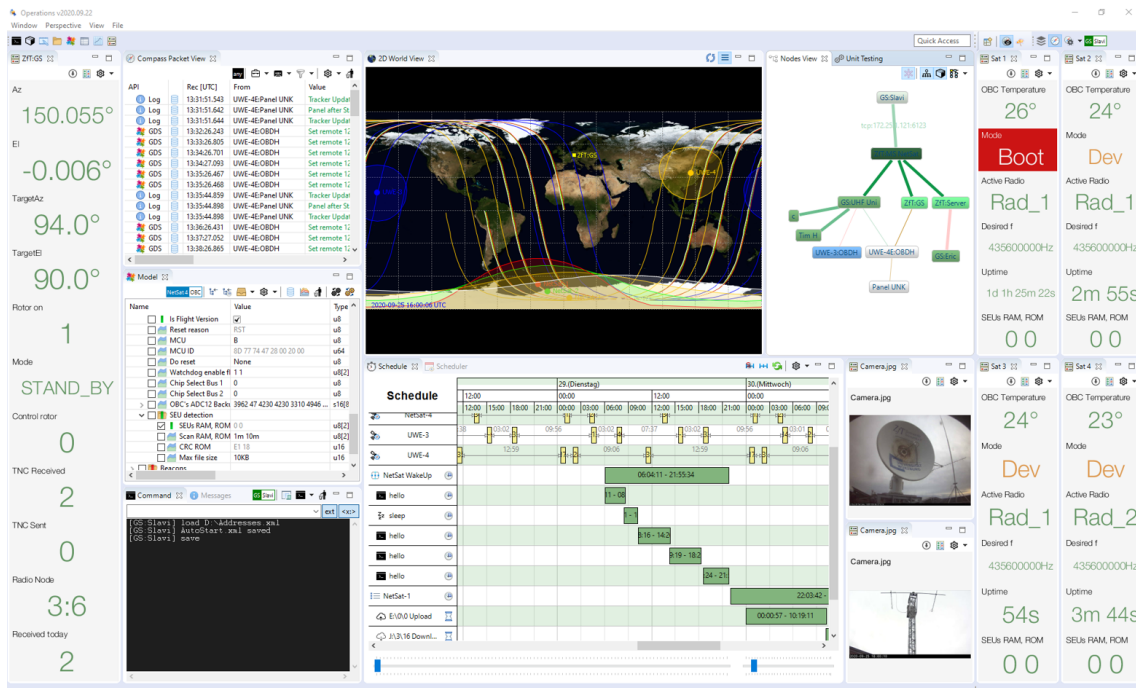


Figure 4: Compass Operations set-up for Ground Support

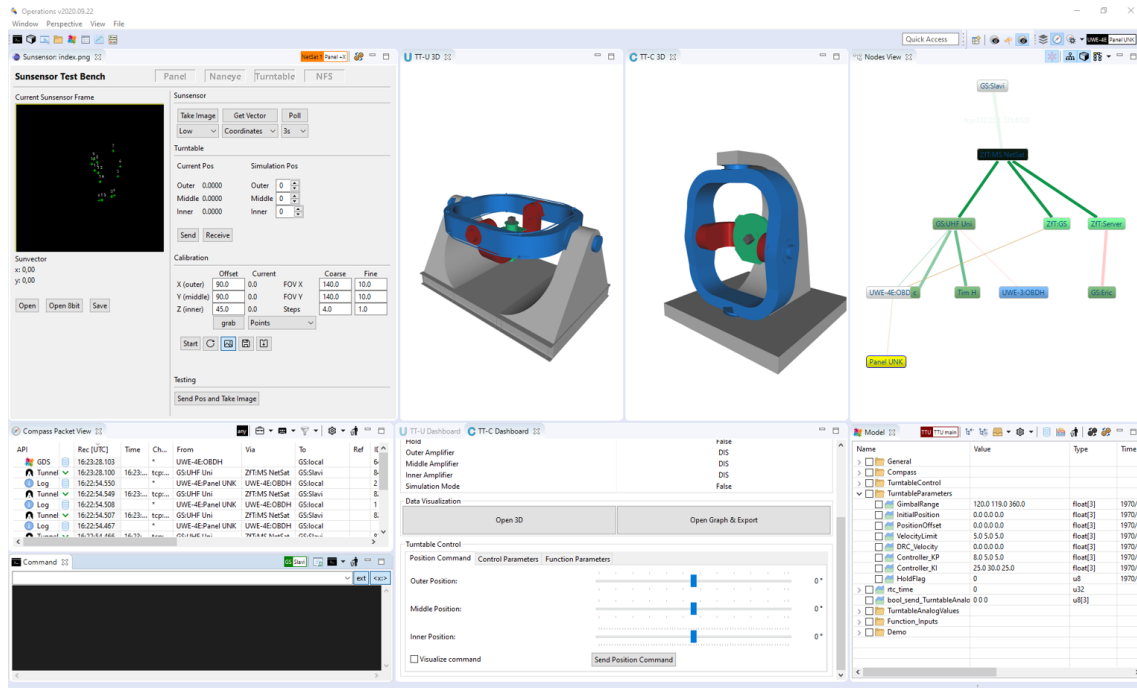


Figure 5: Compass Operations set-up of a Test Engineer

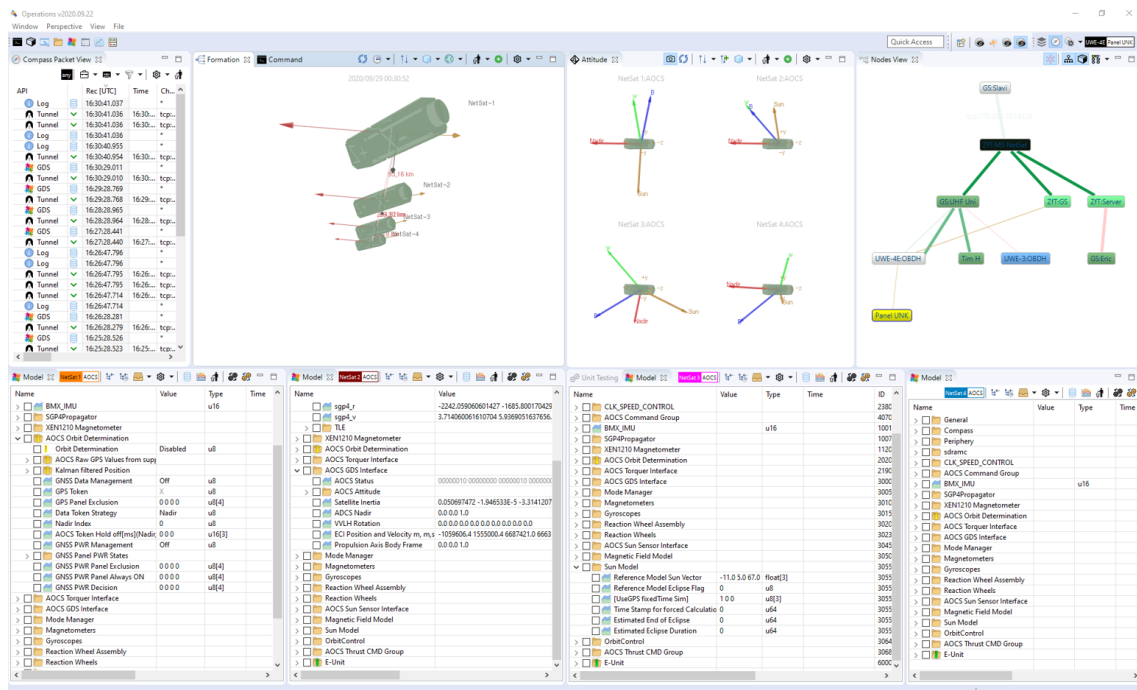


Figure 6: Compass Operations set-up of an Attitude/Orbit Control Engineer

# Traffic comparison

Following figures show a summary of the entire traffic of the UWE-3, UWE-4 and NetSat missions, which has been recorded from September 2013 until 14 November 2020.

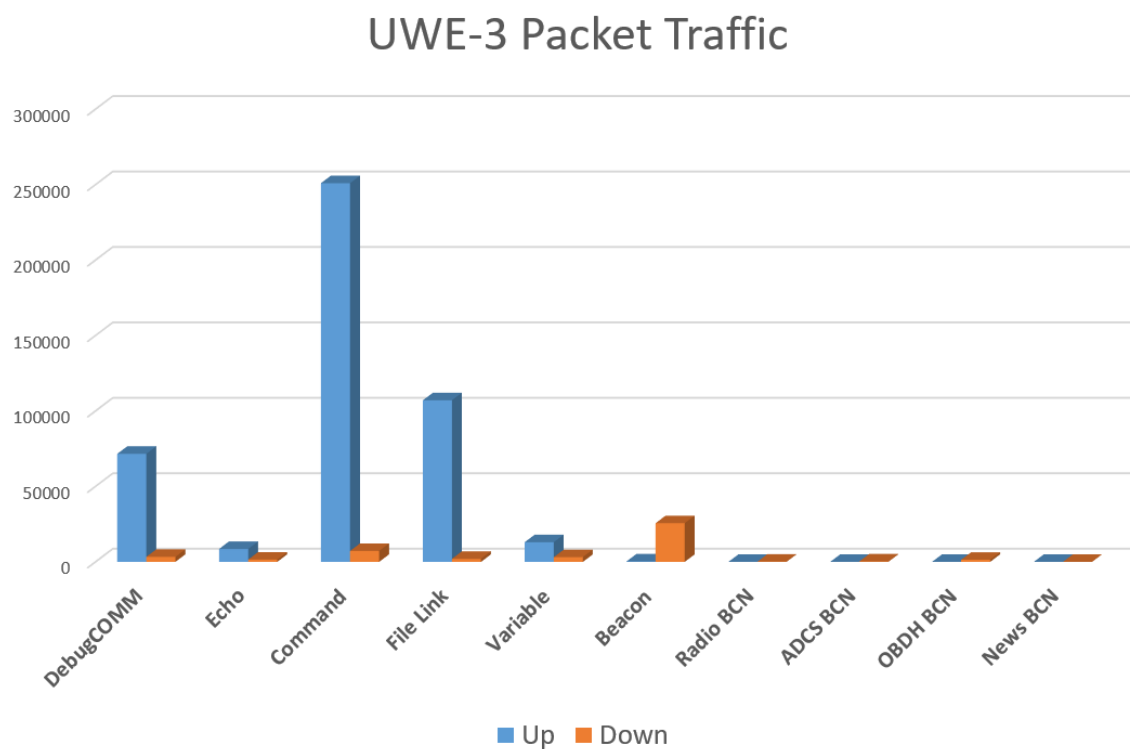


Figure 7: Recorded packet traffic of the UWE-3 mission

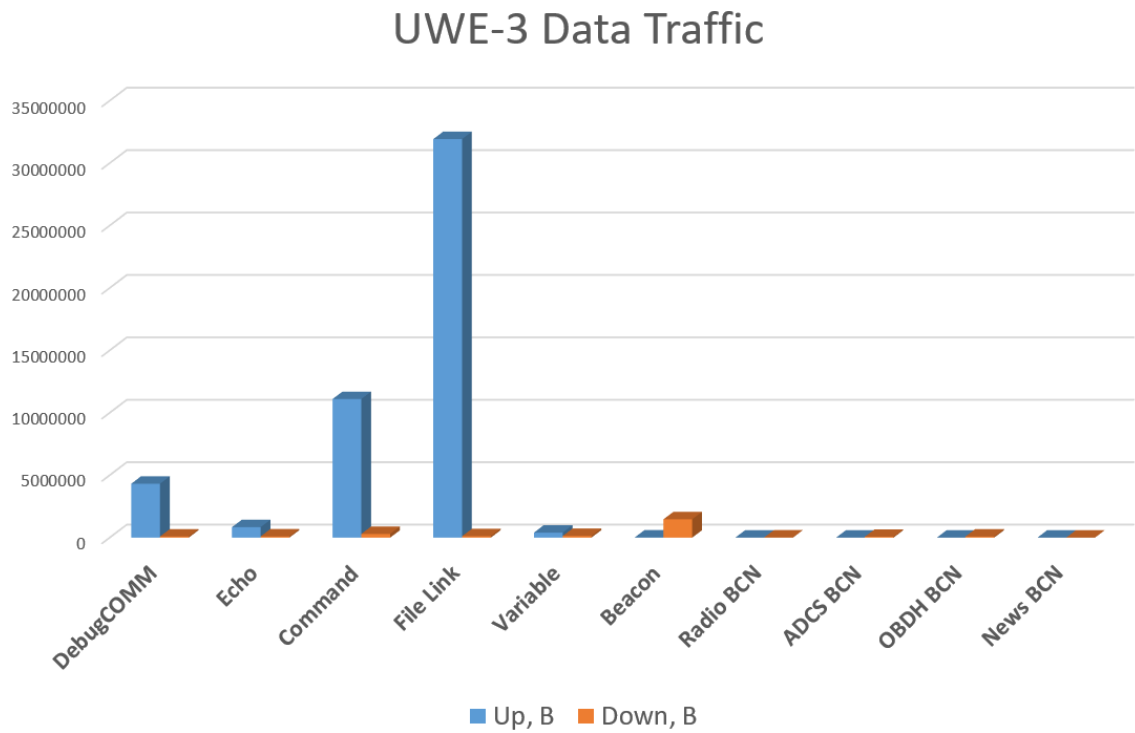


Figure 8: Recorded data traffic of the UWE-3 mission in bytes

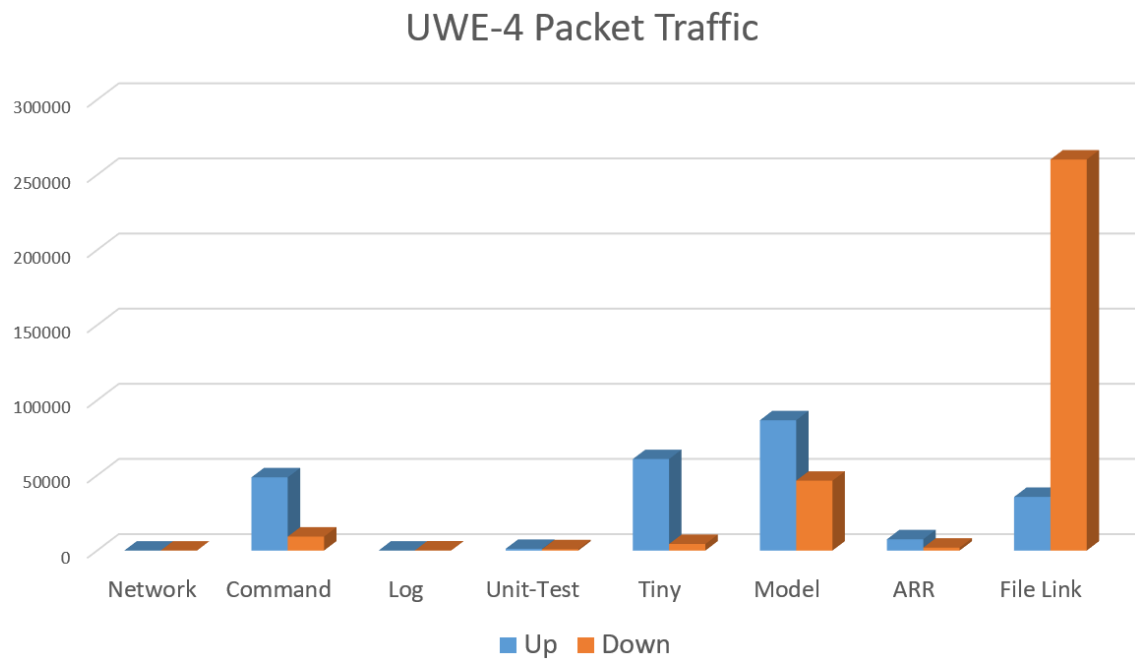


Figure 9: Recorded packet traffic of the UWE-4 mission



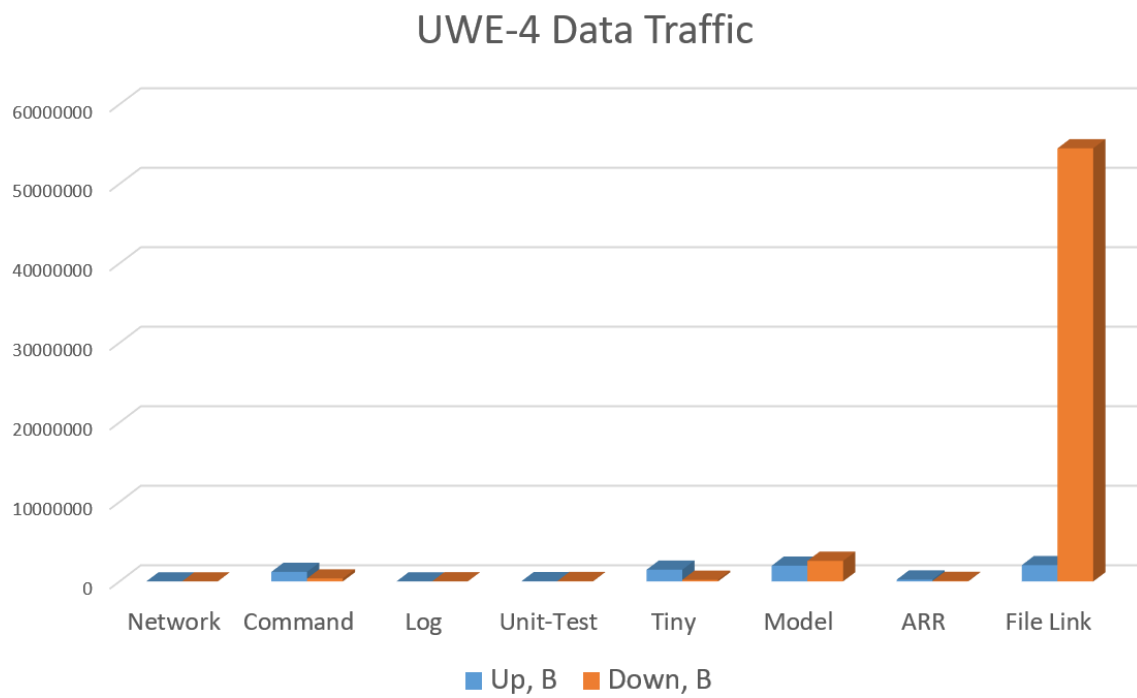


Figure 10: Recorded data traffic of the UWE-4 mission in bytes

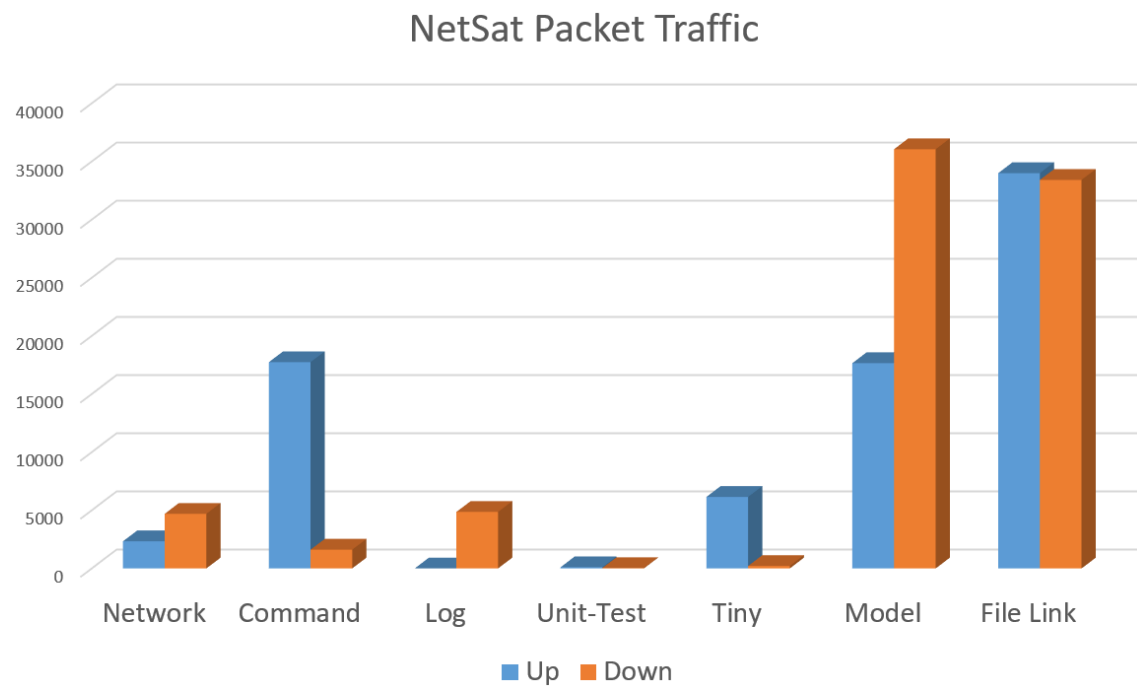


Figure 11: Recorded packet traffic of the NetSat mission

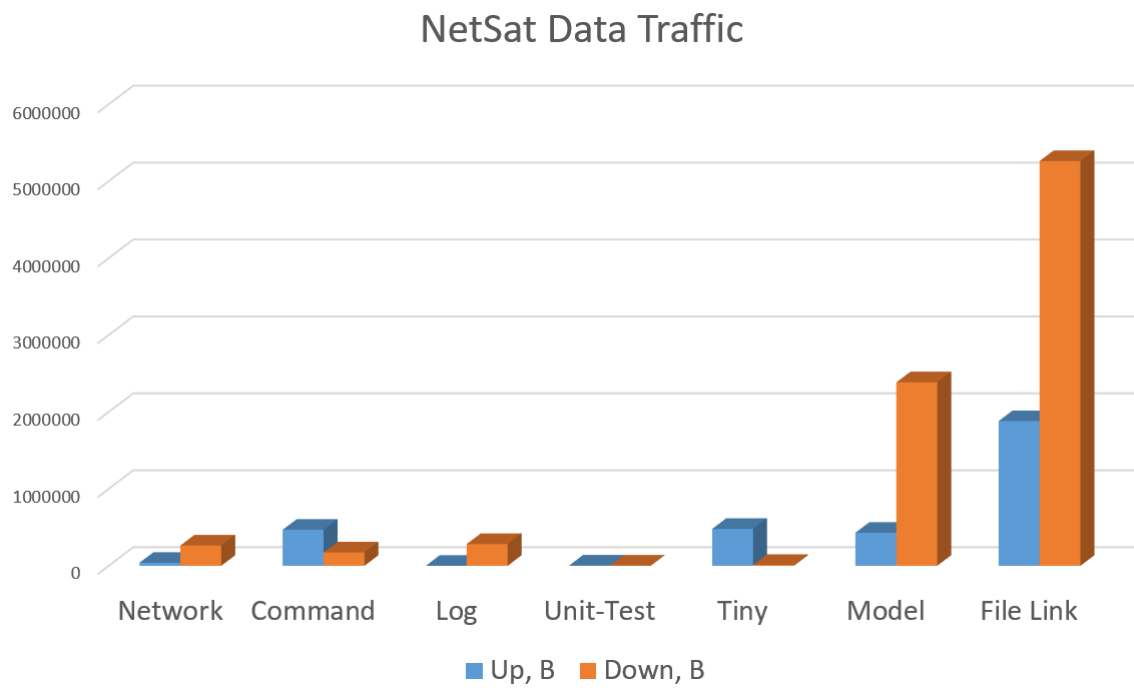


Figure 12: Recorded data traffic of the NetSat mission in bytes

# Examples from NetSat Operations

```

// #### Bug fixes ###
// How to use
// - create as Loop-Script with RepeatTime = 60s
// Used global registers
// - Z: is set to 1 when SIA has been performed
// This script does following
// - once in 60s: switch off PowerControl and activate both paths
// - twice per day perform SIA (21:00 and 09:00)

// 2020-10-08 21:00
#define UTC_NIGHT_S      1602097200
#define GDS_LOG_LOCATION 601
#define GDS_DTN          610
#define GDS_GDSSTORE     131
#define GDS_POWER_CONTROL 1302
#define GDS_POWER_PATH   1303

var(&cmdSIA,      U8, 'SIA');
var(&cmdOnAOCS,  U8, 'on AOCS');

// Persistently change Log location
if(!(gds_get(GDS_LOG_LOCATION, 0), 1)) {
    gds_set(GDS_LOG_LOCATION, 1, 1, 1); // Change Log file to B:\1\1
    gds_set(GDS_GDSSTORE, 2, GDS_LOG_LOCATION); // Persist change
}

// Switch off DTN
if(gds_get(GDS_DTN, 0)) {
    gds_set(GDS_DTN, 0); // Deactivate DTN
}

// Deactivate Power control
if(gds_get(GDS_POWER_CONTROL, 0)) {
    gds_set(GDS_POWER_CONTROL, 0); // Deactivate power control
    gds_set(GDS_POWER_PATH, 3); // Select both paths (0b11 = 3)
}

// Switch on AOCS if required (Z is 1 if SIA was performed during last run)
if(Z) {
    command(&cmdOnAOCS); // SIA
}

// Perform Subsystem ID assignment once in 12h
=(Z, <({-(getTimeS(), UTC_NIGHT_S), 43200), 50}); // 43200s = 12h
if(Z) {
    command(&cmdSIA); // SIA
}

```

Figure 13: Tiny script used to fix minor bugs on all four NetSat satellites

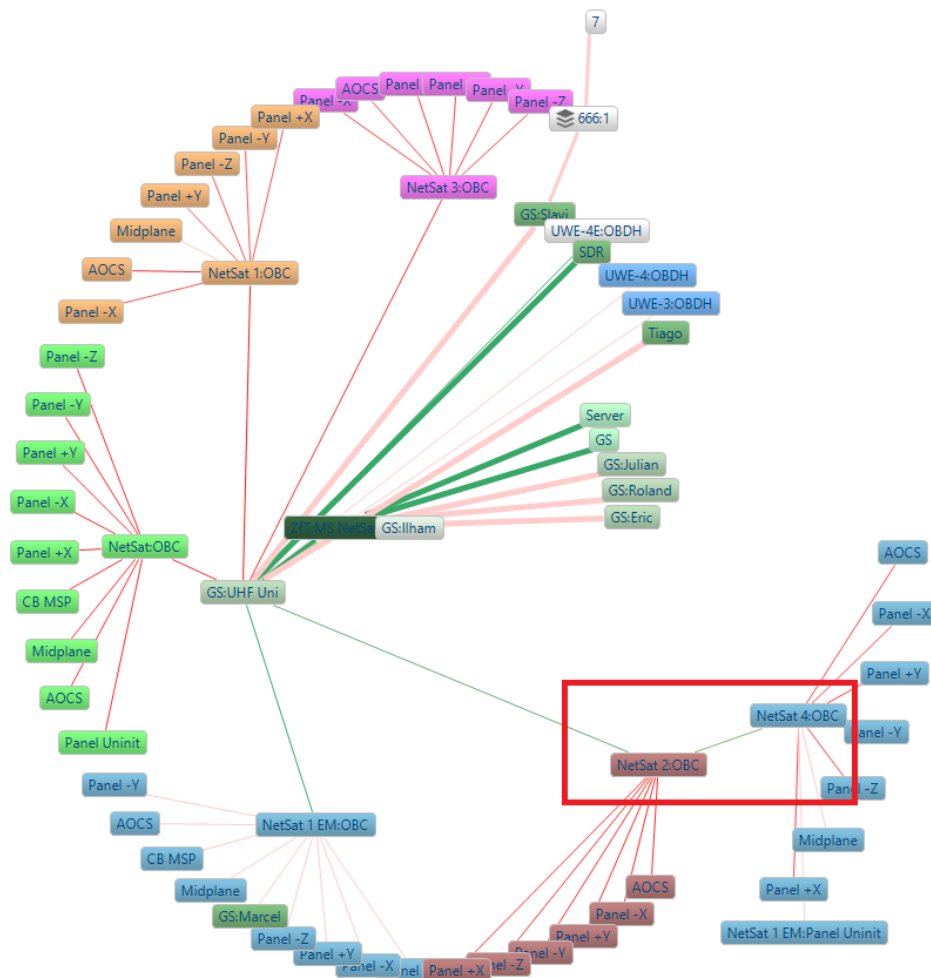


Figure 14: NetSat-4 accessible via NetSat-2 during the active tracking of the NetSat-2 satellite



# Bibliography

- [36017] HawkEye 360. “HawkEye 360 Pathfinder Cluster – Technical Information”. In: Herndon, USA, 2017.
- [AA02] National Aeronautics and Space Administration. “GRACE Launch - Press Kit”. In: 2002.
- [AA14] National Aeronautics and Space Administration. “Overview of Space Communication Protocols”. In: *Green Book*. Washington, DC, USA: CCSDS Secretariat, 2014.
- [AA15a] National Aeronautics and Space Administration. “CCSDS Bundle Protocol Specification”. In: *Blue Book*. Washington, DC, USA: CCSDS Secretariat, 2015.
- [AA15b] National Aeronautics and Space Administration. “Licklider Transmission Protocol (LTP) for CCSDS”. In: *Blue Book*. Washington, DC, USA: CCSDS Secretariat, 2015.
- [AA20] National Aeronautics and Space Administration. “CCSDS FILE DELIVERY PROTOCOL (CFDP)”. In: *Blue Book, Issue 4*. Washington, DC, USA: CCSDS Secretariat, 2020.
- [ABH18] A. Alhilal, T. Braud, and P. Hui. “The Sky is NOT the Limit Anymore: Future Architecture of the Interplanetary Internet”. In: *1810.01093v1*. Hong Kong University of Science and Technology - Hong Kong, University of Helsinki - Finland: arXiv, 2018.
- [Alm14] Lars K. Alminde. “OPSSAT PHASE A-B1”. In: European Space Agency, 2014.
- [Ast21] Astrocast. *Astronode S*. <https://www.astrocast.com/products/astronode-tm-s/>. [Online; accessed January-2021]. 2021.
- [Azz16] T Azzarelli. “OneWeb Access for Everyone”. In: Geneva, Switzerland: Global Conference on Space and Information Society (GLIS 2016), 2016.
- [Bak+16] D.N. Baker, L. Riesberg, C.K. Pankratz, R.S. Panneton, et al. “Magnetospheric Multiscale Instrument Suite Operations and Data System”. In: Springer (open access), 2016.
- [Ban20] P. Bangert. “Magnetic attitude control of miniature satellites and its extension towards orbit control using an electric propulsion system”. PhD thesis. Würzburg, Germany: University of Würzburg, 2020.

- [Ban+15] P. Bangert, S. Busch, S. Dombrovski, A. Kramer, et al. “UWE – Lessons Learned and Future Perspectives”. In: Rome, Italy: 3rd IAA Conference On University Satellite Missions and Cubesat Workshop, 2015.
- [BBS14a] P. Bangert, S. Busch, and K. Schilling. “Performance Characteristics of the UWE-3 Miniature Attitude Determination and Control System”. In: Rome, Italy: 2nd IAA Conference on Dynamics and Control of Space Systems (DYCOSS), 2014.
- [Ban+16] P. Bangert, S. Dombrovski, A. Kramer, and K. Schilling. “UWE-4: Advances in the Attitude and Orbit Control of a Pico-Satellite”. In: Valetta, Malta: Small Satellites, System and Services Symposium (4S), 2016.
- [BGM14] M.F. Barschke, K. Großekathöfer, and S. Montenegro. “Implementation of a Nanosatellite On-Board Software Based on Building-Blocks”. In: Potro Petro, Spain: 4S Symposium, 2014.
- [BS12] M. Battelino and C. Svard. “RAMSES - A modern and flexible checkout and operational ground system for small satellite projects”. In: Solna, Sweden: American Institute of Aeronautics and Astronautics, 2012.
- [Bau12] M. Baunach. “Advances in Distributed Real-Time Sensor/Actuator Systems Operation”. PhD thesis. Würzburg, Germany: University of Würzburg, 2012.
- [Ben10] S. Bennett. “Effective Scripting in Embedded Devices”. In: Mansfield, Australia: WorkWare Systems, 2010.
- [BP19] R. Birkeland and D. Palma. “An assessment of IoT via satellite: Technologies, Services and Possibilities”. In: Washington D.C., USA: 70 th International Astronautical Congress, 2019.
- [BNB12] Per Bodin, Matti Nylund, and Milan Battelino. “SATSIM—A real-time multi-satellite simulator for test and validation in formation flying projects”. In: *Acta Astronautica* 74 (2012), pp. 29–39. ISSN: 0094-5765. DOI: <https://doi.org/10.1016/j.actaastro.2011.11.015>. URL: <https://www.sciencedirect.com/science/article/pii/S009457651100350X>.
- [Bur+16] J.L. Burch, T.E. Moore, R.B. Torbert, and B.L. Giles. “Magnetospheric Multiscale Overview and Science Objectives”. In: Springer (open access), 2016.
- [Bus+14] S. Busch, P. Bangert, S. Dombrovski, and K. Schilling. “UWE-3, In-Orbit Performance and Lessons Learned of a Modular and Flexible Satellite Bus for Future Picosatellite Formations”. In: Toronto, Canada: 65th International Astronautical Congress, 2014.
- [Bus+15] S Busch, P. Bangert, S. Dombrovski, and K. Schilling. “UWE-3, In-Orbit Performance and Lessons Learned of a Modular and Flexible Satellite Bus for Future Picosatellite Formations”. In: *Acta Astronautica, Volume 117, Pages 73-89, 2015*. Acta Astronautica, 2015.



- [BBS14b] S. Busch, P. Bangert, and K. Schilling. “Attitude Control Demonstration for Pico-Satellite Formation Flying by UWE-3”. In: Mallorca, Spain: 4S-Symposium, 2014.
- [BS17] S. Busch and K. Schilling. “CubeSat Subsystem Interface Definition”. In: Würzburg, Germany: UNISEC Europe, 2017.
- [Cho10] M. Choi. “Design and Development of Generic Nanosatellite Bus Ground Control Software Suite”. MA thesis. Toronto, Canada: University of Toronto, 2010.
- [Coe17] C. Coelho. “A Software Framework for Nanosatellites based on CCSDS Mission Operations Services with Reference Implementation for ESA’s OPS-SAT Mission”. PhD thesis. Graz, Austria: Graz University of Technology, 2017.
- [CEK15] C. Coelho, D. Evans, and O. Koudelka. “CCSDS Mission Operations Services on OPS-SATs”. In: Berlin, Germany: 10th IAA Symposium on Small Satellites for Earth Observation, 2015.
- [CK16] K. Colton and B. Klofas. “Supporting the Flock: Building a Ground Station Network for Autonomy and Reliability”. In: Logan, USA: AIAA/USU Conference on Small Satellites, 2016.
- [D’E13] M. D’Errico. *Distributed Space Missions for Earth System Monitoring*. Space Technology Library, 2013. ISBN: 9781461445418.
- [DR18] R. Di Roberto. “GAUSS Approach To The Lean-Satellite Methodology”. In: Kitakyushu, Japan: International Workshop on Lean Satellites, 2018.
- [Die+16] F.-J. Diekmann, I. Clerigo, G. Albin, and L. Malleville. “A Challenge Trio in Space: ”Routine” Operations of the Swarm Satellite Constellation”. In: Prague, Czech Republic: Living Planet Symposium, 2016.
- [Dom10] S. Dombrovski. “Automatische Kalibrierung einer Bodenstationsantenne für Satellitenkommunikation”. MA thesis. University of Würzburg, 2010.
- [Dom12] S. Dombrovski. “UWE-3 Communication and Operation Capabilities”. MA thesis. University of Würzburg, 2012.
- [Dom15a] S. Dombrovski. “Introduction of a new Framework for Intuitive and Rapid Software Evolution”. In: Darmstadt, Germany: European Ground System Architecture Workshop, 2015.
- [Dom15b] S. Dombrovski. *Simple Downlink Share Convention v0.9*. University of Würzburg. 2015.
- [DB15] S. Dombrovski and P. Bangert. “Introduction of a new sandbox interpreter approach for advanced satellite operations and safe on-board code execution”. In: Jerusalem, Israel: 66th International Astronautical Congress, 2015.

- [DRS18] S. Dombrovski, O. Ruf, and K. Schilling. “Uniform, Multi-Level protocol for Ground and Space Segment Operations and Testing”. In: Sorrento, Italy: 4S Symposium, 2018.
- [DS14] S. Dombrovski and K. Schilling. “Approaches for Efficient Global Ground Station Networks for Multiple Small Satellites”. In: Tokyo, Japan: The Second UNISEC-Global Meeting, 2014.
- [DS17] S. Dombrovski and K. Schilling. “In-Orbit Database and Distributed Computing based on Tiny 2 Language”. In: Adelaide, Australia: 68th International Astronautical Congress, 2017.
- [DAS08] D.L.D. Dvorak, A.V. Amador, and T.W. Starbird. “Comparison of Goal-based Operations and Command Sequencing”. In: Heidelberg, Germany: SpaceOps, 2008.
- [Est+12] T.A. Estlin, B.J. Bornstein, D.M. Gaines, R.C. Anderson, et al. “AEGIS Automated Science Targeting for the MER Opportunity Rover”. In: ACM Transactions on Intelligent Systems and Technology (TIST), vol. 3, no. 3, pp. 1-19, 2012.
- [Eva+16] D. Evans, A. Lange, J.L. Feiterinha, J. Nörtemann, et al. “OPS-SAT: Preparing for the Operations of ESA’s First NanoSat”. In: Daejeon, Korea: SpaceOps Conference, 2016.
- [FHM15] C. Forster, H. Hallam, and J. Mason. “Orbit Determination and Differential Drag Control of Planet Labs CubeSat Constellations”. In: arXiv: 1509.03270v1, 2015.
- [Fus+16] S.A. Fuselier, W.S. Lewis, C. Schiff, R. Ergun, et al. “Magnetospheric Multiscale Science Mission Profile and Operations”. In: Springer (open access), 2016.
- [Gal+19] Damien Galano, Delphine Jollet, Karim Mellab, Jose Villa, et al. “Proba-3 Precise Formation Flying Mission”. In: July 2019.
- [Gas12] A. Gasparyan. *UWE File System documentation*. University of Würzburg, 2012.
- [GMV16] GMV. “OneWeb Awards GMV the Contract to Develop OneWeb’s Satellite Constellation Command and Control”. In: <http://www.gmv.com/en/Company/Communication/News/2016/12/satellitesoneweb.html>, 2016.
- [GOM11] GOMSpace. “CubeSat Space Protocol”. In: GOMSpace, 2011.
- [Hab+18] R. Haber, D. Garbe, K. Schilling, and W. Rosenfeld. “QUBE - A CubeSat for Quantum Key Distribution Experiments”. In: Utah, USA: 32nd Annual AIAA/USU Conference on Small Satellites, 2018.
- [HMF08] S. Hall, F. Moreira, and T. Franco. “Operations Planning for the Galileo Constellation”. In: Heidelberg, Germany: SpaceOps, 2008.

- [HPL09] H. Hellman, S. Persson, and B. Larsson. “PRISMA – a Formation Flying Mission on the Launch Pad”. In: Daejeon, Korea: 60th International Astronautical Congress, 2009.
- [Her17] N. Hermannsdörfer. “Mission Analysis of the Nanosatellite Sonate”. MA thesis. Würzburg, Germany: University of Würzburg, 2017.
- [Hla18] M. Hladky. “Vision Based Attitude Control”. MA thesis. Würzburg, Germany: Lulea University of Technology, 2018.
- [Hof+19] J. Hof, V. Karunanithib, S. Speretta, C. Verhoeven, et al. “Low Latency IoT/M2M Using Nano-Satellites”. In: Washington D.C., USA: 70th International Astronautical Congress, 2019.
- [Hor+20] T. Horst, A. Kleinschrodt, A. Freimann, E. Jaeger, et al. “Extended Ground Station Concept and its Impact on the In-Orbit Communication with the Four-Nano-Satellite Formation NetSat”. In: *Radio and Wireless Week 2021*. IEEE, 2020.
- [IFC10] R. Ierusalimschy, L. Henrique de Figueiredo, and W. Celes. “The Implementation of Lua 5.0”. In: Rio de Janeiro, Brazil: Pontifical Catholic University of Rio de Janeiro, 2010.
- [Inc17] Swarm Technology Inc. *Exhibit A to FCC Form 442*. <https://apps.fcc.gov/els/GetAtt.html?id=191177>. [Online; accessed January-2021]. 2017.
- [KS19a] I. Koren; K. Schilling Y. Schechner. “CloudCT – Computed Tomography of Clouds by a Small Satellite Formation”. In: *Proceedings 12th IAA symposium on Small Satellites for Earth Observation*. IFAC, 2019.
- [KS19b] I. Koren; K. Schilling Y. Schechner. “CloudCT: A formation of cooperating nano-satellites for cloud characterisation by computed tomography”. In: *Proceedings 70th International Astronautical Congress 2019, IAC-19 D1.6.54792*. 2019.
- [KS19c] I. Koren; K. Schilling Y. Schechner. “Small Satellite Formations to Characterize 3D Cloud Properties: TOM and CloudCT”. In: *Proceedings 4th COSPAR Symposium on Small satellites for sustainable Science And Development*. 2019.
- [Kap+16] S. Kapitola, S. Weiß, N. Korn, and K. Briß. “Von BEESAT-2 zu BEE-SAT-4: Weiterführende Nutzng als In-Orbit Testplattform”. In: *DocumentID: 420270*. Technische Universität Berlin, Germany: Deutscher Luft- und Raumfahrtkongress, 2016.
- [Kek06] D.D. Kekez. “Development of Flight Software and Communications Systems for the CanX-2 Nanosatellite”. MA thesis. University of Toronto, 2006.
- [Kin18] Kineis. *Press Release 2018/09/10*. [https://www.nexeya.com/wp-content/uploads/2018/09/dossier\\_de\\_presse\\_lancement\\_kineis\\_uk.pdf](https://www.nexeya.com/wp-content/uploads/2018/09/dossier_de_presse_lancement_kineis_uk.pdf). [Online; accessed January-2021]. 2018.

- [Kle+19] A. Kleinschrodt, I. Motroniuk, A. Aumann, I. Mammadov, et al. “TIM: An International Formation for Earth Observation with CubeSats”. In: *12th IAA Symposium on Small Satellites for Earth Observation*. Berlin, Germany, 2019.
- [Kli16] S. Klinkner. “Flying Laptop: Academic Small Satellite Flying Laptop”. In: University of Stuttgart, Institute of Space Systems, Stuttgart, Germany: [https://www.irs.uni-stuttgart.de/dokumente/fact\\_sheet.pdf](https://www.irs.uni-stuttgart.de/dokumente/fact_sheet.pdf), 2016.
- [KBS20] A. Kramer, P. Banger, and K. Schilling. “UWE-4: First Electric Propulsion on a 1U CubeSat—In-Orbit Experiments and Characterization”. In: *Aerospace 2020 no.7*. Aerospace, 2020.
- [Lam+15] K. Lamichhane, M. Kiran, T. Kannan, D. Sahay, et al. “Embedded RTOS implementation for Twin Nano-satellite STUDSAT-2”. In: *IEEE Metrology for Aerospace*. Benevento, Italy: IEEE, 2015.
- [Lan+15] M. Langer, N. Appel, M. Dziura, C. Fuchs, et al. “MOVE-II – der zweite Kleinsatellit der Technischen Universität München”. In: Rostock, Germany: Deutscher Luft- und Raumfahrtkongress 2015, 2015.
- [Lor+08] A. Loretucci, F. Groce, K. Davies, and L. Demonceau. “The GALILEO SCCF”. In: Heidelberg, Germany: SpaceOps, 2008.
- [Mar+18] T.W. Martins, A. Pereira, T. Hulin, O. Ruf, et al. “Space Factory 4.0 - New processes for the robotic assembly of modular satellites on an in-orbit platform based on ”Industrie 4.0” approach”. In: *IAC-18-D1.1.9.x44546*. Bremen, Germany: 69th International Astronautical Congress, 2018.
- [Mau+12] E. Maurer, S. Zimmermann, F. Mrowka, and H. Hofmann. “Dual Satellite Operations in Close Formation Flight”. In: Stockholm, Sweden: SpaceOps, 2012.
- [McD20] J. McDowell. “The Low Earth Orbit Satellite Population and Impacts of the SpaceX Starlink Constellation”. In: *The Astrophysical Journal Letters (ApJL)*, 2020.
- [Mer+10] M. Merri, S. Cooper, B. Behal, D. Feliot, et al. “What has CCSDS SM&C to do with ECSS PUS?” In: Huntsville, Alabama, USA: SpaceOps 2010, 2010.
- [Mor+17] E.F. Moreira, A. Ceballos, C. Estevez, J.C Gil, et al. “Architecting OneWeb’s Massive Satellite Constellation Ground System”. In: Los Angeles, USA: Ground Segment Architecture Workshop (GSAW), 2017.
- [Mui96] B. K. Muirhead. “Mars Pathfinder Flight System Design and Implementation”. In: *IEEE Aerospace Applications Conference*. Pasadena, California, USA: Jet Propulsion Laboratory, 1996.
- [Myr21] Myriota. *Products and Common Questions*. <https://myriota.com>. [Online; accessed January-2021]. 2021.

- [Nog+17] T. Nogueira, S. Dombrovski, S. Busch, A. Gasparyan, et al. “Monitoring and Control of the NetSat Formation: Concepts and Tools for Operations of Multi-satellite Systems”. In: Adelaide, Australia: 68th International Astronautical Congress, 2017.
- [Nog+16] T. Nogueira, S. Dombrovski, S. Busch, K. Schilling, et al. “Photogrammetric Ash Cloud Observations by Small Satellite Formations”. In: Florence, Italy: IEEE Metrology, 2016.
- [NFS17] T. Nogueira, S. Fratini, and K. Schilling. “Planning and Execution to Support Goal-based Operations for NetSat: a Study”. In: Pittsburgh, USA: 10th International Workshop on Planning and Scheduling for Space (IW PSS), 2017.
- [Orr+08] N.G. Orr, J.K. Eyer, B.P. Larouche, and R.E. Zee. “Precision Formation Flight: The CanX-4 and CanX-5 Dual Nanosatellite Mission”. In: Toronto, Canada: Space Flight Laboratory, University of Toronto Institute for Aerospace Studies, 2008.
- [Pec05] N.M. Peccia. “EGOS: ESA/ESOC ground operations software system”. In: Big Sky, MT, USA: IEEE Aerospace Conference, 2005.
- [Pen+20] Luis Penin, Yann Scoarnec, José Ibarz, Carolina Cazorla, et al. “Proba-3: ESA’s small satellites precise formation flying mission to study the Sun’s inner corona as never before”. In: Aug. 2020.
- [PDH10] S. Persson, S. D’Amico, and J. Harr. “Flight Results From Prisma Formation Flying and Rendezvous Demonstration Mission”. In: Prague, Czech Republic: 61th International Astronautical Congress, 2010.
- [PCC18] I. del Portillo, B.G. Cameron, and E.F. Crawley. “A Technical Comparison of Three Low Earth Orbit Satellite Constellation Systems to Provide Global Broadband”. In: Bremen, Germany: 69 th International Astronautical Congress, 2018.
- [Pot21] GFZ Helmholtz Centre Potsdam. *Mission Operations System and Mission Phases*. <https://www.gfz-potsdam.de/en/section/global-geomonitoring-and-gravity-field/projects/gravity-recovery-and-climate-experiment-follow-on-grace-fo-mission/mission-operations-system>. [Online; accessed January-2021]. 2021.
- [Pro14] The CubeSat Program. “CubeSat Design Specification”. In: California Polytechnic State University, 2014.
- [Ray03] T. Ray. “CCSDS FILE DELIVERY PROTOCOL (CFDP) - WHY IT’S USEFUL AND HOW IT WORKS”. In: NASA/Goddard Space Flight Center, 2003.
- [Reg+16] V. Reggestad, K. Symonds, T. Nogueira, M. Stanciu-Manolescu, et al. “ESA Constellation Coordination System – Development made easy by CCSDS Mission Operation Services”. In: Long Beach, California: American Institute of Aeronautics and Astronautics, 2016.

- [RSW12] M. Robichaud, E. Sandjaya, and C. Wagner. “Contribution of the On Board Software Management Tool to the Galileo Operations”. In: Stockholm, Sweden: SpaceOps, 2012.
- [Ruf+17] O. Ruf, S. Busch, S. Dombrovski, and K. Schilling. “Challenges and Novel Approaches for Testing Large Number of Small Satellites”. In: Adelaide, Australia: 68th International Astronautical Congress, 2017.
- [Sae+20] Nasir Saeed, Ahmed Elzanaty, Heba Almorad, Hayssam Dahrouj, et al. *CubeSat Communications: Recent Advances and Future Challenges*. 2020. arXiv: 1908.09501 [eess.SP].
- [Sal+18] S. Salas, H. Darnes, L. Gillot, F. Viaud, et al. “ANGELS SmallSat: Demonstrator for new French product line”. In: Marseille, France: SpaceOps, 2018.
- [San+13] S. Santandrea, K. Gantois, K. Strauch, Frederic Teston, et al. “PROBA2: Mission and Spacecraft Overview”. In: *solphys* 286 (Apr. 2013). DOI: 10.1007/s11207-013-0289-5.
- [Sch13] Nils Schifffhauer. “FunCube Dongle Pro+ V2.0 on Shortwave”. In: Published under [http://ratzer.at/pdf/Funcube2\\_DK8OK.pdf](http://ratzer.at/pdf/Funcube2_DK8OK.pdf), 2013.
- [SA20] K. Schilling and A. Aumann. “CloudCT: Design Challenges for a Formation of 10 Nano-Satellites”. In: (2020).
- [Sch+15] K. Schilling, P. Bangert, S. Busch, S. Dombrovski, et al. “NetSat: A Four Pico/Nano-Satellite Mission for Demonstration of Autonomous Formation Flying”. In: Jerusalem, Israel: 66th International Astronautical Congress, 2015.
- [Sch+18] K. Schilling, I. Motroniuk, A. Aumann, I. Mammadov, et al. “TOM – A Pico-Satellite Formation for 3D Earth Observation”. In: Sorrento, Italy: 4S Symposium, 2018.
- [Sch+17] K. Schilling, T. Tzschichholz, I. Motroniuk, A. Aumann, et al. “TOM: A Formation For Photogrammetric Earth Observation By Three Cube-sats”. In: Rome, Italy: International Academy of Astronautics, 2017.
- [SPS07] M. Schmidt, R.S. Priya, and K. Schilling. “The Pico-Satellite UWE-1 And IP Based Telecommunication Experiments”. In: *Proceedings Volumes, Volume 40, Issue 7, Pages 721-725*. IFAC, 2007.
- [Sch+09] M. Schmidt, K. Ravandoor, O. Kurz, S. Busch, et al. “Attitude Determination for the Pico-Satellite UWE-2”. In: *Space Technology* 28 (2009), pp. 67–74.
- [SGU12] A. Schwab, G. Giese, and D. Ulrich. “TDX-TSX - On-board autonomy and FDIR of whispering brothers”. In: Stockholm, Sweden: SpaceOps, 2012.

- [Sci16] SciSys. “SciSys PLENITER Product to Support OneWeb’s Deployment of the World’s Largest Satellite Constellation”. In: <http://www.scisys.co.uk/who-we-are/media-centre/detailed-news/article/801.html>, 2016.
- [Sec] CCSDS Secretariat. “Proximity-1 Space Link Protocol—Rationale, Architecture, and Scenarios”. In:
- [Sec03a] CCSDS Secretariat. “CCSDS Recommendation for Space Packet Protocol”. In: Washington, DC 20546, USA: Consultative Committee for Space Data Systems, 2003.
- [Sec03b] CCSDS Secretariat. “Ground systems and operations - Telemetry and telecommand packet utilization”. In: Washington, DC, USA: Consultative Committee for Space Data Systems, 2003.
- [Sec10] CCSDS Secretariat. “MISSION OPERATIONS SERVICES CONCEPT”. In: *Green Book*. Washington, DC, USA: Consultative Committee for Space Data Systems, 2010.
- [Sec13a] CCSDS Secretariat. “Proximity-1 Space Link Protocol — Data Link Layer”. In: *Recommendation for Space Data System Standards (Blue Book)*. Washington, DC, USA: Consultative Committee for Space Data Systems, 2013.
- [Sec13b] CCSDS Secretariat. “Proximity-1 Space Link Protocol - Coding and Synchronization Sublayer”. In: *Recommendation for Space Data System Standards (Blue Book)*. Washington, DC, USA: Consultative Committee for Space Data Systems, 2013.
- [Sec15a] CCSDS Secretariat. “AOS Space Data Link Protocol”. In: *Recommendation for Space Data System Standards (Blue Book)*. Washington, DC, USA: Consultative Committee for Space Data Systems, 2015.
- [Sec15b] CCSDS Secretariat. “TC Space Data Link Protocol”. In: *Recommendation for Space Data System Standards (Blue Book)*. Washington, DC, USA: Consultative Committee for Space Data Systems, 2015.
- [Sec15c] CCSDS Secretariat. “TM Space Data Link Protocol”. In: *Recommendation for Space Data System Standards (Blue Book)*. Washington, DC, USA: Consultative Committee for Space Data Systems, 2015.
- [Sec20] CCSDS Secretariat. “Space Packet Protocol”. In: *Blue Book*. Washington, DC 20546, USA: Consultative Committee for Space Data Systems, 2020.
- [SK07] G. Shirville and B. Klofas. “GENSO: A Global Ground Station Network”. In: San Luis Obispo, CA, USA: Electrical Engineering, Cal Poly State University, 2007.
- [Sol21] Solenix. *Solenix’s Elveti to Power Ground Segment of Astrocast*. <https://www.solenix.ch/blog/2016-11-28/solenix’s-elveti-power-ground-segment-astrocast>. [Online; accessed January-2021]. 2021.

- [Mes] “Special Delivery: NASA’s MESSENGER Sends Flyby Data to Earth Using CCSDS File Delivery Protocol Developed for Deep Space by International Team”. In: CCSDS press release, 2005.
- [Swa21] M. Swartwout. *CubeSat database*. <https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database>. [Online; accessed January-2021]. 2021.
- [Sys21a] 12G Flight Systems. *PUSOpen – RELIABLE ECSS PUS / CCSDS COMMUNICATION FOR YOUR MISSION*. 2021.
- [Sys21b] Tyvak Nano-Satellites Systems. *Tyvak Platforms*. <https://www.tyvak.com/platforms/>. [Online; accessed January-2021]. 2021.
- [tea21] SpaceX software team. *Ask Me Anything – reddit session*. [https://www.reddit.com/r/spacex/comments/gxb7j1/we\\_are\\_the\\_spacex\\_software\\_team\\_ask\\_us\\_anything/](https://www.reddit.com/r/spacex/comments/gxb7j1/we_are_the_spacex_software_team_ask_us_anything/). [Online; accessed March-2021]. 2021.
- [Tor21] Space Flight Laboratory University of Toronto. *SFL Satellite Platforms*. [https://www.utias-sfl.net/?page\\_id=89](https://www.utias-sfl.net/?page_id=89). [Online; accessed January-2021]. 2021.
- [Wae16] P. Waeltermann. “Hardware-in-the-Loop: The Technology for Testing Electronic Controls in Vehicle Engineering”. In: Paderborn, Germany: dSpace GmbH, 2016.
- [Whi+15] D.J. White, I. Giannelos, A. Zissimatos, E. Kosmas, et al. “SatNOGS: Satellite Networked Open Ground Station”. In: Valparaiso, IN, USA: Valparaiso University, 2015.
- [Wil12] J. Willmot. “Use of CCSDS File Delivery Protocol (CFDP) in NASA/GSFC’s Flight Software Architecture”. In: 6th ESA Workshop on Avionics, Data, Control and Software Systems - ADCSS, 2012.
- [Woo+07] L. Wood, W. Eddy, W. Ivancic, J. McKim, et al. “Saratoga: a Delay-Tolerant Networking convergence layer with efficient link utilization”. In: Salzburg, Austria: International Workshop on Satellite and Space Communications, 2007.
- [Zim+17] R. Zimmermann, D. Doan, L. Leung, J. Mason, et al. “Commissioning the World’s Largest Satellite Constellation”. In: Logan, USA: AIAA/USU Conference on Small Satellites, 2017.



# List of Figures

1.1	Launched CubeSats, grouped by mission types. Image source: [Swa21] . . .	2
1.2	Mission roadmap of the University of Würzburg (left) and Zentrum für Telematik (right) . . . . .	3
1.3	Ground and space segment of the exemplary NetSat formation mission . . .	5
1.4	Coverage of the implementation chapters: <i>Protocol Chapter</i> (green), <i>Space Chapter</i> (red) and <i>Ground Chapter</i> (blue) . . . . .	8
2.1	Simplified overview of the components involved in the NetSat mission. (A) NetSat flight models, (B) NetSat engineering models, (C) operator’s workstations, (D) GS server, (E) orbit simulator, (F) algorithms simulator, (G) hardware test facility . . . . .	11
2.2	Desired common protocol configuration . . . . .	12
2.3	Solenix Elveti Mission Control System overview. Image source: <a href="http://www.solenix.ch">www.solenix.ch</a> . . . . .	29
2.4	Overview of the MO Service Framework (image source: [Sec10], p.19) . . .	30
2.5	Overview of space protocols recommended by the CCSDS (image source: [AA14], p. 2-4) . . . . .	32
2.6	CCSDS recommendation with Space Packet Protocol as End-to-End forwarding (image source: [AA14], p. 4-3) . . . . .	33
2.7	CCSDS recommendation with IP protocol as End-to-End forwarding (image source: [AA14], p. 4-5) . . . . .	34
2.8	CCSDS recommendation with CFDP protocol as End-to-End forwarding (image source: [AA14], p. 4-5) . . . . .	34
2.9	GENSOO: Participating and Related Universities (image source UNISEC global) . . . . .	37
2.10	GENSOO: Ground Station Management Service (image source UNISEC global) . . . . .	37
2.11	GENSOO: GS Remote Operation Web Service (right) (image source UNISEC global) . . . . .	38
2.12	Available DTN protocols . . . . .	39
2.13	Convergence Protocol . . . . .	40
2.14	Multiple Access techniques (image sources: nature.com and maxiustech.com) . . . . .	41
2.15	SCOS 2000 TC History Display (image sources: esa.int) . . . . .	45
2.16	Pleniter Software Suite (image sources: pleniter.com) . . . . .	46
3.1	Protocol configuration of the UWE-3 mission (before this thesis) . . . . .	49
3.2	Simplified NetSat formation Mission Network . . . . .	50

3.3	MTBA Bridge . . . . .	52
3.4	Simplified view of the NetSat mission MTBA model . . . . .	52
3.5	Model Shadowing . . . . .	53
3.6	Model Based Development . . . . .	55
3.7	Development stages . . . . .	56
3.8	Mixed-Loop-Testing . . . . .	57
3.9	Protocols in a simplified NetSat mission network. (A) embedded intra-system protocol, (B) wireless inter satellite protocol, (C) protocol between workstations, (D) engineering link protocol, (E) orbit simulation handling protocol, (F) Matlab simulation protocol, (G) test facility control . . . . .	58
3.10	Compass Domains Approach . . . . .	60
3.11	Routing Example, connection of two subnetworks . . . . .	64
3.12	Protocol configuration of the UWE-4 and NetSat missions. . . . .	72
4.1	Coverage of the protocol chapter . . . . .	73
4.2	Overview of all services used between ground and space systems in the common UWE-3, UWE-4 and NetSat mission network . . . . .	83
4.3	Overview of all services used between ground systems in the common UWE-3, UWE-4 and NetSat mission network . . . . .	84
4.4	Routing table example . . . . .	87
4.5	Visualization of the local routing map in the Compass Operations front-end. Green and red lines denote active and inactive routes respectively. The width of the lines is denoting the corresponding maximum speed. The age of the knowledge is visualized with fading line color. . . . .	90
4.6	Command service front-end in the Compass Operations front-end – here showing a commanding session with the NetSat 1:OBC. . . . .	92
4.7	Downlink API front-end . . . . .	94
4.8	Uplink API front-end . . . . .	95
4.9	Log service: activation of log transmission and log storage using Model service	97
4.10	Unit-Test service: user front-end . . . . .	98
4.11	NFS service: drive configuration example . . . . .	100
4.12	Model service: example of the UHF ground station model . . . . .	107
4.13	Database service: loading remote history packets with Compass Operations front-end . . . . .	118
4.14	Registry service: example of the local Service map (Compass Operations front-end) . . . . .	120
4.15	Advanced I2C communication . . . . .	122
5.1	Coverage of the space segment chapter . . . . .	125
5.2	Modular pico-satellite bus (image source: [BS17]) . . . . .	126
5.3	Subsystem configuration of one NetSat satellite. Yellow boxes denote Compass OS enabled components . . . . .	129
5.4	Comparison of the embedded and Java-based Compass implementations . .	131
5.5	Conversion from UWE-3 software to Compass middleware . . . . .	132
5.6	Different domains of the Compass OS File System implementation . . . . .	133
5.7	Available drives on the NetSat OBC subsystem visualized in the File View of the Compass Operations front-end . . . . .	133
5.8	Configuration of the NetSat OBC routing table with static entries . . . . .	136

5.9	Command definition one-liner . . . . .	137
5.10	Model group and value definition with a get hook . . . . .	137
5.11	Model examples of NetSat-2 OBC and Panel +X subsystems . . . . .	138
5.12	Unit test definition . . . . .	139
5.13	UnitTest example . . . . .	140
5.14	Download remote files with File View in the Compass Operations front-end	141
5.15	Tiny development cycle . . . . .	142
5.16	Tiny instruction set shown in the IDE's help window . . . . .	144
5.17	Tiny IDE with a source editor (top left), compiled byte-code (bottom) and de-compiled code (top right) . . . . .	145
5.18	Tiny distributed execution . . . . .	148
5.19	Tiny remote code execution . . . . .	149
6.1	Coverage of the ground chapter . . . . .	151
6.2	Ground systems overview before this thesis (spring 2014) . . . . .	152
6.3	Ground systems overview at the time of writing (mid 2019). Left: Univer- sity's sub-network, right: ZfT's sub-network . . . . .	153
6.4	Structure of the CompassNode software . . . . .	156
6.5	Ground Station of the University of Würzburg . . . . .	157
6.6	Software structure of the Ground Station server . . . . .	158
6.7	Model service used to monitor and control a ground station . . . . .	159
6.8	Compass Operations front-end: selection of a mission network entry point.	160
6.9	Model service used to monitor and control a mission server . . . . .	161
6.10	Visualization of UWE-4 beacons with SatNOGS dashboard . . . . .	162
6.11	Compass packets injected by external ground stations . . . . .	163
6.12	Receive locations of UWE-3 packets submitted by radio amateurs . . . . .	163
6.13	Compass Operations front-end . . . . .	165
6.14	Compass Drop-Down Button with some of the available preferences . . . . .	166
6.15	Node selector in a single view . . . . .	166
6.16	Global node selector . . . . .	167
6.17	Network View with direct right-click node control . . . . .	167
6.18	Nodes View buttons . . . . .	168
6.19	Packet View with three collapsible groups: service statistics, packet creator and packet list . . . . .	169
6.20	Compass Packet View buttons . . . . .	169
6.21	Packet View: tools and recording menus . . . . .	169
6.22	Packet View: available actions for selected packets . . . . .	170
6.23	DTN Activation and configuration . . . . .	171
6.24	Command View with selectable human-readable services: Command ser- vice, Chat service, Tiny service, Echo service . . . . .	172
6.25	Command execution using the Model View . . . . .	172
6.26	Model View buttons . . . . .	173
6.27	Model View with currently known (cached) Compass network model . . . . .	174
6.28	Model View: NetSat OBC model shown with (left) and without (right) <i>GUI</i> <i>hints</i> . . . . .	174
6.29	Model View: model subscription menu . . . . .	175
6.30	Available Chunk States . . . . .	176

6.31	Uplink dialog and Uplink View (left), Downlink dialog and Downlink View (right) . . . . .	176
6.32	Unit Testing View with several executed tests . . . . .	177
6.33	Visualization of the ground station parameters . . . . .	177
6.34	Visualization of the satellite's values . . . . .	178
6.35	Auto-detected TLEs by the Schedule View . . . . .	179
6.36	Schedule View: configuration menu . . . . .	179
6.37	Visualization of overpasses and auto-operations tasks . . . . .	180
6.38	Echo View with determined round-trip times and generated report in the second view's tab . . . . .	181
6.39	Tiny View . . . . .	182
6.40	Tiny View dialogs . . . . .	182
6.41	Opened Tiny compiler . . . . .	183
6.42	Scheduler View buttons . . . . .	183
6.43	Scheduler View used for auto operations . . . . .	184
6.44	Scheduler View: task creation in the scheduler's root (left) and inside an existing job . . . . .	185
6.45	Scheduler View: task modification options . . . . .	185
6.46	Scheduler View: recorded tasks . . . . .	186
7.1	Agile Software Development . . . . .	188
7.2	Test results: serial (right), local TCP/C (middle), local TCP/Java (right) .	190
7.3	Test results: LAN (right), WAN (middle), Radio (right) . . . . .	190
7.4	3D model of the Marienberg Fortress in Würzburg being moved on top of the <i>Stäbli</i> mobile platform . . . . .	192
7.5	Development stages as formalized by the <i>modified</i> MDB approach from section 3.1.6 . . . . .	193
7.6	"U-shaped" (front) and "C-shaped" (back) motion simulators . . . . .	194
7.7	Communication loop during the sensor calibration . . . . .	194
7.8	Turntable visualisation in the Compass Operations front-end with the Nodes View showing the mission network during the calibration. The 3D visualization of the turntables was developed by Oliver Ruf . . . . .	195
7.9	UWE team performing pre-flight sensor calibration of the UWE-4 nanosatellite . . . . .	196
7.10	Attitude Control architecture implemented on the ADCS and the panels with appropriate timescales. Tiny is employed as flexible high-level controller. Image used with permission from Philip Bangert . . . . .	198
7.11	Protocol settings for satellite modules in the Ground Station server software (section 6.3) . . . . .	202
7.12	Wall-mounted monitor in all ZfT's offices as a part of the distributed mission control approach . . . . .	202
8.1	Browsing the report directory ( <b>Traffic Post-Processing</b> ) created by the automatic post-processing function of the Compass Operations front-end . .	206
8.2	Operations front-end of an expert at ZfT accessing the NetSat EM model at IRS . . . . .	207
8.3	Pre-flight LEOP exercise: all four NetSat satellites shortly after the antenna deployment . . . . .	207

---

8.4	Pre-flight LEOP exercise: appearance of all four NetSat satellites in the Node View after beacon reception . . . . .	208
8.5	All four NetSat CubeSats ready for launch . . . . .	209
8.6	Top: Real-life example of auto-operations recorded for all NetSat satellites (NS4 job is shown in recording mode). Bottom: Visualization of all overpasses and auto-operations tasks . . . . .	212
8.7	Left: Visualization of NetSat-3 network knowledge in the Operations front-end by using the received network beacon (2020/10/10 09:59). Right: Contents of the same beacon converted to human-readable form . . . . .	215
8.8	NetSat formation at four different time points: 19 October 2020, 31 October 2020, 8 November 2020 and 15 November 2020. Visualized in the <i>Formation View</i> of the Compass front-end . . . . .	217
8.9	Detected ISL activity from the point of view of the NetSat-1, NetSat-2, NetSat-3 and NetSat-4 respectively . . . . .	218
9.1	Example of the dynamic mission network, shown in the Operations front-end, during the NetSat overpass. . . . .	221
9.2	Image showing snowy Alps. Taken by NetSat-3 on 13 December 2020 . . . . .	222
9.3	Comparison of the relative packet and data traffic of different services used in UWE-3, UWE-4 and NetSat missions. The analysis was performed using the entire recorded space-ground traffic – in total over 1.5 million packets – of the corresponding missions. . . . .	225
4	Compass Operations set-up for Ground Support . . . . .	235
5	Compass Operations set-up of a Test Engineer . . . . .	236
6	Compass Operations set-up of a Attitude/Orbit Control Engineer . . . . .	236
7	Recorded packet traffic of the UWE-3 mission . . . . .	237
8	Recorded data traffic of the UWE-3 mission in bytes . . . . .	238
9	Recorded packet traffic of the UWE-4 mission . . . . .	238
10	Recorded data traffic of the UWE-4 mission in bytes . . . . .	239
11	Recorded packet traffic of the NetSat mission . . . . .	239
12	Recorded data traffic of the NetSat mission in bytes . . . . .	240
13	Tiny script used to fix minor bugs on all four NetSat satellites . . . . .	242
14	NetSat-4 accessible via NetSat-2 during the active tracking of the NetSat-2 satellite . . . . .	243



# List of Tables

2.1	Summary of protocol requirements . . . . .	14
2.2	Summary of protocol service requirements for space and ground systems . .	17
2.3	Embedded software requirements as derived from protocol requirements . .	19
2.4	Ground middleware requirements . . . . .	21
2.5	Overview of IoT missions . . . . .	27
2.6	Standard PUS services . . . . .	35
4.1	Supported channels on different platforms . . . . .	74
4.2	Compass packet description . . . . .	76
4.3	Header bytes, with only first byte being mandatory . . . . .	76
4.4	First header byte (mandatory) . . . . .	76
4.5	Second header byte (optional) . . . . .	77
4.6	Third header byte (optional, advanced functions) . . . . .	77
4.7	Payload size field setting . . . . .	78
4.8	Routing Entry if Routing bit is disabled . . . . .	79
4.9	Encryption info byte . . . . .	82
4.10	Currently Implemented services . . . . .	85
4.11	Network service packets . . . . .	87
4.12	Network service: beacon request . . . . .	87
4.13	Network service: (extended) beacon . . . . .	89
4.14	Command service: command request and possible answers . . . . .	91
4.15	Command service: command list request and the corresponding answer . .	91
4.16	Downlink API request . . . . .	93
4.17	Downlink API answer . . . . .	93
4.18	Uplink API Chunk . . . . .	96
4.19	Uplink API State request . . . . .	96
4.20	Uplink API State answer . . . . .	96
4.21	Merge finished packet . . . . .	96
4.22	Log service: Message with optional Severy-String . . . . .	97
4.23	Unit-Test service: test entry description . . . . .	99
4.24	Unit-Test service: execute selected tests and stop execution . . . . .	99
4.25	Unit-Test service: progress packet . . . . .	99
4.26	NFS service: Create file . . . . .	101
4.27	NFS service: Append file . . . . .	101
4.28	NFS service: Delete file . . . . .	101
4.29	NFS service: Format drive . . . . .	101
4.30	NFS service: Close file . . . . .	101

4.31	NFS service: Cut file . . . . .	102
4.33	Tiny service: Execute Tiny byte-code . . . . .	102
4.32	Tiny service packets . . . . .	103
4.34	Tiny service: Execution Answer and Value Types . . . . .	103
4.35	Model service: Get function list (left) and list answer (right) . . . . .	104
4.36	Tiny service: List Tiny threads . . . . .	104
4.38	Tiny service: Add thread . . . . .	104
4.37	Tiny service: Tiny threads List . . . . .	105
4.39	Tiny service: Delete thread . . . . .	106
4.40	Tiny service: Update thread . . . . .	106
4.41	Tiny service: Thread result . . . . .	106
4.42	Tiny service: Load thread from file . . . . .	106
4.43	Tiny service: Save thread to file . . . . .	106
4.44	Model type+length encoding . . . . .	108
4.45	Data Types . . . . .	109
4.46	Model service: local (left) and addressed (right) packet types . . . . .	110
4.47	Model service: Get local (left) and addressed model list . . . . .	110
4.48	Model service: Model list entry . . . . .	111
4.49	Model service: GUI hints of a list entry . . . . .	112
4.50	Model service: set addressed (2/5) or local (20/50) value with or without updating time . . . . .	112
4.51	Model service: get addressed (3) or local (30) value . . . . .	113
4.52	Model service: get addressed (6) or local (60) value . . . . .	113
4.53	Model service: set addressed (7) or local (70) group . . . . .	113
4.54	Model service: get addressed (8) or local (80) group value . . . . .	114
4.55	ARR service: <i>Clear</i> request (left) and <i>Clear Done</i> packet (right) . . . . .	114
4.57	ARR service: single task entry . . . . .	114
4.56	ARR service: request tasks (left) and task entry (right) packet . . . . .	115
4.58	ARR service: create new reporting task . . . . .	115
4.59	ARR service: Successfully created reporting task . . . . .	115
4.60	ARR service: create new recording task . . . . .	116
4.61	ARR service: Successfully created recording task . . . . .	116
4.62	ARR service: delete Task by its TaskID (left) and answer on success (right) . . . . .	117
4.63	ARR service: delete Task by the model ID (left) and answer on success (right) . . . . .	117
4.64	Database service: request and answer packet . . . . .	118
4.65	Registry service: list all, list map and list value . . . . .	119
4.66	Registry service: clear map, delete value and create/update value . . . . .	119
4.67	Registry service: reserved table IDs . . . . .	119
4.68	Tunnel service packet . . . . .	120
4.69	Packet detection strategies . . . . .	121
4.70	SLIP Protocol (left) and encoding rules (right) . . . . .	121
4.71	I2C Master Packet . . . . .	123
4.72	I2C Slave Response . . . . .	124
4.73	I2C Slave Urgent Response . . . . .	124



---

5.1	Currently developed satellite subsystems and their amount in <i>one</i> NetSat satellite. $\mu\text{C}$ shows total number of Compass OS enabled microcontrollers on one subsystem . . . . .	127
5.2	Different MCU types . . . . .	128
5.3	Available channels in Compass OS . . . . .	134
5.4	Services implemented in Compass OS and in CompassNode . . . . .	135
5.5	Routing entries example . . . . .	135
6.1	Java-ability of ground systems . . . . .	155
7.1	Tests with different channel types and the theoretical speed . . . . .	189
7.2	Round-trip-time (RTT) results of the communication channels . . . . .	192
7.3	Ground Station Server: stored ground-space traffic for all three missions (2020/11/13). <i>Ext</i> denotes packets being received from remote stations . . .	200
8.1	Tiny interpreter: available external functions on all UWE-4 and NetSat subsystems . . . . .	213

Major parts of this work was supported by the *European Research Council* (ERC) Grant “NetSat” under the Grant Agreement *No. 320377*. The author also appreciated the support for UWE-4 by German national space agency DLR by funding from the Federal Ministry of Economic Affairs and Energy by approval from German Parliament with reference 50 RU 1501.



# Die Schriftenreihe

wird vom Lehrstuhl für Informatik VII: Robotik und Telematik der Universität Würzburg herausgegeben und präsentiert innovative Forschung aus den Bereichen der Robotik und der Telematik.

Die Kombination fortgeschrittener Informationsverarbeitungsmethoden mit Verfahren der Regelungstechnik eröffnet hier interessante Forschungs- und Anwendungsperspektiven. Es werden dabei folgende interdisziplinäre Aufgabenschwerpunkte bearbeitet:

- Robotik und Mechatronik: Kombination von Informatik, Elektronik, Mechanik, Sensorik, Regelungs- und Steuerungstechnik, um Roboter adaptiv und flexibel ihrer Arbeitsumgebung anzupassen.
- Telematik: Integration von Telekommunikation, Informatik und Steuerungstechnik, um Dienstleistungen an entfernten Standorten zu erbringen.

Anwendungsschwerpunkte sind u.a. mobile Roboter, Tele-Robotik, Raumfahrtsysteme und Medizin-Robotik.

Lehrstuhl Informatik VII  
Robotik und Telematik  
Am Hubland  
D-97074 Würzburg

Tel.: +49 (0) 931 - 31 - 86678  
Fax: +49 (0) 931 - 31 - 86679

[schi@informatik.uni-wuerzburg.de](mailto:schi@informatik.uni-wuerzburg.de)  
<http://www7.informatik.uni-wuerzburg.de>

Dieses Dokument wird bereitgestellt  
durch den Online-Publikationsservice  
der Universität Würzburg.

Universitätsbibliothek Würzburg  
Am Hubland  
D-97074 Würzburg

Tel.: +49 (0) 931 - 31 - 85906

[opus@bibliothek.uni-wuerzburg.de](mailto:opus@bibliothek.uni-wuerzburg.de)  
<https://opus.bibliothek.uni-wuerzburg.de>

ISSN: 1868-7474 (online)  
ISSN: 1868-7466 (print)  
ISBN: 978-3-945459-38-6 (online)

## Zitation dieser Publikation

DOMBROVSKI, V. (2021). Software Framework to Support Operations of Nanosatellite Formations. Schriftenreihe Würzburger Forschungsberichte in Robotik und Telematik, Band 23. Würzburg: Universität Würzburg. DOI: 10.25972/OPUS-24931

Dissertation an der Universität Würzburg im Rahmen der Graduate School of Science and Technology