

Event-driven Software-Architecture for Delay- and Disruption-Tolerant Networking

Der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Johannes Morgenroth
geboren am 28.08.1981
in Braunschweig

Eingereicht am: 19.12.2014
Disputation am: 17.07.2015
1. Referent: Prof. Dr.-Ing. Lars Wolf
2. Referent: Prof. Dr.-Ing. Klaus Wehrle

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen
Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
www.dnb.de abrufbar.

Diss., Technische Universität Carolo Wilhelmina zu Braunschweig, 2015

© 2015 Johannes Morgenroth

Herstellung:
BoD – Books on Demand, Norderstedt

BoD Ref. 0011632925

Abstract

Continuous end-to-end connectivity is not available all the time, not even in wired networks. Thus, the »Off-line first« paradigm is applied to applications of mobile devices to deal with intermittent connectivity without annoying the users every time the network is going to be flaky. But since mobile devices, particularly smartphones, are typically bound to the Internet, they become almost useless as soon as their connectivity is gone. Delay- and Disruption-Tolerant Networking (DTN) allows devices to communicate even if there is no continuous path to the destination by replacing the end-to-end semantics with a hop-by-hop store-carry-and-forward approach.

Since existing implementations of DTN software suffer from various limitations, this thesis presents an event-driven software architecture for Delay- and Disruption-Tolerant Networking and a lean, lightweight, and extensible implementation of a bundle node. Beside general considerations of requirements and use-cases, special attention is paid to the performance and portability to embedded platforms. Further, a set of helpful extensions is selected and integrated to gain a bundle node implementation which is ready to get deployed to common scenarios.

In a comprehensive description of the architecture and the underlying design decisions, this work focuses on eliminating weaknesses of the Bundle Protocol (RFC 5050). One of these is the dependency on synchronized clocks. Thus, this work takes a closer look on that requirement and presents approaches to bypass that dependency for some cases. For scenarios which require synchronized clocks, an approach is presented to distribute time information which is used to adjust the individual clock of nodes. To compare the accuracy of time information provided by each node, this approach introduces a clock rating. Additionally, a self-aligning algorithm is used to automatically adjust the node's clock rating parameters according to the estimated accuracy of the node's clock.

In an evaluation, the general portability of the bundle node software is proven by porting it to various systems. Further, a performance analysis compares the new implementation with existing software. To perform an evaluation of the time-synchronization algorithm, the ONE simulator is modified to provide individual clocks with randomized clock errors for every node. Additionally, a specialized

testbed, called HYDRA, is being developed to test the implementation of the time-synchronization approach in real software. HYDRA instantiates virtualized nodes running a complete operating system and provides a way to test real software in large DTN scenarios. Both the simulation and the emulation in HYDRA show that the algorithm for time-synchronization can provide an adequate accuracy depending on the inter-contact times.

Kurzfassung

Eine kontinuierliche Ende-zu-Ende-Konnektivität ist nicht immer verfügbar, nicht einmal in drahtgebundenen Netzen. Daher wird das »Off-line first« Paradigma bei Anwendungen auf Mobilgeräten angewendet um die Anwender nicht zu irritieren, falls die Verbindung unzuverlässig wird. Aber da Mobilgeräte, insbesondere Smartphones, typischerweise abhängig von dem Internet sind, werden sie fast nutzlos sobald keine Verbindung mehr existiert. Verzögerungs- und unterbrechungstolerante Kommunikation (DTN) ersetzt die Ende-zu-Ende-Semantik mit einem Hop-by-Hop Store-Carry-and-Forward Ansatz und erlaubt es so Geräten miteinander zu kommunizieren, auch wenn es keinen kontinuierlichen Pfad gibt.

Da bestehende DTN Implementierungen unter verschiedenen Einschränkungen leiden, stellt diese Arbeit eine ereignisgesteuerte Software-Architektur für Verzögerungs- und unterbrechungstolerante Netze sowie eine schlanke, leichte und erweiterbare Implementierung eines Bündelknotens vor. Neben den allgemeinen Überlegungen zu den Anforderungen und den Anwendungsfällen wird besonderes Augenmerk auf die Leistung und die Portabilität zu eingebetteten Systemen gelegt. Ferner wird eine Reihe von hilfreichen Erweiterungen ausgewählt und integriert, um eine Implementierung eines Bündelknotens zu erhalten, welche bereit ist in üblichen Szenarien eingesetzt zu werden.

In einer umfassenden Beschreibung der Architektur und den zugrunde liegenden Design-Entscheidungen, konzentriert sich diese Arbeit auf die Beseitigung von Schwächen des Bundle Protocols (RFC 5050). Eine davon ist die Abhängigkeit zu synchronisierten Uhren. Daher wirft diese Arbeit einen genaueren Blick auf diese Anforderung und präsentiert Ansätze, um diese Abhängigkeit in einigen Fällen zu umgehen. Für Szenarien die synchronisierte Uhren voraussetzen wird außerdem ein Ansatz vorgestellt, um die Uhren der einzelnen Knoten mit Hilfe von verteilten Zeitinformationen zu korrigieren. Um die Genauigkeit der Zeitinformationen von jedem Knoten vergleichen zu können, wird eine Bewertung der Uhren eingeführt. Zusätzlich wird ein Algorithmus vorgestellt, der die Parameter der Bewertung in Abhängigkeit von der ermittelten Genauigkeit der lokalen Uhr anpasst.

In einer Evaluation wird die allgemeine Portabilität der Software zu verschiedenen Systemen gezeigt. Ferner wird bei einer Performance-Analyse die neue Soft-

ware mit existierenden Implementierungen verglichen. Um eine Evaluation des Zeitsynchronisationsalgorithmus durchzuführen, wird der ONE Simulator so angepasst, dass jeder Knoten eine individuelle Uhr mit zufälligem Fehler besitzt. Außerdem wird eine spezielle Testumgebung namens HYDRA vorgestellt um eine echte Implementierung des Zeitsynchronisationsalgorithmus zu testen. HYDRA instanziert virtualisierte Knoten mit einem kompletten Betriebssystem und bietet die Möglichkeit echte Software in großen DTN Szenarien zu testen. Sowohl die Simulation als auch die Emulation in HYDRA zeigen, dass der Algorithmus für die Zeitsynchronisation eine ausreichende Genauigkeit in Abhängigkeit von Kontakthäufigkeit erreicht.

Danksagung

Nach einigen Jahren interessanter Forschung hatte ich nun die Gelegenheit meine bisherigen gesammelten Erkenntnisse und Erfahrungen in diesem Werk nieder zu schreiben. Auch wenn die Schrift von mir verfasst wurde, haben dennoch viele mir wichtige Menschen einen ganz wesentlichen Beitrag zu dieser Arbeit geleistet. Auf diesem Wege möchte ich all' jenen meinen herzlichsten Dank aussprechen.

Meinem Doktorvater und ehem. Arbeitgeber Prof. Dr.-Ing. Lars Wolf danke ich in erster Linie für die umfangreiche Betreuung und Unterstützung bei der Entwicklung meiner Fähigkeiten und Forscher-Tätigkeiten. Ohne den gebotenen Freiraum kombiniert mit sehr zielgerichteten Stößen in die richtige Richtung, wäre meine Arbeit sicherlich nicht das was sie nun ist. Selbstverständlich danke ich auch für seine Arbeit als Gutachter und Prüfer im Rahmen meiner Promotion. Ebenso danke ich Prof. Dr.-Ing. Klaus Wehrle für seinen Einsatz bei der Begutachtung meiner Arbeit und der abschließenden Prüfung. Meinen ehem. Kollegen des Instituts für Betriebssysteme und Rechnerverbund der TU Braunschweig danke ich für die moralische Unterstützung und für die vielen Diskussionsrunden, welche mir mehr als einmal erlaubt haben einen anderen Blickwinkel einzunehmen.

Meiner Familie und dabei insbesondere meiner Frau, Julia Morgenroth, danke ich für die tatkräftige Unterstützung bei der Bewältigung des Alltags während der anspruchsvollen Phasen meiner Arbeit. Außerdem Danke ich für das mir entgegen gebrachte Verständnis und für die Ermutigungen die mir halfen diese Arbeit schließlich zu vollenden.

Johannes Morgenroth, August 2015

Contents

1	Introduction	1
1.1	Off-line First	1
1.2	Living without Internet	2
1.3	Store, Carry, and Forward	2
1.4	Delay- and Disruption-Tolerant Networking	3
1.5	Outline	4
2	Fundamentals	7
2.1	Applications	7
2.1.1	ZebraNet	7
2.1.2	DakNet	8
2.1.3	N4C	8
2.1.4	Astronaut E-mail Challenge	9
2.1.5	EMMA	10
2.2	Scenarios	11
2.2.1	Space and Interplanetary Networking	11
2.2.2	Military and Tactical Communications	11
2.2.3	Disaster and Emergency Systems	11
2.2.4	Internet Deployment	12
2.2.5	Wireless Sensor-networks	12
2.2.6	Mobile Internet	13
2.3	Bundle Protocol	13
2.3.1	Architecture	14
2.3.2	Addressing	16
2.3.3	Data-structures	17
2.3.4	Fragmentation	22
2.3.5	Administrative Records	23
2.3.6	Custody Transfer	26
2.3.7	Application Agent	27
2.3.8	Security	28
2.3.9	Dependency on Synchronized Clocks	29
2.4	Implementations	29

2.4.1	DTN2 Reference Implementation	30
2.4.2	Interplanetary Overlay Network	31
2.4.3	Postellation	33
2.4.4	Bytewalla	34
2.4.5	Other	35
2.5	Time-synchronization	35
2.5.1	Synchronicity in Computer Networks	36
2.5.2	Challenged Networks	37
2.5.3	Sensor-networks	40
3	Architecture	43
3.1	Considerations	44
3.1.1	Platform Dependencies	45
3.1.2	Processing Overhead	46
3.1.3	Latency and Queuing	50
3.1.4	Resource Constrains	54
3.2	Components	54
3.3	Entities	56
3.4	Event-system	59
3.5	Concurrency	63
3.6	Work-flows	64
3.6.1	Routing Data Exchange	64
3.6.2	Bundle Selection	67
3.6.3	Bundle Forward	69
3.6.4	Bundle Reception	70
3.6.5	Bundle Delivery	71
3.6.6	Bundle Retransmission	71
3.6.7	Multi-stage Connection Set-up	72
3.7	Fragmentation	73
3.8	Neighbor Discovery	74
3.9	Extensions	76
3.9.1	DTN-DHT	76
3.9.2	Streaming	78
3.9.3	Compression Extension Block	80
3.9.4	Tracking Extension Block	82
3.9.5	Security Extensions for the TCP Convergence-Layer	83
4	Implementation	87
4.1	Objectives	87

4.2	Concept	88
4.3	Performance Considerations	92
4.3.1	Data Processing and Policy Checking	92
4.3.2	Serialization and Parsing	94
4.3.3	Routing	94
4.3.4	Application Delivery	96
4.4	Software Structure	97
4.5	Concurrency	99
4.5.1	Mutex	99
4.5.2	MutexLock	99
4.5.3	Conditional	100
4.5.4	Semaphore	101
4.5.5	Thread	101
4.5.6	Queue	102
4.5.7	Reference Counting	103
4.5.8	BLOB	103
4.5.9	Concurrent Read-only Locking	104
4.6	Data Serialization	105
4.7	Event-system	106
4.8	Data Storage	107
4.9	Routing	111
4.9.1	Neighbor Routing	114
4.9.2	Static Routing	119
4.9.3	Flooding	119
4.9.4	Epidemic Routing	120
4.9.5	PRoPHET Routing	120
4.10	Convergence-Layers	124
4.10.1	Neighbor Discovery	125
4.10.2	UDP Convergence-Layer	126
4.10.3	TCP Convergence-Layer	127
4.10.4	Datagram-based Convergence-Layers	129
4.10.5	File Convergence-Layer	136
4.11	Applications	138
4.11.1	Embedded Applications	138
4.11.2	Application Interface	139
4.11.3	Tools	142
4.12	Protocol Extensions	149
4.12.1	Age Extension Block	149
4.12.2	Bundle Encapsulation	151

4.12.3	DTN Scope Control using Hop Limits	153
4.12.4	Bundle Security Protocol	154
4.13	Challenges	158
4.13.1	Time-dependencies	158
4.13.2	Platform-independence	159
4.13.3	Performance	160
4.13.4	Fragmentation	161
4.13.5	Security Policy	162
5	Time-synchronization	165
5.1	Requirements	165
5.1.1	Identification and Referencing	166
5.1.2	Validity and Expiry	166
5.1.3	Routing	167
5.1.4	Energy-saving	168
5.1.5	Security	168
5.1.6	Limitations	169
5.1.7	Alternatives	170
5.2	Time-reference Distribution	171
5.2.1	Algorithm	172
5.2.2	Delay-tolerant Network Time Protocol	174
5.2.3	Security	177
5.2.4	Local Clock Adjustment	178
5.2.5	Scope and Network-wide Time	178
5.2.6	Implementation	179
6	Evaluation	181
6.1	Portability	181
6.1.1	OpenWrt	181
6.1.2	OS/X®	183
6.1.3	Android™	186
6.1.4	Microsoft® Windows®	188
6.2	Performance	189
6.2.1	Experimental Setup	189
6.2.2	API and Bundle Storage Performance	190
6.2.3	Network Throughput	198
6.2.4	TCP-CL Segment Length	205
6.2.5	DTN2 Throughput Variances	207
6.2.6	ION Storage Variants	208

6.2.7	Conclusion	209
6.3	Continuous Integration	210
6.3.1	Building	210
6.3.2	Unit-testing	212
6.3.3	Performance Testing	213
6.4	Time-synchronization	214
6.4.1	ONE Simulator	214
6.4.2	HYDRA	220
7	Conclusion	229
7.1	Connecting the Dots	229
7.2	Contribution	230
7.3	Reasonable Limitations	233
7.4	Future Work	234

Acronyms

AA Application Agent	15
ACK Acknowledgement	130
ADU Application Data Unit	16
AEB Age Extension Block	39
AES Advanced Encryption Standard	158
API Application Programming Interface	30
ARP Address Resolution Protocol	84
ASB Abstract Security Block	157
BAB Bundle Authentication Block	155
BLOB Binary Large Object	57
BPA Bundle Protocol Agent	15
BPB Bundle Payload Block	81
BP Bundle Protocol	4
BSP Bundle Security Protocol Specification	78
BSR Bundle Status Report	19
BT Bit-Torrent	76
CA Certificate Authority	168
CBHE Compressed Bundle Header Encoding	32
CEB Compression Extension Block	81
CI Continuous Integration	210
CLA Convergence Layer Adapter	15

CPU Central Processing Unit	106
CRC Cyclic Redundancy Check	132
DCCP Datagram Congestion Control Protocol	32
DCS Distributed Asynchronous Clock Synchronization	40
DHCP Dynamic Host Configuration Protocol	133
DHT Distributed Hash Table	76
DINET Deep Impact Network Experiment	37
DNS Domain Name System	77
DTP Double-pairwise Time Protocol	39
DTLSR Delay Tolerant Link State Routing	31
DTN Delay- and Disruption-Tolerant Network	3
DVB Digital Video Broadcasting	79
EID Endpoint Identifier	16
ESB Extension Security Block	83
EWMA Exponential Weighted Moving Average	132
FCS Frame Check Sequence	135
GCM Galois/Counter Mode	158
GPS Global Position System	10
GUI Graphical User Interface	46
HMAC Keyed-Hash Message Authentication Code	157
IANA Internet Assigned Numbers Authority	16
ICMP Internet Control Message Protocol	23
ION Interplanetary Overlay Network	10
IP Internet Protocol	3
IPC inter-process communication	191

IPND IP Neighbor Discovery	74
ISS International Space Station	9
JNI Java Native Interface	186
LTP Licklider Transmission Protocol	30
MAC Message Authentication Code	84
MSB Most Significant Bit	18
MTU Maximum Transmission Unit	126
NACK Negative Acknowledgment	130
NAT Network Address Translation	84
NBF Neighborhood Bloom Filter	75
NDK Native Development Kit	186
NORM Nack-Oriented Reliable Multicast	30
NPTL Native POSIX Thread Library	182
NTP Network Time Protocol	37
ONE Opportunistic Network Environment	214
PAN Personal Area Network	135
PCB Payload Confidential Block	154
PDU Protocol Data Unit	15
PIB Payload Integrity Block	152
PKI Public-Key-Infrastructure	168
PRoPHET Probabilistic Routing Protocol using History of Encounters and Transitivity	8
RPA Routing Protocol Agent	121
RTT Round-Trip Time	80
SCHL Scope Control using Hop Limits	65

SDNV Self-Delimiting Numeric Value	18
SDU Service Data Unit	57
SNMP Simple Network Management Protocol	35
SQL Structured Query Language	111
SSP scheme-specific part	16
STL Standard Template Library	31
TCP-CL TCP Convergence-Layer	78
TCP Transmission Control Protocol	30
TCS Tetherless Computing Architecture	31
TEB Tracking Extension Block	83
TLS Transport Layer Security	33
TLV Type Length Value	157
TTL Time to Live	153
UDP User Datagram Protocol	30
URI Uniform Resource Identifier	16
UTC Coordinated Universal Time	41
WSN Wireless Sensor Network	12
XML Extensible Markup Language	30

1 Introduction

Although mobile Internet connectivity based on infrastructure networks is growing and being improved constantly, there are still gaps in the coverage. Closing those gaps is not only a matter of funding. Local policies, the cabin of a train, or simply walls can effectively eliminate any kind of connectivity. Thus, even if the availability of infrastructure-based mobile communication is getting better, there are gaps and ever will be. Technology supporting vertical hand-over was invented to mask these gaps and allows mobile devices to switch to a different technology in order to stay connected to the Internet, if the primary connectivity is gone. But as soon as there is no alternative to switch to, mobile devices become almost useless.

1.1 Off-line First

Actually, there is an upcoming trend to design mobile applications using the »Off-line first« [Fey13] principle. Accordingly, applications should be designed in a way, that most of the functionality is available even without any kind of connectivity. In order to do that, a working set of data must be synchronized to local caches when there is an opportunity to communicate to the corresponding server. In all cases, the applications should not disturb the user-experience while it is disconnected from the Internet. Synchronization with a server must be a background task and in the best case the user does not even notice the absence of connectivity.

Despite this approach is quite plausible for applications, because the same principle is state of the art on Desktop systems (e.g. Email applications), it is less common for web applications. Recently, the W3C published a working draft for so called *Service Workers* [RS14, Arc14]. This specification allows web-pages to register a data-cache which is refreshed, if the device has Internet connectivity. If the user opens the web-page without being connected, the cached content is used to render the web-page and at least a part of the web-application stays functional.

Accessing cached content on disconnection is not a nouveau approach. It is best practice in well-designed applications, but a developer needs to realize a concept to organize the internal cache structure, synchronization of the content, and the detection of connectivity for every single application. Since this is a reoccurring task, it would be good if that could be handled by a layer underneath the application.

1.2 Living without Internet

Even if applications stay functional using good caching strategies, they are almost completely isolated as soon as there is no connectivity to their associated server. In general, peer-to-peer protocols are mentioned to overcome this limitation by exchanging data directly with peers in their communication range. In addition to communicate directly with peers, there exist approaches to form meshed networks. Those allow a sender to utilize peers within its range as intermediate hop to reach nodes, even if they are not in range of the sender. However, many peer-to-peer approaches need a bootstrap strategy which often involves a server located in the Internet. Thus, their autonomy is questionable.

To keep an application functional, even if the Internet-connectivity is gone, is a quite demanding task. The use-cases of such an application must be designed to interact with peers directly instead of only talking to or through a central server. Additionally, the underlying networking technology has to be utilized to form a peer-to-peer network which allows an application to forward and deliver data to other applications. Assuming that ideally every or at least many applications should be capable in peer-to-peer communication, this is also a task which should be solved once by a generic implementation.

1.3 Store, Carry, and Forward

All common networking technologies are based on continuous end-to-end paths. Received data are forwarded immediately towards the destination, or dropped if no direct path is known. Applications must know, or at least detect, if the receiver of some data is connected to a network or not. Communicating with nodes which are connected intermittently is barely possible.

A traditional network scenario with two alternately connected nodes offers no way to exchange data between them. The only way to achieve that would be to store data on an intermediate node which acts as proxy. The store-carry-and-forward paradigm is based on a similar idea. Analogous to parcel services, where the postman retain packages until he can deliver them to the recipient, nodes on such a network do not drop undeliverable packets immediately. Instead, routing algorithms are used to decide whether there might be an opportunity in the future to forward the packets towards the destination. If yes, the packets are stored on the node until they are forwarded or a policy dictates to drop them. By exploiting the mobility of nodes, it is even possible to move packets physically towards to the destination. A simple scenario is a network consisting of three nodes A, B, and C. A and C are not within communication range of each other and B can not

communicate with A and C simultaneously. Instead, B moves around and shuttles between A and C. Thus, once it is connected to A and later to C and then again to A. Using common networking approaches it would not be possible for node A to send data to C and vice versa. The store-carry-and-forward paradigm would solve that problem by storing all packets on node B which forwards them as soon as it is in communication range of the destination.

1.4 Delay- and Disruption-Tolerant Networking

Delay- and Disruption-Tolerant Networking is an approach designed for space missions and covers the loss of connectivity by breaking up the end-to-end semantics of common protocols such as Internet Protocol (IP) with a hop-by-hop store-carry-and-forward approach [CBH⁺07]. Nodes carry messages called »bundles« and forward them to other nodes they encounter later. Using that approach, applications are able to send their bundles into the network even if there is no continuous path to the destination. Although originally developed for space-crafts and satellites, the approach also solves terrestrial issues with intermittent connectivity. The difference here is that the transmission delay between the nodes is much lower than in space-scenarios with long-distance transmissions. For that reason, the term is often simplified to Disruption-Tolerant Networking [Falo3].

The *Bundle Protocol Specification* [SB07] specifies how an implementation of a Delay- and Disruption-Tolerant Network (DTN) should behave and also defines a format for the exchange of bundles to gain interoperability between different implementations. Based on that document, several well-known implementations exist. However, none of them is sufficient lightweight and portable enough to get adapted to requirements of embedded systems. Those systems are an important target-platform for many applications and usage scenarios, for example if mobile devices or vehicles should be utilized to form a DTN. The project »EMMA« [LDPLo6] utilizes a vehicular network based on the public transport system to measure the air pollution in urban areas. For that purpose, each vehicle performs measurements and forwards the collected data using a DTN approach to a sink. The authors of the paper performed their initial experiments using standard laptop hardware, but also stated that it is necessary to target embedded systems for productive operations in order to lower the energy consumption as well as the hardware costs. For that reason, the focus of this work is on providing a lightweight, modular, and highly portable implementation of a DTN communication stack. Moreover, this work combines proposed and nouveau extensions in addition to the basic specification to address open issues such as the dependency to time-synchronization. The objective here is to find approaches to mitigate the

dependency and provide a mechanism to synchronize clocks, even if there is no continuous end-to-end path between the nodes. Further, the implementation should provide security features for integrity, confidentiality, authenticity and non-repudiation for all transmissions. Therefore, the existing security extensions for the *Bundle Protocol* have to be considered in regard to uncovered attacks.

Last but not least, the development process of such an implementation includes thoughts on the architecture, which has to be thin but flexible enough to stay extensible and allow replacements for alternative implementations of specific functionalities. To prove stability and performance of the new software, several performance measurements are part of this work.

1.5 Outline

The Chapter 2 starts with fundamentals important for this work. At first, we identify applications and scenarios in which a DTN is used to solve connectivity challenges. Then, we present the Bundle Protocol (BP), which is a protocol specifically designed for DTNs, and existing implementations of that protocol. The last part of that chapter is dedicated to the issue of time-synchronization and how this is bypassed or solved in various networks.

Based on the fundamentals, the Chapter 3 presents further considerations in regard to platform dependencies and performance relevant characteristics of DTNs. These considerations are used to design a lean, lightweight, and extensible architecture for bundle nodes. First, an overview is given for the necessary components and entities. As next, concepts and work-flows are presented to realize an efficient bundle node according to the previous findings.

Chapter 4 explains design decisions made for the aspired bundle node implementation. It focuses on performance, extensibility, and efficient usage of the selected programming language. Additionally, implementation details as well as necessary restrictions of existing extensions, which are not part of this work, are presented. The chapter concludes with the major challenges encountered during the development. One important challenge is the dependency on synchronized clocks. Chapter 5 is dedicated to that topic. It discusses the reasons for that dependency in detail and proposes an algorithm to distribute time information in a DTN. This information is augmented by a rating based on the encounter distance to the master clock source and used to adjust the clocks of the nodes. Further, the chapter presents an approach to measure clock differences between two nodes by exchanging bundles. Finally, details on the integration into the previously developed software are given.

The evaluation in Chapter 6 presents the testing procedures used during the development. They measure performance as well as correctness using a continuous integration system. The second part of the chapter is dedicated to the time-synchronization approach introduced in the previous chapter. First, the algorithm is evaluated in a DTN simulator and then the actual implementation of that approach is tested within a specialized testbed. Finally, we conclude the work presented in this thesis in Chapter 7 and discuss future work.

2 Fundamentals

Before we start to develop a new system, we take a closer look at the fundamentals of our intension. Based on the *State of the Art* document [Dav09] of the N4C project, we start with a study of existing applications and scenarios for DTNs. The resulting summary allows us to estimate the requirements for a generic system for DTN communication. As next, we will present and explain the *Bundle Protocol* which is the standard protocol for DTNs and already used by many experimental implementations to realize DTN communication. These implementations will be introduced in Section 2.4. To estimate how the implementations would fit to the discussed scenarios, we tried to figure out all their specific features and constraints. In Section 2.5, we will discuss minds and thoughts related to clock respectively time-synchronization in computer networks. Since this is a very challenging task and reoccurring issue in DTNs, the Chapter 5 is dedicated to present an approach to synchronize clocks between nodes in a DTN.

2.1 Applications

This section presents existing examples of applications designed on-top of DTN technology. Although most of them are motivated by research, they are practical use-cases in scenarios where classical networking would fail or require a lot more effort to deploy.

2.1.1 ZebraNet

In 2002, researchers at the Princeton University (USA) started to explore wireless protocols and position-aware computation from a power-efficient perspective [JOW⁺02]. By attaching position tracking hardware to animals, their movement and behavior should be observed across large regions with little communication infrastructure. Each node consists of a CPU, flash storage, a radio communication device and a GPS receiver. Tucked in a collar, these components are determined to be worn by African zebras. Since there is no cellular service or broadcast communication covering the region where these animals are studied, the researchers have chosen to implement a peer-to-peer approach to deliver the data, gathered by the nodes in a store-and-forward fashion, to a base station located at

the biologists who also move around during their studies. Since existing protocols were only capable in delivering data to a central sink, a nouveau approach was necessary. To achieve a high »data homing« success rate, the routing approaches direct delivery, flooding, and a history-based protocol similar to Probabilistic Routing Protocol using History of Encounters and Transitivity (PRoPHET) [LDS03] has been evaluated and compared in simulations. In early 2004, the *ZebraNet* group headed to Kenya for its first test deployment.

2.1.2 DakNet

The *DakNet* [PFH04] has been initiated in 2002 by the MIT Media Lab, Cambridge, USA and the Media lab Asia, India. It addressed communications challenged communities in rural areas of developing countries, beginning with India, and was commercialized by the First Mile Solutions, Cambridge, USA. By exploiting existing communications and transportation infrastructure, data was ferried using a store-and-forward approach between kiosks within outlying villages to form some sort of Intranet. Additionally, hubs at places with Internet connectivity (e.g. a post office or a cybercafé) act as gateway for non-real-time Internet access and E-Mail traffic. Buses, motorcycle, or even bicycles were equipped with mobile access points (MAPs) and used to ferry the data. These automatically transfer data using low-cost Wi-Fi radio transceivers over short point-to-point links with kiosks and hubs.

The business case was to provide network connectivity and asynchronous services to outlying villages where permanent connection or even a telephone would be infeasible. The *DakNet* offered *E-Mail services*, *VoiceMail services* and *Cached Web*. Using computers located within Kiosks, the villagers were able to send and receive E-Mails. Even if they were not connected to the Internet, they could communicate with residents of other villages. With the *VoiceMail services*, it was even easier to use this nouveau type of communication. Users do not have to know how to operate a computer, they can simply dial a phone number to send a voice recorded message. Finally, the *Cached Web* offered the knowledge of the Internet to the villagers. Using a Web search engine called *Time Equals Knowledge (TEK)*, users were able to send requests for searches or specific URLs. A service, connected to the Internet, downloaded the requested content, converted it into a more efficient format, and sent it back to the requester.

2.1.3 N4C

With the project *Networking for Communications Challenged Communities (N4C)* [FCB14], a large group of researchers started in summer of 2008 to develop the

Internet for the remote regions where it is not simple, or not cheap or, even, not feasible to deploy it the usual way. To achieve that goal, they examine existing approaches, standards, software solutions, and hardware-platforms to finally adapt these to match the needs of two test beds in northern Sweden and in the Kočevje region of Slovenia. Both are »communications challenged« in the sense that they have little or none of infrastructure that is needed to support today's conventional wired and wireless Internet communications. The low population density there is mostly transitory or semi-nomadic that there is little fixed accommodation to which cables could be laid. The terrain is mountainous which makes cellular telephone networks unfeasible.

Satellite communications as alternative offer only low bandwidth for very high cost both in terms of equipment purchase and ongoing per-octet traffic costs which is not compatible with the relatively low financial return on the industries indigenous to the regions. Moreover, arctic regions in northern Sweden restricts availability to communications through polar orbit satellites due to their high latitude. As result, only an intermittent and uncertain service is available. Another alternative may be the wider area broadcast technologies such as Digital Radio Mondiale (DRM) which might offer a somewhat higher bandwidth but unidirectional service using frequencies that are not limited to line of sight. However, such services have not yet been implemented in the relevant regions and the economics are uncertain.

Until summer 2011, the researchers deployed and experimented with DTN nodes in these regions. Helicopters and cars were used to ferry data between stations which are deployed in the middle of nowhere. Since there is also no electricity, the nodes are powered by solar panels or wind generators. The developments in the field involved real users. Certainly, these tests were done at small scale but operating under real conditions in scenarios that approximate ordinary life. During the experiments they unveiled open issues and evaluated protocol extensions to solve them. As result, the protocol extensions are published in public standard documents.

2.1.4 Astronaut E-mail Challenge

Astronauts on the International Space Station (ISS) currently use Microsoft Outlook connected to a Microsoft Exchange Server on the ground to send and receive E-Mails. This system uses TCP/IP on links that are delayed and frequently disrupted due to ISS structural blockage and handovers of the Tracking and Data Relay Satellite System (TDRSS). The combination of delay and disruptions causes E-Mail to frequently have problems and become unusable, particularly when send-

ing E-Mails with large attachments, such as pictures or videos. In future, two DTN gateways based on Interplanetary Overlay Network (ION) will be available on the ISS. These are used to transfer DTN traffic between the ISS and the Ground Station over the space-to-ground link.

As part of the NASA's Disruption Tolerant Networking Challenge Series [Bur14b], a solution is being developed to connect the Microsoft Outlook Client with the Microsoft Exchange Server using the *Bundle Protocol*. It should be realized either as plug-in to Outlook or a type of gateway if a plug-in is not feasible. All features of Microsoft Outlook are subject to synchronization but at least E-Mail and calendars are mandatory. The system must deal with unpredictable disruption of up to 4 hours, unpredictable loss of data, and round-trip times on the order of 0.6 s to 1 s.

2.1.5 EMMA

The project *Environmental Monitoring in Metropolitan Areas* (EMMA) [LDPLo6] was designed to fulfill measurements of environmental data in city areas, which has become an important issue for densely populated areas. Those need constant monitoring of environmental conditions to assure the citizen's health as well as compliance with political directives. Since stationary measuring stations are inflexible, cost-intensive and limited to monitoring distinct key areas, this decentralized architecture for environmental monitoring in metropolitan areas promises to perform that task at low-costs using Car2X communication techniques. The system is designed as self-organizing and requires only little to no set-up efforts.

By integrating sensor nodes that communicate with each other into vehicles of the public transportation systems it is possible to cover even large areas with a few devices. These nodes combine measured data with Global Position System (GPS) positioning information and time-stamps. The measured data is forwarded using the store-carry-and-forward principle to a central visualization engine to get analyzed and published. A monitoring instance can react to high pollution levels by e.g. closing streets for trucks. Public transportation systems typically have one or only a few central points where many lines meet. These points are suitable locations for central gateways to the evaluation server and the traffic management center.

The researchers proofed their basic idea and concept in a field test. Further they selected hard- and software to realize a prototype implementation for more advanced testing. The general approach has been evaluated using four prototype nodes equipped with a Wi-Fi transceiver and a GPS receiver.

2.2 Scenarios

All applications presented in the previous section can be categorized into a bunch of scenarios. A generic DTN implementation should be capable in handling all the challenges explained in each scenario.

2.2.1 Space and Interplanetary Networking

Originally, delay-tolerant networking was considered as solution for communication between ground-stations and space crafts (e.g. Space Shuttle). As described in Section 2.1.4, connections in space are often interrupted by a missing line of sight between the communication partners. Additionally, the long signal propagation time makes common approaches for communication unsuitable.

Different to opportunistic networks, space communication is most of the time predictable. Thus, the routing algorithm can calculate an optimal path to all destinations and an optimal bandwidth utilization. A discovery mechanism is not required here. The challenge in this scenario is to deal with long propagation delays which can vary between seconds up to hours.

2.2.2 Military and Tactical Communications

Different to space links, where are just a few but predictable opportunities exist to communicate with other stations, military scenarios are dominated by being mostly connected with many unpredictable interruptions due to difficult terrain, weather, interference, node movements, destruction of infrastructure, or jamming. Communication between nodes is realized used wired or wireless technology with short propagation delays.

The challenge is to provide a reliable transmission of data over a network that is often spontaneously broken without a continuous end-to-end connection. Due to the frequent topology changes, the network must be self-organizing and the routing can barely count on scheduled contacts.

2.2.3 Disaster and Emergency Systems

Disaster and emergency scenarios are similar to military and tactical communications. Both have to deal with unpredictable topology and must organize themselves in order to operate. Disaster and emergency systems are deployed typically using drop-boxes to replace a broken infrastructure. Fire fighters and other helpers in the disaster zone act as relay and data-mule. They are equipped with devices which communicate with each other and the drop-boxes.

The challenge is to provide a reliable transmission of data. Additionally, such a network must operate even if there are losses of both data and even nodes. Thus, a replicating routing approach, capable to work with frequent unpredictable topology changes, should be used.

2.2.4 Internet Deployment

In developing countries as well as in rural areas where classical connectivity to the Internet is not deployed because it is too expensive or unfeasible, a DTN technology offers an affordable way to realize access to the Internet. The DakNet and the N4C project (Section 2.1.2 and 2.1.3) are dedicated for such a challenge. Both utilize existing transportation systems and wireless links to shuttle data from connected areas to rural ones and back. While nodes on-board of those vehicles are not energy-constrained, the power management of stationary nodes acting as relay or post-box is quite important. [DRW11]

Links are likely predictable but less accurate as in space scenarios. Challenges are more financial. Hardware must be affordable to the citizens and rugged to survive even adverse conditions. Moreover, the usability of applications is very important. Approaches are required to link the continuously connected Internet to endpoints in the intermediately connected DTN. An obvious application for those scenarios is an E-Mail service which can be integrated seamlessly because existing protocols are already designed to deal with long message delays.

2.2.5 Wireless Sensor-networks

The most power-challenged scenarios for DTNs are those classified as Wireless Sensor Network (WSN). Usually they are battery-powered and deployed, except of power harvesting strategies like solar power or wind generators, without any external power source. For that reason, nodes in those networks are often equipped with low-powered CPUs and a low memory capacity. The measured data of each node is typically forwarded to a sink which processes these and generates reports.

In static WSNs, the nodes are permanently within contact-range and since there is no change in the topology, an optimal route can be calculated. Instead of solving a routing issue, the store-and-forward approach offers a high potential to save energy by suspending nodes and delay data transfers to scheduled slots when a pair of nodes is awake. In dynamic environments, a self-organizing DTN is easy to deploy and can even proceed to operate if nodes disappear or fail completely. The challenges here are similar as in disaster and emergency systems, thus, a predicting and replicating routing approach is necessary.

The EMMA project and the ZebraNet, presented in Section 2.1.5 and 2.1.1, can be considered as dynamic WSNs. Both collect data from moving nodes and exploit the store-carry-and-forward approach to bring the data to a sink, even if the sink itself is moving.

2.2.6 Mobile Internet

As considered in the introduction, beside all the scenarios mentioned above, a DTN is capable in improving the user-experience while using Internet on the move. Since the *Delay-Tolerant Networking Architecture* is designed as overlay-network, it can utilize any kind of connectivity to another node and switch seamlessly between different wireless technologies.

Even in areas where a good cellular coverage is expected, a disconnection happens quite often. The up-link to the Internet is usually lost by walking into an underground garage or the basement of a larger building. There exist even upper apartments in large cities where the cellular signal is too weak for phone calling caused by an unfavorable architecture.

The opportunity for DTN technology is not only to realize a permanent connection using peer-to-peer multi-hop forwarding. Moreover, the experience on the devices can be improved significantly by not annoying the user while there is no connection at the moment. Instead of aborting a transaction if connectivity is gone, applications can send their request in anticipation that it will arrive the destination as soon as possible.

2.3 Bundle Protocol

Nowadays, Internet protocols have been deployed all over the world and are generally accepted to be the standard for networking. Those protocols are designed to work on top of almost stable links and therefore based on several assumptions.

- An end-to-end path between a source and a destination node exists for the duration of a communication session.
- Retransmissions based on timely and stable feedback from data receivers is an effective means for repairing errors within reliable communication.
- End-to-end loss is relatively small.
- All routers and end stations support the TCP/IP protocols.
- Applications do not need to worry about communication performance.

- Endpoint-based security mechanisms are sufficient for meeting most security concerns.
- Packet switching is the most appropriate abstraction for interoperability and performance.
- Selecting a single route between sender and receiver is sufficient for achieving acceptable communication performance.

These fundamental assumptions prevent the deployment of Internet-technology in challenging environments with disruption and large transmission delays as presented in the Sections 2.1 and 2.2. Therefore, the BP [SBo7] was designed to realize networks with intermittently connected nodes by utilizing a store-and-forward approach. Although IP networks are also based on the same operations, the »storing« will not persist for a long time. In contrast, the BP does not expect that network links are always available or reliable. Bundles may be stored for a long time until they are finally forwarded to another node.

In this section, we will summarize the major topics of the *Bundle Protocol Specification* and *Delay-Tolerant Networking Architecture*. Rather than repeating the whole documents, we will focus on design aspects and technical details relevant for the upcoming considerations in Chapter 3 and 4.

2.3.1 Architecture

The BP is based on the *Delay-Tolerant Networking Architecture* [CBH⁺07] which relaxes the assumptions made for the design of Internet protocols with the following design principles.

- Instead of using streams or limited-sized packets, messages of a variable length are used.
- A very flexible naming syntax is used to support a wide range of naming and addressing conventions.
- By providing storage within the network, store-and-forward operations can be performed over multiple paths and potentially long timescales.
- End-to-end reliability is optional and in most of the cases a hop-by-hop mechanisms should be sufficient to guarantee reliable transmissions.
- Security mechanisms should discard unsolicited traffic as quickly as possible and therefore protect the infrastructure from unauthorized use.

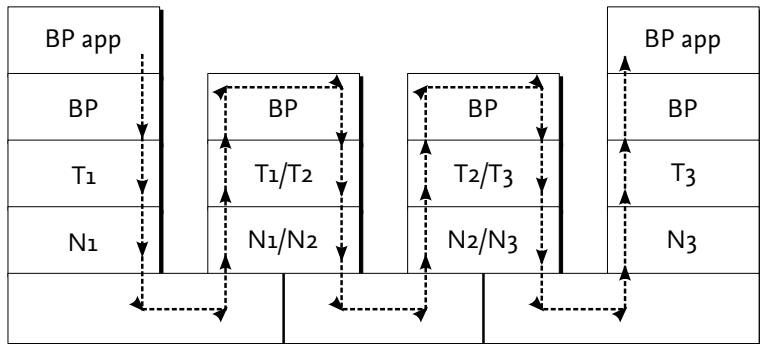


Figure 2.1: Bundle Protocol on top of other underlay networks

- Coarse-grained classes of service, delivery options, and a way to express the useful life-time of data is necessary to better deliver data and serving the needs of applications.

While the *Delay-Tolerant Networking Architecture* describes the basic idea and the abstract mechanisms of the store-and-forward approach, the *Bundle Protocol Specification* describes the format of Protocol Data Units (PDUs) passed between entities participating in BP communications. These entities are called »bundle nodes« (or simply »nodes«) and can send and/or receive PDUs as specified by the BP. They may be realized as dedicated processes or threads on a general purpose computer. Alternatively, they may be implemented as objects on object-oriented operating systems or even as special purpose hardware devices. In any case a node consists of a Bundle Protocol Agent (BPA), an Application Agent (AA), and a set of Convergence Layer Adapters (CLAs). A BPA is a component of a bundle node that offers BP services and executes procedures of the BP. The AA utilizes the BP services to effect communication for some purpose. Finally, the CLAs provide the ability to transfer PDUs between several bundle nodes by utilizing underlying networking protocols.

To cope with intermittent connectivity and take advantage of scheduled, predicted, and opportunistic connectivity (in addition to continuous connectivity), the BP adds custody-based retransmission for improved reliability and an approach for late binding of overlay-network endpoint-identifiers to constituent network addresses. Not part of the specification are operations in the convergence-layers, bundle routing algorithms, and mechanisms for populating the routing or forwarding information of bundle nodes.

The BP sits at the application layer of some number of underlay networks, forming a store-and-forward overlay-network. Native Internet protocols as well as any other kind of networking technology can be utilized to realize communication between bundle nodes. The interface between the BP and a specific underlay networking protocol suite is termed a »convergence layer adapter«. Figure 2.1 depicts the corresponding architecture and shows how an Application Data Unit (ADU) is being forwarded through three different transport- (T_1, T_2, T_3) and network-layers (N_1, N_2, N_3) until it reaches the destination.

2.3.2 Addressing

A bundle endpoint (or simply »endpoint«) is a set of zero or more bundle nodes that are addressable for BP purposes. An endpoint that never contains more than a single node is termed a »singleton« endpoint. Every bundle node must be a member of at least one singleton endpoint. Further, it is possible that a single node is a member of multiple bundle endpoints.

To attain a high degree of flexibility, endpoints are identified by text strings termed Endpoint Identifiers (EIDs). Their syntax is based on the definition of Uniform Resource Identifiers (URIs) [BLFM05] and therefore each of them is a compound of a leading scheme name and a scheme-specific part (SSP). For practical reasons, both are limited to a maximum of 1023 bytes. All schemes following the URI syntax are coordinated by the Internet Assigned Numbers Authority (IANA) [Kly14]. The scheme name »dtn« has been registered for the BP. URIs provide significant flexibility in structuring EIDs. They can address single nodes, interest groups, or even contain database-like queries. The standard address scheme chosen for the bundle protocol consists of a host and an application. E.g. the EID »dtn://foo/bar« addresses the application »/bar« on the host »/foo«.

For a more efficient encoding of EIDs within the primary bundle block, all scheme names and SSPs are encoded in a dictionary. The primary bundle block and potentially all extension blocks contain references to strings contained in there. Such an encoding is especially useful if multiple references point to the same EID. For example, if the source and the report-to address is equal, then the strings for these EIDs are only stored once within the dictionary and the references for the source and the report-to EID point the same numeric values. Since EIDs for report-to, custodian, and those referenced within extension blocks can be altered on the path of the bundle, the dictionary in the primary bundle block must be revised on every change of any of them. Further, all EID-references in the primary bundle block and in all extension blocks must be adjusted according to the changed values in the dictionary.

An EID may address a single application or a group of nodes. When referring to a group of size greater than one, the delivery semantics may be of either anycast or multicast variety. For anycast, the bundle is to deliver to a single node of a group of potentially many nodes. Multicast specifies delivery to all members of a group. In classical network structures with low-delay, nodes are considered to be part of a group if they have expressed their interest to join. In a DTN, the nodes may wish to receive data sent to a group during an interval of time earlier than when they are actually able to receive it. How to realize multicast or anycast is left as research issue.

In conjunction with the string representation of EIDs, the *Bundle Protocol Specification* introduces an approach called »Late Binding«. Interpreting a SSP of an EID for the purpose of carrying an associated message towards a destination is called »binding«. In IP-based networks, this would be the resolution of an URI to a corresponding host which is done before the actual traffic is sent towards the destination. In contrast to this, the string-based EID is not resolved once at the sender. Instead, the SSP is reinterpreted at every hop of the path. Such an approach is necessary, because it is possible that the validity time of the binding is shorter than the time it takes to deliver the bundle. Moreover, this way the amount of administrative information that must be propagated through the network is reduced.

2.3.3 Data-structures

The BP encapsulates ADUs in messages of arbitrary size and aims to deliver these as complete units to applications. In the context of the BP these PDUs are called »bundles«. Although it is possible to spread data over multiple bundles and stitch them together on the receiver side, the idea behind such messages and their potentially large amount of payload is that all data which forms a logical unit is encapsulated into a single bundle. For example, if a web-page is transported within a bundle, it would make sense that all the images and supplementary files are also packaged in the same unit.

Bundles consist of two or more blocks, while they have always exact one »bundle payload block« and a single »primary bundle block« which is always the first block of the chain. Bundles can be of almost arbitrary length, because they can carry an infinite number of blocks. Moreover, the bundle payload block can contain up to $2^{64} - 1$ bytes of »bundle payload« (or simply »payload«). If a bundle is generated by an application, the payload is called an ADU. During the processing of the bundle, the payload may be altered by various extensions in order to applying encryption or compression to the payload.

Bundle blocks different to the bundle payload block or the primary bundle block are called »extension blocks« and hold information typically found in the headers or payload portion of PDUs in other protocol architectures. The term »block« is used instead of »header« because blocks may not appear at the beginning of a bundle. For example, signatures are added at the end of the bundle or at least behind the bundle payload block.

A network may contain multiple instances of the same bundle distributed over multiple BPAs. Further, the representation may be different whether the bundle is held in memory or in a persistent storage. In order to transfer a bundle from one node to another, the *Bundle Protocol Specification* defines the exact representation to guarantee interoperability.

SDNV

In order to avoid wasted bandwidth but keep the bundle structure flexible to address requirements not yet identified, Self-Delimiting Numeric Values (SDNVs) are utilized to encode almost all numeric values. This design decision allows the protocol to scale across a wide range of network scales and payload sizes.

In general, a SDNV is an unsigned numeric value encoded in N bytes while the Most Significant Bit (MSB) of the last byte is set to zero; every other byte have the MSB set to 1. This way it is possible to encode 7 bits in each byte. Values up to 127 can be encoded in a single byte. Larger values always require at least two bytes. In any case, the overhead is always 1 : 7. Although the encoding of SDNVs allows infinite values, for practical reasons the implementations are allowed to consider values larger than $2^{64} - 1$ as invalid.

Primary Bundle Block

Each bundle starts with its primary bundle block. That block contains information which is used to make scheduling and path selection decisions. According to the *Delay-Tolerant Networking Architecture* each PDU contains an originating timestamp, a useful life indicator, a class of service designator, and a length.

Figure 2.2 show the complete structure of a bundle's primary bundle block. Each encoded PDU starts with a 1-byte version field which is fixed to the value `0x06`. The following SDNV contains the bundle processing control flags. They indicate options necessary to decode the bundle correctly as well as properties used to make scheduling or path selection decisions.

The bits 0-6 are used for general indications. If the bit 0 is set, then the bundle represents a fragment and contains a subset of the payload of a bundle. Further, the primary bundle block of a fragment contains the optional fields »fragment offset« and »total application data unit length«. Bit 1 marks the bundle as admin-

Version	Bundle processing ctrl flags (*)
	Block length (*)
Destination scheme offset (*)	Destination SSP offset (*)
Source scheme offset (*)	Source SSP offset (*)
Report-to scheme offset (*)	Report-to SSP offset (*)
Custodian scheme offset (*)	Custodian SSP offset (*)
	Creation Time-stamp time (*)
	Creation Time-stamp sequence number (*)
	Life-time (*)
	Dictionary length (*)
	Dictionary byte array (variable)
	Optional: Fragment offset (*)
	Optional: Total application data unit length (*)

(*) marked fields are encoded as SDNV

Figure 2.2: Primary Bundle Block Format

istrative record. In this case, the payload within the bundle payload block contains an encoded Bundle Status Report (BSR) or a custody signal as described in Section 2.3.5 below. To indicate that a bundle is not allowed to be fragmented, the bit 2 can be used to deny such an attempt. If a bundle should be handled by custody operations as explained in Section 2.3.6 below, the bit 3 can be set. Bit 4 is used to classify the destination EID as singleton (see Section 2.3.2 for details). To request an acknowledgement of the application, the bit 5 is set. The remaining bit is reserved for future use.

The bits 7-13 specify the class of service for the bundle and therefore are important for scheduling and path selection decisions. Each bundle is marked with a specific priority which differentiate traffic based upon an application's design to affect the delivery urgency for ADUs. At the moment, the DTN architecture offers three relative categories for priorities. Bundles in a class of »bulk« are shipped on a »least effort« basis. No bundles of this class will be shipped until all bundles of the other classes bound for the same destination and originating from the same source have been shipped. To classify bundles as more important, the class »normal« can be used. These bundles are shipped prior to bulk bundles, but after bundles with the priority class »expedited«. These are shipped prior to bundles of all other classes. Priorities are represented using the bits 7 and 8 of the bundle

processing control flags. The remaining class of service bits (9-13) are reserved for future use. If the bits 7 and 8 are unset, then the bundle is in class bulk. The 7th bit indicates a normal priority and the 8th bit marks the bundle as expedited. The state where both bits are set is reserved for future use.

Important to highlight is that a priority class of a bundle is only required to relate to other bundles from the same source. Thus, a bundle marked as expedited from one source may not be delivered faster than a bulk bundle of another source. The bundle will be handled preferentially only if the source is equal. However, it is part of the specific forwarding/scheduling policies to apply priorities to bundles with different sources or not.

The bits 14-20 of the bundle processing control flags are used to request reporting capabilities. Status reports can be requested in case of reception of a bundle (bit 14), acceptance of custody (bit 15), a bundle has been forwarded (bit 16), a bundle has been delivered (bit 17), and in case a bundle has been deleted (bit 18). The bits 19 and 20 are left reserved for future use.

The next field in the bundle primary block is the »Block length«. It indicates the amount bytes of the remaining block. That information can be used to seek to the next block of the bundle without parsing all other fields of this block.

The EID-references for destination, source, report-to, and the custodian follow the block length. Each of them consists of a scheme offset and a SSP offset. Both offsets point to an entry within the »Dictionary« which is also part of the primary bundle block and contains a set of concatenated strings. If an EID is not set, the URI »dtn:none« is encoded instead. The dictionary reduces the overhead introduced by string-based EIDs by allowing a scheme or a SSP be referenced multiple times. The dictionary itself is encoded as »Dictionary length« plus all strings separated by the string-termination symbol within the »Dictionary byte array«.

For unique identification the primary bundle block contains a »Creation Time-stamp time« expressed in seconds since the start of the year 2000 and a »Creation Time-stamp sequence number«. In combination with the source EID, this information forms a globally unique identifier for the bundle. In order to differentiate fragments, the »Fragment offset« as well as the payload length is also relevant for identification.

The last part to mention here is the »Life-time«. This field contains a number of seconds which indicate how long a bundle is valid. In order to expire a bundle correctly, the time-stamp of the block is used in conjunction with the life-time to determine when the bundle will expire.

Block type	Block processing ctrl flags (*)
	Block length (*)
	Block body data (variable)

(*) marked fields are encoded as SDNV

Figure 2.3: Block layout without EID-reference list

Block type	Block processing ctrl flags (*)
	EID-reference Count (*)
Ref. Scheme 1 (*)	Ref. SSP 1 (*)
Ref. Scheme 2 (*)	Ref. SSP 2 (*)
Block length (*)	
Block body data (variable)	

(*) marked fields are encoded as SDNV

Figure 2.4: Block layout with two EID-references

Extension Blocks

Blocks proceeding the bundle primary block are either bundle payload blocks or extension blocks. While there is exactly one bundle payload block in each bundle, extension blocks are optional and may be inserted or appended in an arbitrary order prior to or after the bundle payload block.

The general structure of blocks proceeding the primary bundle block is shown in Figure 2.3. Each block starts with a 1-byte »Block type« which is `0x01` for a bundle payload block and larger for extension blocks. Assignments of type numbers are coordinated by the IANA [ian14]. The second field in this block contains the block processing control flags. Beside indication of the last block of the bundle (bit 3), the flags also specify whether a block should be replicated in every fragment (bit 0). Further, the flags can specify how to proceed if the block is not supported and therefore can not be processed by a BPA. Available options are to generate a BSR (bit 1), discard the block (bit 4), or delete the whole bundle (bit 2). In case the block has been forwarded without being processed, the bit 5 indicates this to subsequent BPAs. Finally, the bit 6 indicates if an EID-reference list is present in the block header.

An EID-reference list can be used to add extension-specific EIDs-reference to the bundle. E.g. the *Bundle Security Protocol Specification* [SFWL11] defines a security-source and a security-destination which is stored within such an EID-reference list. Figure 2.4 shows a block which contains two EID-references. In addition to the basic structure shown in Figure 2.3, an »EID-reference Count« and two pairs of scheme and SSP offset are inserted right after the block processing flags. The »Block length« contains the number of bytes stored within the »Block body data« which contains extension-specific payload. If that block is of type 1 and therefore a bundle payload block, the block body data would contain the application-specific ADU.

2.3.4 Fragmentation

Since the bundle payload can be of almost arbitrary size, it may be reasonable to split-up a bundle into multiple constituent bundles called »fragments«. Fragmentation should improve the efficiency of bundle transfers by avoiding retransmission of partially-forwarded bundles in case of an interruption. Basically, there exist two approaches to perform a fragmentation. During a pro-active fragmentation, a block of application data is divided into multiple smaller blocks while each block is placed into an independent bundle. In this case, the final destination is responsible for reassembling the blocks to gain the original ADU. This approach can be applied if the contact volumes are known (or predicted) in advance. In contrast, a reactive fragmentation do not divide bundles before they are transmitted. Instead, the sender and the receiver generate fragments cooperatively in case of an interrupted transmission. On the receiver side, the bundle-layer transforms the partially received bundle into a fragment. The sender utilizes knowledge of the convergence-layer protocols to determine how much of the bundle has been received on the other side. On future contacts, the sender can transmit the remaining data as a fragment, even to different next-hops. During this type of fragmentation it may happen that the payload of the fragments overlaps.

The primary bundle block of each fragment is equal to the original bundle except of the additional »Fragment offset«, the »Total application data unit length«, and the set fragment bit within the bundle processing control flags. Further, the extension blocks that precede the bundle payload block must be replicated in the bundle fragment with the lowest offset. Other blocks are replicated in the bundle fragment with the highest offset. Each extension block is replicated once in a single fragment unless the option »Block must be replicated in every fragment« is set. In any case, the order of all blocks must stay the same as in the original bundle.

Fragmentation as specified by the *Bundle Protocol Specification* is partially optional. Not all bundle nodes must be capable in generating fragments, but each of them should be able to reassemble fragments in order to deliver the content to the applications. Moreover, although fragments are typically reassembled at destination endpoints, it is possible that an ADU is reassembled on the route towards the destination by some other node.

2.3.5 Administrative Records

The *Delay-Tolerant Networking Architecture* defines two types administrative records which are used to report status information or error conditions related to the bundle-layer using BSRs and custody signals. These are similar to Internet Control Message Protocol (ICMP) [Pos81a] messages in IP. But instead of always sending them to the source of the message, those records are sent to the report-to or the custodian endpoint which may address a different singleton endpoint.

A BSR may be generated on different conditions and is addressed to the report-to endpoint of a bundle. An application can request a BSR if the ADU is *delivered* to its intended recipient(s) or if the ADU has been acknowledged by the application. For network diagnostic purposes it is possible to request BSRs if a bundle has been received, forwarded, deleted, or a node has accepted custody for it. The use of diagnostic options should be restricted, because they can cause network congestion.

Custody signals are generated in one of two cases: on success or failure of a custody transfer. A custody transfer was successful if a subsequent node on the path agrees to be the new custodian. In that case, the node will send a success custody signal to the formerly custodial node. Since that node no longer have to take care of the bundles delivery, it is allowed to free resources (e.g. retransmission timers) that were necessary to take care of that bundle. When a failure custody signal should be sent is only defined for the cases if the destination is a singleton endpoint and the node's custody transfer timer expires or if a failure custody signal for that bundle has been received. In any case the signal is directed to the formerly custodian that was specified in the primary bundle block.

No matter if the administrative record is a BSR or a custody signal, it must always refer to a specific bundle. This is accomplished by inserting the unique identifier of bundles. This consists of the creation time-stamp time, creation time-stamp sequence number, and the source endpoint. If the referred bundle is a fragment, the fragment offset as well as the fragment length must also be copied to the administrative record. Whether the referenced bundle was a fragment, is indicated by the first of the four administrative record flags which follow the 4-bit record type code at the beginning of each bundle payload block. The remaining 3

Status Flags	Reason code	Fragment offset, if present (*)
		Fragment length, if present (*)
		Time of receipt of bundle X, if present (a DTN time)
		Time of custody acceptance of bundle X, if present (a DTN time)
		Time of forwarding of bundle X, if present (a DTN time)
		Time of delivery of bundle X, if present (a DTN time)
		Time of deletion of bundle X, if present (a DTN time)
		Copy of bundle X's Creation Time-stamp time (*)
		Copy of bundle X's Creation Time-stamp sequence number (*)
Length of X's source endpoint (*)		Source EID of bundle X (variable)

(*) marked fields are encoded as SDNV

Figure 2.5: Bundle Status Report Format

bits of the administrative record flags are reserved for future use. The administrative type code is used to differentiate a BSR (`0x01`) and a custody signal (`0x02`).

The specific administrative record content in its type-specific format is placed after the administrative record type and flags. Figure 2.5 shows the format for BSRs. These start with their status flags (1-byte) indicating which type of status report is included. This bit-set is important in order to process the record content correctly, since it indicates which of the optional fields are present. Possible types and their bit combinations are shown in Table 2.1. The »Reason code« (1-byte) specifies the reason for the BSR in more detail. At the moment, there are 9 different reasons defined as listed in Table 2.2.

Value	Meaning
00000001	Reporting node received bundle.
00000010	Reporting node accepted custody of bundle.
00000100	Reporting node forwarded the bundle.
00001000	Reporting node delivered the bundle.
00010000	Reporting node deleted the bundle.
00100000	Unused.
01000000	Unused.
10000000	Unused.

Table 2.1: Bundle Status Report Flags

Value	Meaning
0x00	No additional information.
0x01	Life-time expired.
0x02	Forwarded over unidirectional link.
0x03	Transmission canceled.
0x04	Depleted storage.
0x05	Destination EID unintelligible.
0x06	No known route to destination from here.
0x07	No timely contact with next node on route.
0x08	Block unintelligible.
(other)	Reserved for future use.

Table 2.2: Bundle Status Report Reason Codes

Status	Fragment offset, if present (*)
	Fragment length, if present (*)
	Time of signal (a DTN time)
	Copy of bundle X's Creation Time-stamp time (*)
	Copy of bundle X's Creation Time-stamp sequence number (*)
Length of X's source endpoint (*)	Source EID of bundle X (variable)

(*) marked fields are encoded as SDNV

Figure 2.6: Custody Signal Format

The »Fragment offset« and the »Fragment length« are only present if the BSR references to a bundle fragment. They complement the »Creation Time-stamp time«, »Creation Time-stamp sequence number«, and the »Source EID« used to identify the corresponding bundle. The »Time of ...« fields in the BSR contain a *DTN time* which expresses the moment when that signal was triggered. These are only present if the corresponding bit in the flags was set. DTN time values are a compound of two SDNVs while the first contains the number of seconds since the start of the year 2000 and the second the number of nanoseconds since the start of the indicated second.

The format of custody signals starts with a single byte as status field. The first status bit indicates custody success or failure and the remaining 7 bits are used to add a reason to this indication. Possible values are listed in Table 2.3. As in the format of BSRs, the custody signal format contains the »Creation Time-stamp

time«, »Creation Time-stamp sequence number«, the »Source EID«, and in case of a fragment, the »Fragment offset« and the »Fragment length« of the bundle which corresponds to that signal. Finally, each custody signal has a »Time of signal« which indicates the exact time when the signal was triggered.

Value	Meaning
0x00	No additional information.
0x01	Reserved for future use.
0x02	Reserved for future use.
0x03	Redundant reception.
0x04	Depleted storage.
0x05	Destination EID unintelligible.
0x06	No known route to destination from here.
0x07	No timely contact with next node on route.
0x08	Block unintelligible.
(other)	Reserved for future use.

Table 2.3: Bundle Custody Signal Reason Codes

To avoid traffic amplifications issued by administrative records, the use of processing control flags is restricted. Thus, if an administrative record is transported using a bundle, the custody transfer requested flag and all status report flags of the bundle processing control flags must be zero. Further, the »Transmit status report if block can't be processed« of each extension block must be set to zero.

2.3.6 Custody Transfer

The basic service of the bundle-layer is unacknowledged, prioritized unicast message delivery. In order to add some portion of reliability, the application itself can implement their own mechanism using end-to-end acknowledgements as in IP-based networks. But long delays between sender and receiver makes such an approach ineffective, because it would require long lived timers and the retransmission would take a long time. Further, that approach would require a lot of resources on every node along the bundle's path.

An approach, that would better fit the long transmission delays in a DTN, is called custody transfer and delegates the retransmission responsibility to nodes along the path instead of relying on a retransmission performed by the sender only. As soon as a bundle node on the path has accepted custody for a given bundle, the sender can recover retransmission-related resources. Ideally, this may happen relatively soon after sending a bundle. This way, the responsibility to deliver the

bundle moves along the path towards the destination. Nodes which accept the delivery responsibility for a given bundle are called custodians.

The bundle processing flags indicate if a bundle requires custody or not. The corresponding flag can be set by applications which also can specify if it is required that the source node must accept custody for the bundle. On each reception of a bundle with a requirement on custody, a node should determine if the available resources permit the acceptance of custody for that bundle. If this is the case, the node will store the bundle (e.g. in persistent storage) and send a bundle containing a success custody signal to the formerly custodian which is noted in the primary bundle block as custodian endpoint. The formerly custodial node can now release the retransmission timer and all other resources associated with that bundle. If a node decides to not accept custody, it simply forwards the bundle along the path. Thus, a bundle with enabled custody transfer can be forwarded multiple hops until a node accepts custody for it. Therefore, a custody transfer is not a true hop-by-hop mechanism. Nodes can choose to not act as custodian due to running on low power or being congested. Sending a custody signal reporting a failure is only done if the node's custody transfer timer expires or if a failure custody signal for that bundle has been received.

To mention here is that this approach is in an experimental stage and is only defined for unicast delivery of messages. If the bundle is the subject of multicast or anycast delivery, the exact meaning and the required procedures are not defined. Moreover, decisions involving custody transfer can effect the chosen path. Because it might be useful to forward a bundle as quickly as possible to a custodian.

2.3.7 Application Agent

In addition to the BPA and associated CLAs, the *Delay-Tolerant Networking Architecture* defines the AA as further component of a bundle node. An AA utilizes the BP services to effect communication for some purpose and consists of two elements: the administrative element and an application-specific element. The latter one requests the transmission of ADUs, accepts them for delivery, or processes them. The administrative element constructs and requests transmission of administrative records (BSRs and custody signals). Further, it processes any custody signal that the node receives and accepts delivery of them. The AA may serve BP services to multiple applications at the same time. In case a node acts as router only, the AA may have no application-specific element at all.

An application which want to transmit or receive ADUs have to indicate which endpoints are of interest. For that purpose, a »Registration« is usually created and defines a given node's membership in a given endpoint. A node may have any

number of registrations associated at the same time. Each registration must be in active or passive state. In active state, a bundle can be directly delivered to an application. In passive state, the associated application is suspended and can not process bundles. Thus, a »delivery failure action« decides whether the bundle is stored for deferred delivery or being abandoned. An application can switch the state of a registration at any time or even poll bundles of a passive registration without switching it to active.

In addition to the BP design principles presented before, the application-layer protocols and their implementations should also follow some guide-lines. First, the number of round-trips should be minimized. Due to the potential large transmission delays, a handshake with several round-trips are not reasonable. Further, each application should be capable to restore pending network transactions after a restart due to a failure. Finally, applications should augment bundles with useful life-time values and declare the relative importance of data to be delivered.

2.3.8 Security

The BP has been designed with the notion that it will be run over networks with scarce resources. These might be networks with limited bandwidth, limited connectivity, constrained storage in relay nodes, etc. Thus, these scarce resources must be protected against un-authorized usage.

Authentication of users, traffic, and applications is required to prevent unauthorized access. Otherwise, the network is threatened by unauthorized users which may flood the network with unsolicited traffic and disturb other services. Even if the users are authenticated to inject bundles, these may not be authorized to utilize certain network links or they might be somehow rate-limited. In case of the network is managed, the control over the network must also be protected from unauthorized applications. Further, the network should discard damaged or improperly modified bundles in transit and be capable in detecting and de-authorizing compromised entities.

Many existing protocols for authentication and access control are designed for low-delay networks. In detail, they rely on frequently updating an access control list, revoking credentials, or have a centralized server which is frequently accessed in order to complete an authentication or authorization transaction. These approaches do not perform well in DTNs, because the potentially long delays would dramatically slow down the communication. Therefore, a mechanism is proposed which supports the hop-by-hop as well as the end-to-end schemes for authentication and integrity checking. The different approaches are capable in checking

access permissions of forwarding nodes in a hop-by-hop manner or authenticate users using the end-to-end principle.

The *Bundle Protocol Specification* briefly introduces a concept which uses three extension blocks to realize confidentiality, verification of integrity, and authentication of bundles. But instead going further into details, the exact specification of those blocks and corresponding security algorithms is part of the *Bundle Security Protocol Specification* [SFWL11].

2.3.9 Dependency on Synchronized Clocks

The DTN architecture and therefore also the BP depend on time-synchronization among nodes for various mechanisms. Time information is used for bundle and fragment identification, it is referenced in BSRs and custody signals, used for expiration of bundles and registrations, and is necessary to perform routing with scheduled or predicted contacts.

Each bundle is uniquely identified by their source, creation time-stamp, and creation time-stamp sequence number. To differentiate fragments, the fragment offset and the payload size is also relevant. Identifications are used to reference bundles in BSRs, custody transfer messages, and during the reassembly of bundle fragments. They are also useful to detect loops and to avoid redundant transmissions. Synchronized time is also necessary to interpret BSRs and custody signals. These contain DTN time values which point to an exact time when the report or signal has been generated.

The expiration of bundles or registrations is typically realized with time-stamps. Similar to bundles, registrations last only for a finite period. Multicast registrations may also specify a time range or »interest interval« to select bundles they would like to receive. Traffic generated during that period is delivered to the registration unless it is already expired. Since it is expected within a DTN that a node is inactive or may shutdown itself for some period, a node can not always monitor bundles or registrations for expiration using relative time.

Finally, routing with scheduled or predicted contacts heavily depends on synchronized clocks. Exchanged schedules or predictions can not be interpreted correctly if it is not clear when in time an encounter will happen.

2.4 Implementations

The software implementation presented in later chapters of this work, is not the first and not the last of its kind. In this chapter, we will present the major colleagues and take a closer look on their features. Due to missing documentation of

many of them, we have done code inspection to figure out the supported extensions.

2.4.1 DTN2 Reference Implementation

DTN2 [DBF⁺04] is the reference implementation of the *Delay-Tolerant Networking Architecture* [CBH⁺07] and the *Bundle Protocol Specification* [SB07]. Although this software is the result of research and targets mainly research-specific needs, the developers of DTN2 target to provide a flexible software framework for experimentation, extension, and real-world deployment.

The software architecture is mainly object-oriented and mixes C and C++ programming patterns. For a better platform abstraction, the underlying OASYS library is developed in conjunction with the DTN2 package. OASYS encapsulates threading, memory management, serialization, XML processing, storage abstraction, and networking capabilities for AX.25, Bluetooth, TCP, UDP, and SMTP. Furthermore, it includes some miscellaneous utilities. These abstractions should make porting the software to new platforms more easy. However, except of some defines surrounding Win32-specific code, there exist no native port to any non-POSIX platform. Instead it is suggested to compile DTN2 for Windows using the Cygwin project which can set-up an POSIX-compatible environment on Windows.

While the OASYS library contains generic classes for abstraction of platform-specific code, the DTN2 package contains the actual implementation of the *Delay-Tolerant Networking Architecture* and the *Bundle Protocol Specification*. According to the Functional Specification [Dav10] documented during *Networking for Communications Challenged Communities* (N4C) project, the central component is the *Bundle Daemon Core*. Further, there are components called *API Server*, *Fragmentation Manager*, *Router*, *Persistent Storage*, and *Contact Manager*. The latter one manages the various convergence-layer modules and discovery routines. Over the past years, many experimental features has been integrated into DTN2 during research projects and some of them were stable enough to stay there. For that reason, DTN2 can show more features than other existing implementations. There are convergence-layers for the protocols AX.25, Bluetooth, Ethernet, Licklider Transmission Protocol (LTP), Nack-Oriented Reliable Multicast (NORM), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). Additionally, a Extensible Markup Language (XML)-based Application Programming Interface (API) allows an external convergence-layer module to attach directly to the BPA. Since DTN2 is the most used software for experimental evaluation of routing algorithms, there are plenty of them integrated. Beside the usual suspects

for static, flooding, and epidemic routing, the bundle node supports routing using the PRoPHET, Delay Tolerant Link State Routing (DTLSR), and Tetherless Computing Architecture (TCS) algorithm. Beside those, a XML-based API allows an external routing module to attach directly to the daemon. Further, there exist bundle storage implementations for storing bundles in memory, within a file-system, or using external libraries. To integrate those libraries there exist code for the Berkeley DB library or SQL-based databases supported by the ODBC interface (e.g. SQLite or MySQL). Like other replaceable components, it is possible to connect an external storage implementation using a XML-based API.

A bunch of extensions complete the feature set of the DTN2 software. Experimentally, the Bundle Security Protocol [SFWL11] and IP Neighbor Discovery Protocol [EAGB12] has been integrated. However, Apple's Bonjour protocol or the internal proprietary approach for discovery is still the recommended option. Additionally, the extensions *Bundle Protocol Query Extension Block* [FLKL10], *DTN Bundle Age Block for Expiration without UTC* [BFB10], *Delay-Tolerant Networking Meta-data Extension Block* [Sym11a], and *Delay-Tolerant Networking Previous-Hop Insertion Block* [Sym11b] are integrated.

The comprehensive capabilities of the software have been widely used in scientific research to evaluate new algorithms. For that reason, the software provides extensive logging for debugging and operational purposes. Some minor optimizations are included, but the authors also confess that more optimizations or even a redesign is necessary to achieve a satisfactory performance. Moreover, the stability and robustness is to complain about. The use of assertions in large parts of the source-code leads to unexpected failures and thus a termination of the software. In some cases it is even sufficient to send a corrupt data-frame to the daemon to provoke a termination. Furthermore, the software uses several libraries, such as Tcl, which restrict the portability to common platforms for embedded system. Further, most of the code is written in C although it is surrounded by C++ classes. Features as exception handling, Standard Template Library (STL) libraries, and the concept of streams are barely used. This is one reason why the software structure is not as clear as it could be. To conclude, the software is pretty nice for research proposes but the discussed issues restrict the use of DTN2 in productive environments.

2.4.2 Interplanetary Overlay Network

Interplanetary Overlay Network (ION) is an implementation of the *Delay-Tolerant Networking Architecture* [CBH⁺07] and the *Bundle Protocol Specification* [SBo7] with focus on space related scenarios. As mentioned in [Buro9], a software for this field of operation has to be very robust, reliable, and efficient. Because of the time and

effort it takes to perform hardening against radiation, the performance of systems dedicated for space operations is far behind of those used in terrestrial environments. Thus, it is even more important to process bundles efficiently. Further, the software must run on top of real-time capable systems usually used for space-missions and dynamic allocation of system memory is prohibited during runtime. As result, the behavior of the software will be more predictable and will not affect other component of the system.

ION is modularized and distributes its work-load among multiple processes which are connected by shared-memory inter-process communication. This separation allows each single process to be much less complex than a monolithic program which performs all tasks. Further, functionality encapsulated into processes can be attached or removed during run-time without affecting the remaining functionality of the system. By linking all processes using a shared-memory approach it is possible to achieve a performance as if all processes were part of a single one. This is especially advantageous for management programs or applications. They can inject or pull-out bundles with a negligible overhead. Section 6.2.2 contains an evaluation on the API performance of the major DTN implementations including ION. Also beneficial for a high efficiency is a Zero-copy principle which is applied to many data structures. That approach avoids copying of data if possible by working with counted references. Those references are even used to pass data between multiple networking layers.

Since ION is dedicated for space-missions, its feature-set focuses on extensions to support related scenarios. Links for communication in those scenarios are inherently wireless and usually asymmetric. The amount of control data towards the space-crafts are small compared to the telemetry data send to ground-station. Thus, the bandwidth of the up-link towards space-crafts is typically at 1 Kbps to 2 Kbps while the down-link varies between 256 Kbps and 6 Mbps. To utilize the low bandwidth more efficiently, ION implements Compressed Bundle Header Encoding (CBHE) [Bur11] which reduces the EIDs contained in the primary bundle block to SDNV [ED11] encoded numbers. Although, the stack is optimized for that kind of encoding, it also supports other string-based EID schemes. Further, ION routes bundles according to priorities defined in the Primary Bundle Block and supports multi-cast EIDs. Beside static routing entries, the implementation supports contact graph routing which is very common for space-missions. Additionally, it contains the extensions *DTN Bundle Age Block for Expiration without UTC* [BFB10], *Delay-Tolerant Networking Previous-Hop Insertion Block* [Sym11b], *Bundle Protocol Extended Class Of Service* [Bur14a], and an undocumented *Custody Transfer Enhancement Block*. Various convergence-layer modules adapt the protocols LTP, Datagram Congestion Control Protocol (DCCP), TCP, and UDP for communica-

tion between bundle nodes and there exist source-code for all extension blocks defined by the Bundle Security Protocol [SFWL11].

In terms of portability, the software has been designed to utilize POSIX-compatible methods as much as possible. In cases where that is not possible, wrapper methods encapsulate platform-specific code. This way, the software can be easily adapted to new platforms as long as they are POSIX-compliant. The supported and tested set of platforms are Linux™ (Red Hat®, Fedora™, and Ubuntu™, so far), Solaris® 9, OS/X®, VxWorks® 5.4, and RTEMS™, on both 32-bit and 64-bit processors.

To conclude, ION is well-designed and provides a robust and stable implementation of a bundle node. However, this software is optimized for Contact Graph Routing and LTP communication. The concept of planned contacts is deeply integrated into the software and there is no alternative dynamic routing approach. Moreover, there is no dynamic discovery approach like the IP Neighbor Discovery Protocol [EAGB12]. Obviously, such a mechanism would not make sense for the scenarios ION was designed for. Therefore, ION, as the name suggests, is specialized for interplanetary communications only and unsuitable for opportunistic DTN scenarios.

2.4.3 Postellation

A relatively young member in the family of DTN implementations is Postellation [Bla12]. The goal of the project is to use Internet connectivity to provide the simplest possible starting point for the end user. This is achieved by skipping the need for a configuration after installation and the automatic inter-connection through the Internet with a network of DTN nodes running Postellation. Moreover, the standard package includes a HTTP proxy which allows the retrieval of web-pages over the DTN connectivity. For that purpose, the software acts as local HTTP proxy. Once the browser on the system is configured correctly, all requests are sent through the DTN to a HTTP proxy within a dedicated cloud.

The binary software package is offered as free download and has been ported to major operating systems including Windows®, Linux™, and OS/X®. The software is written in standard C, which makes it a good candidate for embedded systems. Beside the implementation of the BP, the feature-set includes convergence-layers for UDP, TCP, and a proprietary protocol realizing Transport Layer Security (TLS) over TCP. Moreover, IP version 4 as well as version 6 is supported. The distribution includes a set of useful tools for sending and receiving bundles, testing network connectivity using the echo-reply-scheme, the HTTP proxy server, and a tool to retrieve RSS news over DTN.

The easy installation allows a user to set-up and use a DTN without any further knowledge. However, the source-code of the software is not publicly available and there is no way to extend the software by new features or routing algorithms. That makes the software unsuitable for research and prevent a deployment in other scenarios which requires extensions not included in the standard packages.

2.4.4 Bytewalla

Bytewalla [YAMA11] was the first DTN implementation officially targeting the Android™-platform. To not start from scratch, the authors decided to port the source-code of the DTN2 reference implementation to the Java environment. But carbon-copying of classes to Java is not as easy as it sounds, because C++ provide multiple-inheritance while this is not possible in Java. Further, the code base of DTN2 is quite large and it is often necessary to redirect calls made to C++ specific code to an equivalent implementation in Java.

Basically, the software is divided into a *DTNService*, which runs in background, and a front-end for configuration, monitoring, and managing called *DTNManager*. The implementation ships two sample application which are connected to the *DTNService*. *DTNSend* is used to send text messages and *DTNReceive* is used to retrieve them.

As DTN2, Bytewalla is structured as a modular system. There are modules called *Bundle Daemon*, *Contact Manager*, *TCP Convergence-Layer*, *Discovery*, *Persistent Storage*, *Registration*, *Bundle Router*, *Fragmentation Manager* and *APILib*. New functionalities can be easily added or replaced. An implementation of a new routing algorithm is integrated by extending the *BundleRouter*. Other modifications are not necessary. Generally, the system is event-driven. Events are raised in case a bundle has been transmitted or received, or a contact has been initiated. The discovery procedure is based on the proprietary UDP-based approach implemented in DTN2 and static routing based on tables is supported.

The goal of the project was to bring the feature-set of DTN2 to the Android™-platform. Despite the Bytewalla project had a good start, the authors never reached their goal and the software has still many limitations. As routing protocol only static routes are supported. Moreover, they must be configured prior to starting the software which restrict the usage in larger scenarios or even dynamic environments. Further, applications must run in the same shared process. This is fine for research, but is not secure and not even feasible for third party applications. Although the authors claim the software as being more portable than other implementations, in fact, Bytewalla needs Android™ underneath. Therefore,

Bytewalla inherits its portability from the Android™-platform and runs on every hardware that is capable in running Android™.

Today, the project has almost vanished from the Internet. Many references in papers point to invalid pages. There exist some sequels of that project, but the latest attempt [KZ11] ends in fall 2011.

2.4.5 Other

There exists even more minor implementations of the BP. The inventors of the Probabilistic Routing Protocol using History of Encounters and Transitivity (PRoPHET) approach implemented a software named *PLUTI* [GGV12]. *JDTN* [Moo14] is a Java-based implementation of the BP and the Licklider Transmission Protocol (LTP). It was developed with mobile platforms, such as Android™, in mind. The Service platform for social-aware mobile and pervasive computing (*SCAMPI*) [PKO⁺12] can utilize the BP to exchange messages between nodes. It targets Android™ and is therefore written in Java. Beside running on the Android™-platform, it is possible to run the router component on other systems capable in running Java. An image for the Raspberry Pi credit-card sized computer is available for download.

2.5 Time-synchronization

The issue of synchronizing clocks is a recurring topic for computer networks and especially challenging for DTNs. Thus, it is remarkable that the BP defines synchronized clocks as mandatory although this seems to be an unsolved problem to date. However, the BP is not the only protocol with a dependency on synchronized clocks.

The *Simple Network Management Protocol* (SNMP) [CFSD90] uses time-stamps to order events and specify their duration. It is not classified as critical, if two different systems are not synchronous to each other. The only requirement is that the time on a single system increases monotonically in order to arrange events locally. A common time base is only necessary to get events of different systems in relation to each other.

Much more critical is the dependency to synchronized clocks in a network with an attached *Kerberos Network Authentication Service* [KN93]. The underlying protocol uses time-stamps to detect and stop attacks, e.g. reply attack. For this purpose, each request contains a signed time-stamp and the respective receiver determines the age of a request using the local time. If the request is too old, it will be discarded. The maximum age of a message should be adjusted depending on the type and

precision of the used time-synchronization method. The protocol specification mentioned a loose synchronization as requirement.

Even the *Internet Protocol* [Pos81b], as designed originally, used an aspect of time. The Time-To-Live field defined the number of seconds that a packet is valid. However, it has been found as infeasible to synchronize all computers on the Internet and this field has been reinterpreted in implementations of that protocol to a more practical approach. Now, the value of that field get decremented on each hop. Thus, the Time-To-Live field is semantically equivalent to a maximum number of transmissions instead of a life-time.

All the protocols mentioned above have one thing in common. They have been designed for continuously connected networks and are not applicable to disrupted end-to-end connectivity as common in DTNs. In case of continuously connected environments, it is possible to realize a sufficient synchronization for particular applications. The *Network Time Protocol* [Mil85] is often used in static computer networks and most parts of the Internet. It provides a high accuracy of up to 233 ps, but also has a high complexity. For protocols such as SNMP or *Kerberos*, less precise methods such as the *Time Protocol* [PH83] may be used in order to obtain sufficiently synchronized clocks. Both methods are synchronized using a master, which provides the most accurate clock.

2.5.1 Synchronicity in Computer Networks

In order to adjust several computers to a common time, a shared clock or several clocks are required which measure the same reoccurring event to estimate a time interval. If several independent clocks measure different events, a variation in accuracy is expected. Therefore, these clocks would be never in sync with each other, even if one tries to set them to the same value. Moreover, Freris et al. [FKK10] shows that it is even impossible to synchronize two clocks. Therefore, we declare two clocks as synchronized if the difference between them is acceptable small depending on the use-case, which may be several nanoseconds or even minutes. Furthermore, it is necessary to distinguish whether two clocks should show the same time-stamp or if they should provide synchronized clock ticks.

If an accurate time is required in a stationary computer network, a common clock can determine the right time using a reproducible method and distribute it to all connected stations. The accuracy depends on the size of the network and the transmission delay between clock and receiver. Such a clock reference can not be used if the network is too large or not continuously connected. Instead, several references are required while each of them has its own clock. To ensure that each reference provides the same time to its connected stations, all of them have to be

synchronized. The most common protocol for synchronizing clocks in computer networks is the Network Time Protocol (NTP) [Mil85]. It is based on UDP and distributes the time in a fixed hierarchical structure. A server on top of the hierarchy uses a clock with a very high precision to adjust a reference clock (stratum 0). In deployments, such a server is usually connected to an atomic clock. The resulting time reference signal (stratum 1) is provided to other servers on the second level of hierarchy. Signals provided by servers lower in the hierarchy, are categorized with stratum $2 \dots n$, depending of its »distance« to stratum 0. All servers steadily compare and adjust their own clock with one or more servers above in the hierarchy. In mobile computing networks, such a fixed hierarchical structure is hard to apply.

2.5.2 Challenged Networks

So called »challenged networks« cover networking scenarios under difficult conditions. Most of them have to deal with a missing continuous end-to-end connectivity which is necessary for classical networking approaches. Originally devised for interplanetary communication where long transmissions and interruptions are very common, the DTN approach solves many communications challenged scenarios with unstable connections, sporadically available links, and long delays. That includes connections between vehicles, sensor networks, and even the data transport via storage media.

The interplanetary communication is one of the applications of such networks. More precisely, a DTN approach can deal with the common case of interrupted connections or the absence of connectivity. The *Deep Impact Network Experiment (DINET)* [WTSB09] is an experiment to test DTN technology in an interplanetary environment. During the experiment, data were transmitted from a stationary node on the surface to a Space Shuttle in the orbit and then to another ground station. The Space Shuttle acts as relay, the data are neither created nor processed there. To forward the data from one hop to another, the software ION was deployed on all participating nodes. Thus, the data were encapsulated into bundles according to the *Bundle Protocol Specification* [SBo7]. The time-synchronization required by the protocol was not implemented in this project. Instead, an already available mechanism of the Space Shuttle was used, which determines the time difference to the ground station to provide a reliable reference for on-board systems [Bur].

Another example of a challenged network is the *ZebraNet* project, which tracks movement patterns of wild animals. In a field trial, Zebras were equipped with sensor nodes that tracks and store their own position at regular intervals using GPS. The challenge was to transfer the data from the mobile sensors to a sink

without the need of encounters between each node and the sink. The scientists choose to realize an approach which is very close to today's DTN concepts. The nodes distribute their collected data to the neighbors on each encounter. If there is insufficient storage on the receiving node, older records are omitted to store more recent data. Once a node is in transmission range to the sink, the data is delivered to it [JOW⁺02]. Time-synchronization was not part of this experiment. A time-stamp is received using the GPS receiver module and recorded together with the coordinates. Thus, each data entry already contains a reliable time-stamp. Furthermore, the storage policy only sorts the data using the time-stamps and drops the tail if necessary. An additional clock or a mechanism for synchronization is therefore not necessary.

In both presented experiments utilized a common time base, but no mechanism was given that could realize a synchronization without infrastructure. Instead, the necessary time-data was already present in that given scenario. Since the authors of the *Delay-Tolerant Networking Architecture* [CBH⁺07] expect that this would be always the case, the protocol specification includes a dependency on a global synchronized time. The authors of the document declare in Section 3.7:

DTN nodes are assumed to have a basic time-synchronization capability.

Outgoing data get a global time-stamp and a life-time assigned. This indicates the age of data and when they may be discarded. In addition, the time-stamp is used, in combination with the sender's address, as unique key which is used to refer specific bundles.

The dependency on a common time-base has led to intensive discussions in the DTN research group. In a discussion on the DTNRG mailing list[Ish], *Stephen Farrell* called it a mistake to assume that nodes in a network always have a good time-base. Whereupon *Lloyd Wood* drew attention to the fact that the *Delay-Tolerant Networking Architecture* implies that.

Kevin Fall, a co-author of that specification, justified the original decision to use an absolute time-stamp with scenarios in which the usual mechanisms fail. For example, it is not possible to determine whether data is still valid, if these were transported on a passive medium, such as a USB stick. Because such a device can not determine the transmission or storage time of a bundle. He also mentioned that a consistent time-base are already required solely for many routing procedures and some applications. Thus, existing time-data can be utilized for the protocol itself. Before the discussion stopped, *Joseph Ishac* asks whether it requires truly global synchronized clocks in order to solve the issues which are at the moment solved by using time-stamps.

Without Clock Synchronization

The dependency on synchronized clocks is quite challenging for some DTN scenarios. Thus, an Internet-Draft [FMO09] was published by the DTN Research Group, which proposes a way how DTNs based on the BP can operate without clock synchronization. Since the time-stamps in each bundle are mainly necessary to identify expired data, two mechanisms are proposed to mitigate that requirement.

■ Hop Counter

Instead of a time-stamp, a counter can be used, which is incremented by every station. Once it reaches the previously defined maximum value of the counter, a bundle is considered as invalid. This method is suitable for networks which generally do not have a global knowledge of time.

■ Deferred Window Scheme

If a single node in the network does not have any time information because it has been recently booted and not synchronized yet, it has the ability to add a special marker to the bundle. In those cases, the bundle gets a life-time but no time-stamp is assigned. Once such a bundle is forwarded to a node with reliable time information, the bundle is altered and a valid time-stamp is assigned to it. Thus, even if the bundle has exceeded its life-time, a bundle node will only consider that amount of time for expiration, that a bundle has existed after a valid time-stamp has been set.

Another approach is presented in [BFB10]. Using an *Age Extension Block* (AEB), the relative age of a bundle is tracked. The approach requires that each node can measure how long a bundle spent on a node before it gets forwarded. A simple approach would be to associate an incoming bundle with a time-stamp on arrival. Before a bundle is forwarded to another node, the time it has been spent on the current node is added to its relative age. Furthermore, it is necessary to consider the age of bundles located in the persistent storage. If the relative age exceeds the maximum life-time of a bundle, the bundle is considered as obsolete and must be deleted. As Joseph Ishac already suggested, no synchronized time base is needed for this approach.

Clock Synchronization

All the proposals mentioned before reduce the dependency on global time-data in the network. However, there exists routing algorithms and applications that rely on synchronized clocks (see Section 5.1 for more details). The *Double-pairwise Time Protocol* (DTP) [YCo8] considers synchronization in DTN scenarios using NTP. In

order to compensate intervals of disconnection, the approach improves the accuracy by sending pairs of messages to estimate the relative clock drift. In classical networking the accuracy of NTP is sufficient due to the short synchronization intervals. As soon as the intervals become larger, the time error gets more significant. While the DTP approach still rely on NTP structures and improves their accuracy, the Distributed Asynchronous Clock Synchronization (DCS) algorithm [CS10] is the first publication that deals exclusively with the synchronization of clocks in DTNs. This approach avoids the synchronization to a globally given time. Instead, the nodes collect information about all other clocks in the network and form a consensus. In addition to the calculation of a common time, the clock-skew is adjusted using the collected measurements.

2.5.3 Sensor-networks

In the future vision of sensor networks, thousands of small sensor-nodes are distributed in an area to monitor. Each of these nodes has its own power supply and an interface for wireless communication. Instead of forwarding all collected data directly to a sink, intelligent nodes may process the data cooperatively within the network. They can perform a simple analysis to determine whether the gathered data is of interest or not. For example, a node could be programmed so that only a message is sent, if the measured temperature exceeds a certain threshold value. The requirements of such sensor networks are similar to those of DTNs. Thus, we will consider at this point existing work on time-synchronization in sensor networks.

The design of sensor nodes is subject to certain conditions. First, the unit price must be kept low in order to build an affordable large scale network. Second, a sensor node should be small and independent of an external power supply. Most of the time a battery or a solar cell is used. Both options require a well-planned energy strategy.

The need for time-synchronization gets clear if a sensor network is considered as a distributed system. In order to take advantage of distributed measuring, the data gathered by each individual node must be correlated in a temporal sequence. As an example, we consider two nodes which can perceive noises. If both nodes recognized a sound, they can send their measurement data to a central node for processing. That node decides whether these measurements detected the same noise or two individual ones. Unfortunately, this is only possible in a network without jitter, because otherwise the temporal sequence would not be clear for the central node. Since this assumption is unrealistic in a wireless network, a correct

analysis of the data is only possible if each measurement has a global time-stamp associated.

Synchronized clocks are not only useful for assigning time-stamps to events. They also offer a high potential for energy savings. Because a sensor node consumes a significant amount of energy to keep the radio listening. Thus, those systems typically toggle the radio on and off according to a global schedule. To determine the time when the radio can be turned off or another node is listening, a clock synchronization is required. The *Network Time Protocol* [Mil85] already mentioned is used for this purpose in traditional network such as the Internet. However, this protocol has assumptions, which are not present in sensor networks.

- The moderate use of the CPU is at no energy costs.
- The listening on network packets does not increase energy costs.
- Occasional transmissions are negligible.

Based on these assumptions, the NTP performs a continuous harmonization of the local clock. Furthermore, the approach requires a hierarchy in which a master (stratum 0) must be selected in order to derive a time from it. Apart from the fact that such a master requires an accurate time source, such a hierarchy is usually not present in sensor networks. Even if it would be possible to determine a hierarchy for all topologies, such a structure would lead to nodes with a poor synchronization if their distance to the master is too high.

Jeremy Eric Elson illustrated in [Els03], the need of time-synchronization between nodes in a sensor network and explains the assumptions a sensor network brings with. The distance between two nodes is usually short and the wireless communication channel is shared among all nodes in range. Further, nodes may move around and topology changes must be considered in those cases. In [SBK05], a number of methods are described for time-synchronization in sensor networks. A selection of common procedures is described below.

- The **Global Position System (GPS)** provides a time-synchronization to the Coordinated Universal Time (UTC) with an accuracy of 20 ms or even better. However, it takes some time to get a sufficient signal from the satellites. Receiver hardware are usually large, expensive, consume a lot of energy, and need a line of sight to the satellites.
- **Römer's synchronization protocol** [Ro1] is originally designed for ad-hoc networks and requires assumptions typical for them. For example, it requires that the transmission delay between two nodes must be determined

with a high accuracy and the maximum drift of the local clock is known. Based on that knowledge, the approach avoids the synchronization of clocks. Instead, the time-stamp of outgoing and incoming messages is transformed to a common time. The advantage of this is that local clocks do not need to be adjusted. This in turn saves energy that would be spent in order to do so. In addition, there is no need for periodic messages to realize a continuous synchronization. The disadvantage of this method lies in the ephemeral and the locality of the synchronization, because it is only performed on data exchanges. Moreover, the magnitude of error increases with each intermediate station the data packet traverses to its destination.

- **Reference broadcast synchronization (RBS)** [Els03] approach exploits the properties of shared media where the same message is received at the same time by multiple nodes. To synchronize several nodes, each period a single node is selected to act as reference signal. It sends a reference message and the receivers exchange their time when the message has been received. Using this approach, the measurement of the transmission time, which is very hard to measure, is no longer needed. The accuracy of this approach goes up to $1.6\text{ }\mu\text{s}$. In order to extend the synchronization across multiple nodes, the authors propose a multi-hop extension. The downside of this method is that the sender of the reference can not be synchronized. At least three nodes are always needed and a synchronization during an encounter between two nodes is not possible. In addition, $O(n^2)$ messages must be exchanged to synchronize n nodes.

The approaches mentioned above have been proved to be robust and efficient for their specific scenario. However, some mentioned assumptions are not applicable to DTNs. For a start, a broken end-to-end connection is an exception in sensor networks, but in DTNs very likely. None of the approaches can maintain a synchronization in a permanently partitioned network. Furthermore, most sensor networks are static, although distributed dynamically at the beginning and the dynamic nature of DTNs would be a problem. Finally, it should be mentioned that the exhaustive consideration of energy constraints in DTNs would be not as strict as in sensor networks. Thus, approaches unsuitable for these networks might fit a DTN scenario.

3 Architecture

Many applications and scenarios presented in Chapter 2 require embedded hardware. But the existing implementations, presented in Section 2.4, are either bloated, lack features, or are not well-suited to be deployed on embedded devices due to their architecture. Even if there were implementations that implement all DTN-related and public available specifications, they still would not provide a complete solution because all would depend on an external clock source. The last resort for many deployments is a GPS receiver to get a time signal. As soon as the usage of such an external device is not reasonable, e.g. due to energy constraints or a missing line of sight, the network must somehow synchronize itself. But existing approaches can not achieve that under the conditions of DTNs. Another issue is that many approaches defined by the *Bundle Protocol Specification* are specified too broad to work flawlessly. Especially, the fragmentation approach need constraints which are not yet defined.

In this chapter, we define a scalable bundle node architecture for embedded devices as well as large-scale systems. Design decisions introduced within this chapter are based on findings presented before and the considerations of Section 3.1. We argue that an ideal architecture for production environments as well as in future research projects should be *lean*, *lightweight*, and *extensible*. Although the *Delay-Tolerant Networking Architecture* [CBH⁺07] and the *Bundle Protocol Specification* [SB07] is not perfect, both documents contain enough good concepts to use them as solid base to work on. The challenge here is to find or invent the right extensions to the BP and define reasonable policies to make the protocol work in almost any environment. That also implies that we need to find a way to deal with the potentially huge data amount each bundle can hold. Since we want to process that data even on small scale devices, this is a challenge which must be taken care of in the architectural design.

A major goal of this work is to keep the architecture extensible for requirements not yet identified. If it makes sense to replace or add components of a specific work-flow, it should be feasible. To achieve that goal, we introduce groups of components in Section 3.2. Those groups contain replaceable components and are arranged in a flat hierarchy. They provide interfaces for several tasks to other components or react to events triggered by others. The event-system is described in

Section 3.4 and distributes events to components. Those events may carry entities, which in turn contain required data in order to process the corresponding event. To allow a more intuitive programming, we choose to unveil the strength of threads and allow each component to work independently from other components. That allows us to implement un-interruptible algorithms and simplifies the flat hierarchy of concurrently working components. In Section 3.5, we will explain the details of this subject. In addition to handling data contained in entities, a bundle node implements work-flows which describe how data is being processed and forwarded to other nodes. Section 3.6 describes how the bundle node reacts to specific conditions and handles incoming as well as outgoing bundles.

The fragmentation support as specified for the BP is basically a good approach to deal with large bundles on interrupted links. However, the concept of fragmenting a bundle at any time in any place has unsolicited side-effects. But we refuse to drop fragmentation support completely because it potentially improves network throughput. Fragmentation in general must not be harmful to the network, if the bad scenarios are identified and avoided by defining the right policies. In Section 3.7, we go into the details of related issues and propose a reasonable policy to apply fragmentation to large bundles.

Since the opportunistic discovery of neighboring nodes is an important capability of a bundle node, Section 3.8 describes how the draft document for IP-based neighbor discovery has been integrated into this architecture. Different to the proposed draft, the protocol has been adopted as generic protocol for all kinds of beacon-based discovery mechanisms.

Finally, we present custom extensions to the BP in Section 3.9. They include streaming capabilities, extensions to work on top of the Internet, data compression support, bundle tracking for maintenance, and security extensions to prevent attacks not covered by standardized DTN protocols.

3.1 Considerations

The considerations presented in this chapter are the basis for the design decisions of the following architecture. We start with an explanation of platform dependencies. In order to design an architecture which fits to as many platforms as possible, we have to be aware of their differences. The next section discusses the processing overhead in DTN systems and their impact on the overall performance. Also relevant for the performance of a bundle node is the message queuing. Section 3.1.3 explains how that mechanism effects the achievable latency and throughput. Finally, we talk about resource constraints of the previously selected platforms and scenarios.

3.1.1 Platform Dependencies

In order to provide a bundle node implementation which is capable in running on as many platforms as possible, the architectural design must be prepared to get ported to a system with different method signatures or even completely different approaches to control capabilities of the underlying system. The developers of DTN2 and ION has introduced a special layer to encapsulate system-specific code into wrapping methods and classes. Exceptions and adaptations for different systems are made within that abstraction to normalize the behavior and to achieve platform-independent access to system capabilities. The actual networking implementation is then built on-top of that abstraction which is therefore platform-independent.

By utilizing the standardized POSIX [The] interface, it is possible to cover many platforms. Even uncertified platforms such as Linux™ or FreeBSD® are sufficiently compliant, so that only non-standardized routines have to be implemented separately. Considering the major operating systems on the market, the platform that differs the most from the POSIX interface is Microsoft® Windows®. Many aspects are handled differently and, thus, it requires a lot of glue code to support this platform in addition to POSIX-compliant systems.

A networking stack for DTNs need access to several categories of system procedures. As next, we summarize the most relevant ones and list them below.

■ Data management

A DTN software must handle large amount of data in each bundle as proposed in the *Bundle Protocol Specification*. Further, it must store bundles persistently in order to forward them on future encounters. To recover all bundles even after a system reboot, a non-volatile storage system is required. A simple solution is to organize all data in files. Depending on the platform, a program must access files differently. The differences can be a different path delimiter or how exactly the file-size is determined.

■ Threading and synchronization

In principle all modern platforms provide threading or at least processes. The exchange of data or simple signals between processes and threads needs synchronization using mutexes, semaphores, barriers, etc. The differences between the platforms are nuances in their behavior and how specific operations are performed.

■ Networking

DTNs are designed as overlay network and therefore can utilize any kind of

connectivity to another bundle nodes. Beside the common socket interface which is even on Microsoft® Windows® very close to the POSIX standard, there exists special protocols (e.g. IEEE 802.15.4) which require special handling dependent on the underlying platform.

■ Link management

In order to react dynamically to topology and connectivity changes, a DTN implementation must be aware of active links and their corresponding addresses. Depending on the platform, a dedicated interface provides access to information about interfaces and might even notify a listening process about changes. In the latter case, a process does not have to request all details over and over again. What kind of information is accessible and how this is accomplished is platform-specific.

Beside the identified categories from above, a good DTN implementation has usually logging capabilities which should be mapped to system-specific routines for logging. Additionally, timing functions are required to manage expiration of data or to more precise time measurements. Luckily, the very system-specific routines for a Graphical User Interface (GUI) is hardly relevant for a networking stack, because it operates ideally completely in the background and is controlled by third party applications.

3.1.2 Processing Overhead

A networking stack is considered as efficient if the throughput of data is high and may utilize all the available bandwidth. If interruption occurs very often and the nodes are not constantly connected to each other, we have to consider the *contact utilization* instead of the bandwidth only. In this section, we will discuss the reasons for processing overhead which may result in a low contact utilization.

Data-flow

The datagram in Figure 3.1 shows the typical flow of a PDU traversing a DTN node. There are only two ways a PDU enters the scope of the DTN node: It is generated by an *Application* or is received through a *Convergence Layer*. First, we consider the *Convergence Layer* as entry point. A received PDU has to be decoded or parsed. During this process it is possible to enforce some policies, which might restrict large PDUs, deny invalid security signatures or simply checks if the current PDU already has been received before.

As next an optional *short-cut* decision may be made. The DTN node can decide to forward the data directly to another peer instead of storing it first in the *Persistent Storage*. This is a very complex decision and uncommon in DTN environments.

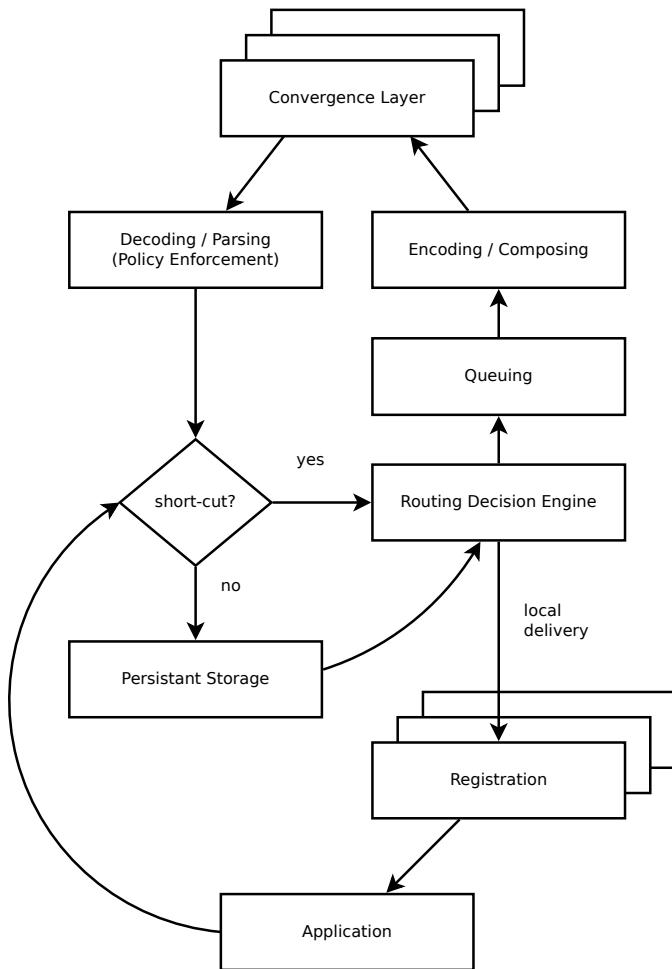


Figure 3.1: Flow of a PDU passing a DTN node

After the whole PDU has been stored, the *Routing Decision Engine* tries to determine a route. Further, it has to decide if the PDU should be delivered locally to an *Application* using its *Registration* or not. If the PDU is not local, a route has been found, and the next peer is reachable, the PDU will be queued for forwarding. Then the PDU has to be encoded or composed for the transmission and passed to the outbound *Convergence Layer*.

Alternatively, a PDU appears within the DTN node as if it is generated by an *Application*. In this case the processing starts at the *short-cut* decision and then the PDUs are handled like the others.

Storage

A challenge of DTNs, and especially for those based on the *Bundle Protocol Specification* [CBH⁺07], is the huge amount of data a single PDU can carry. This leads to a complex processing of PDUs and makes a persistent storage system essential. Since each PDU has to pass the storage, it takes place as the central and most significant performance factor in a classic DTN architecture. Typically the storage fulfills several simple, but frequently used, procedures.

■ Duplicate checking

Every time a PDU should be stored, the storage should check if a copy of that PDU is already contained in the storage. This is necessary, because the different routing approaches in DTNs might generate copies which could be received over different paths. If all PDUs are stored using a linked list, the whole list must be iterated to find out that a specific PDU is not in the storage.

■ Insertion

If a PDU passed the duplicate check and should be added to the storage, it is inserted to some data-structure, e.g. a linked list. Using a linked list, the insertion complexity is $O(1)$ which is quite good, but on the other hand duplicate checking or other operations like searching a specific PDU will have a greater complexity. Thus, a more performance oriented storage will implement some sort of index which could be arbitrarily complex. A quite simple index data-structure with better characteristics would be a sorted set. The insertion complexity is $O(\log n)$ while double checking is included in this operation.

■ Removal

The storage has to provide a way to delete a specific PDU. The deletion might

have several flavors depending on the specific implementation and the underlying platform. In case of a file oriented storage, it might be necessary to purge the corresponding files. In other systems, it would be sufficient to mark the amount of space as free. Implementations with one or more index data-structures have to modify them according to the specific removal mechanisms, which also cost some resources.

■ **Expiration**

A storage should take care of expired PDUs to re-gain space. Thus, it has to monitor when a PDU will expire. In addition to the costs of the removal process, the storage needs a way to identify expired PDUs. One way is to iterate over all PDUs and check each of them for expiration, but this approach is expensive in processing costs. Another approach is to manage an additional set of references sorted by the expiration time. The PDU which expires next is always the first PDU of such a set. Thus, checking if there are PDUs to expire is just checking the head of the set ($O(1)$). If one PDU is found to expire, a process for removal is executed.

■ **Iteration**

A very frequently used operation on a storage is the iteration. Every time the routing searches for PDUs to forward or deliver, it has to iterate through all PDUs to queue a subset of them. Typically, only a subset of meta-data is required to decide which PDU has to be selected. The payload is not necessary for this operation. A way to improve the performance here would be to maintain a routing specific index, which lowers the complexity of the search process and avoid the iteration over all PDUs.

The only way to avoid such an exhaustive search completely, is to queue a PDU in specific contact queues right after receiving/storing it. But those queues must get revised in case the topology changes. In opportunistic routing schemes this happens on every contact. Thus, such an approach is suitable for planned or predicted routing only.

As explained previously, a storage for DTNs might be complex and can contain several optimizations. In particular if performance matters, it is hard to balance between memory usage for index structures and performance. One way to by-pass the storage is shown in Figure 3.1. The junction *short-cut* tries to determine if a direct forwarding to a connected neighbor is possible. This is a very special case and is only possible if strict conditions are met. One of these is that the next chosen hop has to be connected to the DTN node. However, an implementation using such a mechanism has not yet seen in public available DTN implementations.

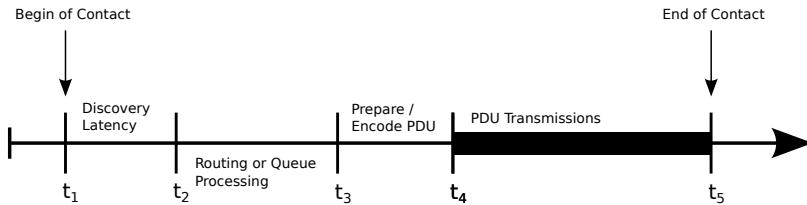


Figure 3.2: Time-line scheme for the contact utilization

PDU Encoding

As shown in Figure 3.1 each PDU has to be encoded for the transmission. Due to complex encoding schemes, this may take some time and may involve specific encoding policies.

Basically, the encoding can be done during the transmission or *a priori*. *A priori* encoding is useful if the PDU is stored encoded anyway or the DTN node has enough time to prepare the next transmission because it knows about upcoming contacts. If the encoding of the PDUs has to be done during the contact time, it takes too long to encode the PDU first, which means to copy the whole data and then send the encoded data to the foreign peer. A better approach is to encode the PDU during the transmission and write each encoded data-part directly to the CLA.

On the receiver side the PDU must be decoded or at least parsed to determine the next-hop for it. This process might also be very expensive and should be done without any copying during the transmission. Only that way it is possible to apply policy checking directly on the received data.

3.1.3 Latency and Queuing

Even in delay-tolerant networking structures it is a reasonable goal to achieve a low end-to-end delay. Thus, in this section we will discuss the additional latency introduced by the processing overhead during the contact and common scheduling approaches.

Contact Utilization

As already considered in Section 3.1.2, the DTN node claims some processing capacity for each PDU it has to handle. The different factors are broken down in the time-line scheme for the contact utilization (Figure 3.2).

The first mark (t_1) denotes the begin of a contact. During the interval $[t_1, t_2]$ the DTN node needs some time to discover the foreign peer. Under ideal conditions or when using hard-scheduled contacts, this interval would be zero. Once a node has discovered another one, it can select the PDUs to transfer. Depending on the implementation this involves the *Routing Decision Engine*. The cheapest approach would be to just select the corresponding queue for the peer and select the first PDU of it. Also this would cost some amount of time. During the interval between t_3 and t_4 , the first PDU has to be encoded for the transmission. Even if it is possible to encode the PDU partially during the transmission as described in Section 3.1.2, the encoding of the first segment consumes some time here.

As already discussed in Section 3.8 the interval $[t_1, t_2]$ is possibly huge due to energy constrains and can significantly limit the usable contact duration. Similarly to the interval $[t_2, t_3]$, the process to select PDUs to forward could be arbitrarily complex and might involve some sort of handshake to exchange routing data (e.g. summary vectors), which implies several round-trips. A way to keep that interval small is to prepare routing queues for each contact. Then the complexity is reduced to selecting the right queue. However, that approach would involve many resources if each node need to store additional data for every other node in the network. Moreover, many routing approaches depends on a data exchange during the contact. That makes an a priori queuing impossible. For that reasons, it is necessary to search for PDUs to forward during the contact opportunity in typical DTN environments. In turn, the selection of PDUs depends on the storage performance as explained in Section 3.1.2. In the worst-case all these processing takes to long for a contact. More precisely, if the interval between t_1 and t_4 is greater or equal the contact itself, there will be no chance to transmit any data.

Throughput

If the previously discussed mechanisms are sufficiently fast for the contact duration, the DTN node will try to forward as many queued PDUs as possible. While the maximum amount of data to transfer is naturally limited through the contact capacity, the PDU frequency (PDUs per time) depends on the processing overhead during the transmission.

As shown in Figure 3.1 there are several tasks to do. The sender has to load and encode each PDU for the transmission. This process involves the persistent storage and as mentioned before, this is the bottle-neck in the whole system. Further, the *Routing Decision Engine* of the sender has to select PDUs to transfer. If the queue runs short or new PDUs has been received, the selection of PDUs starts over again.

But the sender do not have to be the slowest part of the transmission in all cases. Assuming that the routing algorithm is not very complex and the storage can read

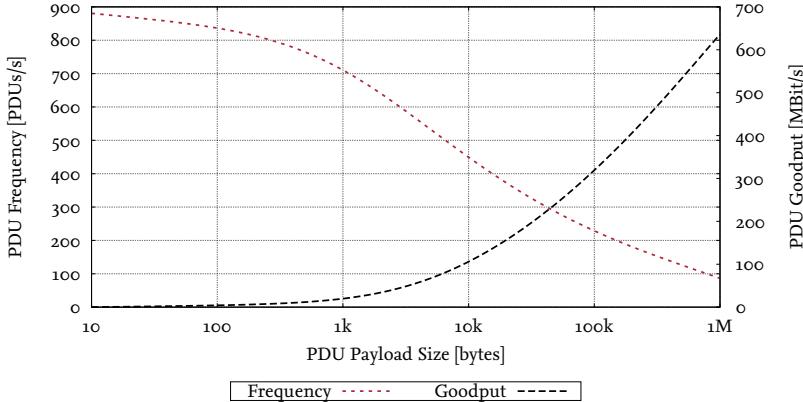


Figure 3.3: PDU frequency during a transmission in DTN2

PDUs very fast, the receiver has to decode and store the received PDUs. Since writing is more expensive than reading in general, the transmission is dominated by the receiver side. One way to improve the contact utilization would be to cache received PDUs in memory and delay the actual store procedure. But this would only help if the contacts are very short and the amount of data is not too large, because memory is limited on most platforms.

Figure 3.3 illustrates the performance dependency. It shows a smoothed result of a throughput measurement using the DTN2 reference implementation which was introduced in section 2.4.1. More details on how this measurement is performed will be given in section 6.2. In this case, DTN2 is transferring several PDUs between two stations. The shared y-axis compares the PDU frequency (on the left side) and the achieved good-put (on the right side) dependent on the payload size of each PDU.

As result, the DTN node seems to be limited to a maximum bound while sending PDUs with a very small payload. Here the processing overhead is the limiting factor and consumes all available resources to encode, decode and process the PDUs. Since the processing of the payload itself is extremely slight, the processing overhead of each PDU is dominating here. This starts changing with larger payload sizes, where the processing of the payload grows and becomes significant.

With increasing payload size the achieved throughput is also raised until an upper bound is reached. This bound is defined through the available bandwidth

minus the protocol overhead. Additionally, there might exist gaps in the transmission due to processing overhead. These gaps lower the maximum achievable throughput further.

Scheduling

Beside the obvious performance limitations caused by processing overhead and bandwidth bounds, there exists an additional factor which might have an impact on the delivery delay. As explained before each DTN node uses *Convergence Layers* to send and receive PDUs. Typically these *Convergence Layers* provide only a single channel to other peers. If the channel is occupied by a transmission, other PDUs are delayed until they are at the front of the queue. The result is that large PDUs might congest the transmission opportunity and smaller PDUs get delayed because larger ones are queued in front of them. This is especially problematic if there are PDUs with a higher priority, because PDUs can not be queued before an enduring transmission, at least as long as it is impossible to abort it. It is even possible that small PDUs starve, because they expire before they can be transmitted.

One approach to mitigate this issue would be to fragment all PDUs if they exceed a maximum size using pro-active fragmentation. That would limit the payload volume of all the PDU in the network and introduce a predictable upper bound for transmissions. Another way would be to multiplex several transmissions using one channel, but it is hard to decide how many of concurrent transmissions are useful. Transmissions of several PDUs in parallel imply more overhead and prolong the transmission time of individual PDUs.

At this point an example might help to clarify the mentioned issue. If we assume a radio channel with 4 MBit/s bandwidth and there are three PDUs to transfer. In encoded format the PDUs have sizes of {1}: 100 kbyte, {2}: 10 kbyte, {3}: 1 kbyte. The PDU {1} takes at least 200 ms to transmit. The second 20 ms and the third 2 ms. If all three PDUs are queued to the same destination, the third one will be delayed by at least 222 ms. If we introduce scheduling with priorities, it is possible to prioritize {3}. That would change the queuing order to {3}, {1}, {2}, because {3} will be inserted into the queue in front of {1}. But if we queue {2} and {3} after the transmission of {1} has been started, the other PDUs have to wait until the transmission is done. Thus, they will be delayed by at least 200 ms. Fragmentation would mitigate this issue by splitting {1} into several smaller PDUs, e.g. parts of 10 kbyte. In such a case, the maximum additional delay for a prioritized PDU would be only 20 ms instead of 200 ms.

Basically there exists many scheduling approaches with different goals. Some might prioritize PDUs using special flags and others prefer PDUs with a short remaining life-time to raise the probability that they reach the destination before

they expire. An evaluation [DW12] of different common scheduling schemes for DTNs shows that they do not raise the global throughput of the network. But at least if the individual latency of each PDU matters, a specific scheduling approach should be applied.

3.1.4 Resource Constraints

DTNs are deployed in space, rural areas, and in areas with broken infrastructures. All in common, these areas have usually no cables, no wires, no external power. Either the nodes are already equipped with a power source that lasts as long as necessary or they must harvest energy by utilizing solar power modules or wind generators.

To reduce the energy consumption, nodes are realized using low-powered embedded hardware or even sensor-node platforms [vZBPW12]. For Internet deployment scenarios, the costs for those devices must be low to keep a deployment affordable. In consequence of these constraints, the hardware offers only limited resources. The CPU is clocked low and the volatile memory is sized between 16 MB to 128 MB. This in turn affects achievable capacity of the network as mentioned before in Section 3.1.2.

In all scenarios presented in Section 2.2, nodes are communicating using wireless technology which is counter-productive to the mentioned energy-constraints. Driving a wireless transceiver costs a lot of energy even if this is just listening for incoming transmissions. For that reason, radios are often duty-cycled to save-energy [LWSW11]. But that increases the discovery latency in opportunistic scenarios where beacons are used to detect encounters.

3.2 Components

Basically, there exist two categories of components. The static components are permanently required at run-time and typically not designed to get replaced by another implementation. Dynamic components provide a functionality defined by an interface via a specialized implementation. Some of the dynamic components are replaceable but also required because they provide basic functionality like the storage sub-system, while others are optional (e.g., CLAs).

Before the complete architecture is explained in detail, the five major component groups of the architecture, shown in Figure 3.4, are discussed. All groups interact with each other via synchronous calls or raised events. In any case, the Core is the central group and glues all the components together.

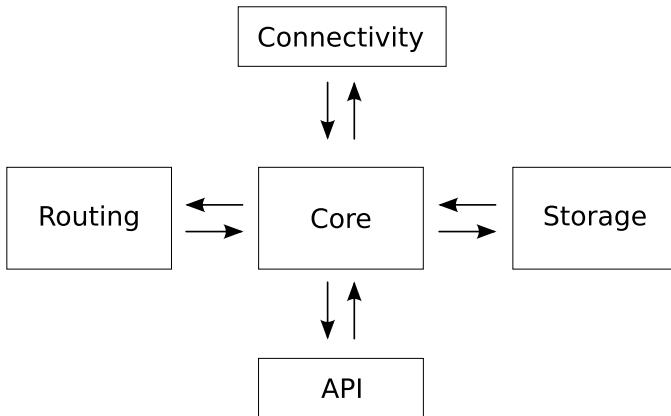


Figure 3.4: Architecture of the component groups

Core

Depending on the configuration, the Core instantiates and manages all components and allows them to interact with each other. Additionally, to direct function calls from one component to another, the core provides the event-system which generates and dispatches immediate, asynchronous and bounded events to interested components. An additional fragment manager tracks all the bundle fragments in the system and reassembles them if all fragments of a full bundle are available. Furthermore, the Core generates an event every second to provide a centralized clock-tick to time-dependent components.

Storage

The storage sub-system is reachable via the Core. Since there are several ways to implement a storage, this is a required but dynamic component. It can easily be replaced by another component which implements the *storage* and *seeker* interface. The *storage* interface is used to persistently store, retrieve, and remove bundles. For each of these operations, the bundle is identified by its unique bundle identifier.

The *seeker* interface provides a way to select bundles matching specific criteria defined by a *bundle selector*. Such a selector provides a callback function which is called by the seeker to decide, if a bundle should be selected or not. Each time a component requests a set of bundles, the storage iterates through all bundles until a limit of bundles is selected or all bundles have been considered. The *seeker*

interface is not necessarily implemented by a *storage component*, instead a dedicated *bundle index* may provide the *seeker interface* for the selection of bundles. This way, a customized ordering of bundles within a custom *seeker* implementation, affects all components independently of the storage component.

Connectivity

A bundle node would be almost useless if it could not inter-connect with other nodes. The *Connectivity* group covers all connectivity related components, including all kinds of CLAs and mechanisms to discover other nodes. CLAs are optional dynamic components and provide an interface to send and receive bundles using a specific underlay network. In some cases, such a component does not connect to a conventional network. Instead, any sort of approach to transfer bundles to other nodes meets the requirements to get adopted as CLA. For example, a USB flash drive might be used to store bundles and deliver them to other connected hosts.

Routing

The group of components related to routing is responsible for queuing bundles for delivery as soon as a contact becomes available. Beside the specific algorithm like epidemic [VBoo] or PRoPHET [LDS03] routing, this component group contains several supplementary components to handle retransmissions, exchange summary vectors and other routing data between bundle nodes, and algorithms for scheduling and flow-control.

Application Programming Interface

The last component group to present includes all components related to the API. According to the *Bundle Protocol Specification* [SB07] an API allows applications to create and destroy registrations as well as send and receive bundles. Beside the application endpoint related functions, the bundle node should provide several management functions via the API to retrieve data about the available neighbors or the stored bundles. Since there exist more than one way to implement an API, there are also several co-existing components in the bundle node to provide different ways to call the API functions.

3.3 Entities

Beside the components in the system there exist several entities generated and processed by components. All entities presented in this work are data-objects which

contains bundle-data, references, some kind of state, or collected data. In this section, the major entities will be discussed.

BLOB

A central aspect of the bundle routing is the storage. Received bundles are stored in the storage, will be forwarded and might be deleted afterwards. Since the size limit of a bundle is quite large and each bundle may have an unlimited number of blocks attached, an approach is required to store the Service Data Units (SDUs) of the blocks on a sufficient large storage device or in memory, if a persistent storage is not available. The interface to access the SDU should be generic and transparent concerning the actual data location.

The Binary Large Object (BLOB) handles large amounts of data in a common and efficient way, no matter if the data is stored on a persistent device or in volatile memory. For performance reasons, each entity instance may be replicated without copying the SDU by holding a reference to it. The entity is also responsible in protecting the SDU against concurrent access through multiple copies.

Event

The event-system is one of the fundamental mechanism in this architecture. As explained before, events allow to decouple components from each other using asynchronous and bounded events. Events are distinguished by different event-types which are derived from the Event entity. The interface of this entity allows us to access the priority parameters for generic processing in the event-switch, while specific event-types may encapsulate additional entities. Section 3.4 contains a complete list of all event-types.

EID

The EID entity encapsulates an endpoint as def and makes it comparable to others. It may contain optimized scheme-specific methods to extract and compare the host or application part of the EID. Instances of EID are used as element in many other entities like BundleID, Bundle, MetaBundle, Node and Registration.

BundleID

As specified in Delay-Tolerant Networking Architecture [CBH⁺07], each bundle is unique within the source endpoint, creation time-stamp, and creation time-stamp sequence number. In case of a fragment, the data offset/length is also considered. The BundleID entity encapsulates all these values to a unique identifier for each bundle, which is used as reference to a unique bundle or fragment.

Bundle

The entity **Bundle** is based on the `BundleID` and contains references to the complete data of a bundle. It provides access to the primary block content and the attached extensions.

Block

A **Block** is part of a **Bundle** and contains block specific payload and optionally some `EID` entities. This is the base for more specific entities implementing extensions to the BP.

MetaBundle

The entity **MetaBundle** is based on the `BundleID` and extends it with a summary of useful parameters of the origin bundle. The additional data includes the life-time, the destination `EID`, the `EID` to send reports to, the custodian `EID`, the application length, the processing flags, the expiration time-stamp of the bundle, the remaining hop count limit, and the network priority.

Node

A **Node** entity represents another discovered or connected peer. A **Node** is identified by its corresponding `EID` and contains additional data about how to transfer bundles to the peer. There may exist several CLA addresses and additional attributes. Whether a peer, represented by a **Node**, is marked as available, depends on the availability of usable CLA addresses.

Registration

Registrations are used to store the state of applications. Using the API an application creates a **Registration** and can attach several `EID` entities to it. If the destination of an incoming or stored bundle matches one of the attached `EIDs`, then this bundle will be offered to the application. A **Registration** may be in *active* state when the application is continuously connected or in *inactive* state if the application is mostly not connected and should poll for new bundles periodically. For the latter case, the application has to specify a life-time for the **Registration** at the moment it is put in *inactive* state.

Additional to multiple `EID` instances, the **Registration** holds a set of bundles already delivered to the application. This is required to select only undelivered bundles using the `seeker` interface, every time the **Registration** is notified about new bundles in the storage.

3.4 Event-system

The event-system is part of the Core and permits concurrency as well as isolation of different threads. It accepts raised events and forwards them to those components which subscribed to the specific event-type. Each event-type is derived from the Event entity and inherits a Dispatcher which manages the subscriptions of the components for a single event-type. A raised event is either *immediately* delivered to all its subscribers or detached to an event-queue for *asynchronous* delivery. Events for *immediate* delivery block the calling procedure until all components have processed the event. This is useful in some cases to prevent congestion in the event-queue or to distribute the processing over multiple threads. An asynchronous event-type is always forwarded to the event-switch and queued to one of three event-queues. These queues provide the priorities *high*, *standard*, and *low*. Thus, an event with a *high* priority might overtake an event with *standard* or *low* priority.

The Figure 3.5 outlines the flow of a generated event. In this case the component which raised the event is Component 1. It calls the Dispatcher to generate a new Event instance. As next the Dispatcher decides if this event is an *immediate* event or not. If yes, all the subscribers (here Component 2) are called to process the event. If the event was not an *immediate* event, it is queued in the Switch and the procedure ends. As next, the thread of the Switch has to handle the event in the queue. This is shown in Figure 3.6. The Switch retrieves the Event instance from the queue and calls the Dispatcher to distribute the event to all subscribers (here Component) and the procedure ends. In an implementation of that mechanism, this procedure would run in a loop so that this thread only ends if the software is terminated.

To prevent an overflow of a queue, the event generation has to be slowed down in some cases. This is achieved by using a blocking call of *immediate* events or using an *asynchronous* event with a semaphore to limit the amount of instances. The latter approach is named *bounded* event and allows more than one event of a specific type at once in the queue, but limits the quantity to a hard limit. The event-processing can benefit in this case from multi-processor architectures and works similar to a pipeline. To increase the performance further, the event-switch can instantiate multiple workers to process the event-types in the queues in parallel. This way the concurrency can scale with the system resources.

In this architecture, several event-types are required. Each event-type is raised for an individual reason and may carry different entities to the subscribers. In the following, the different event-types will be explained.

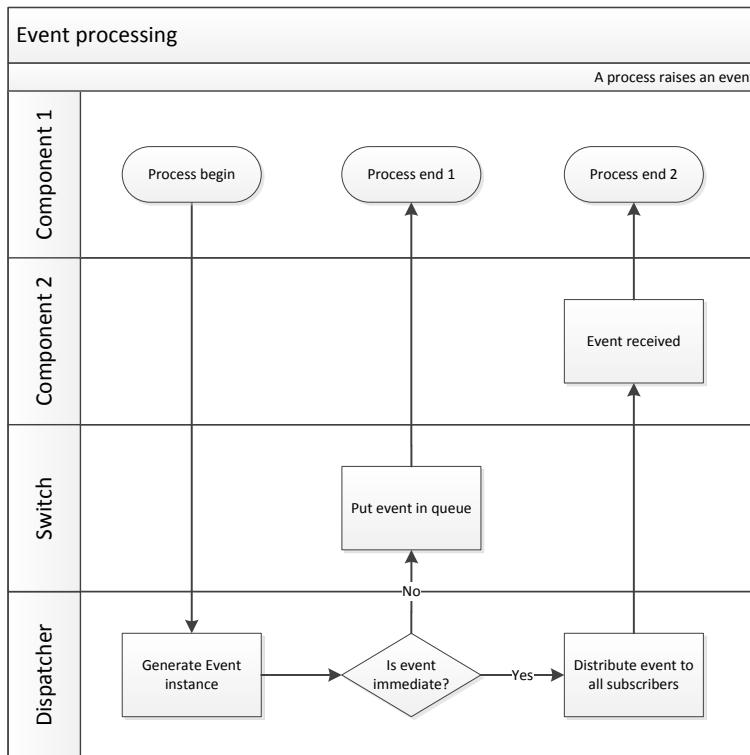


Figure 3.5: Flowchart event generation procedure

■ BundleEvent

A BundleEvent is raised, if a bundle is deleted, forwarded, delivered, received, stored, or custody was accepted. The event carries a MetaBundle as reference to the original bundle. Instances of BundleEvent are directed to the component which generates BSRs.

■ BundleExpiredEvent

If a bundle expires while it is stored in the storage, then this event is raised. The event carries a BundleID as reference to the origin bundle.

■ BundlePurgeEvent

If a bundle is finally delivered, the bundle is released to get purged out of

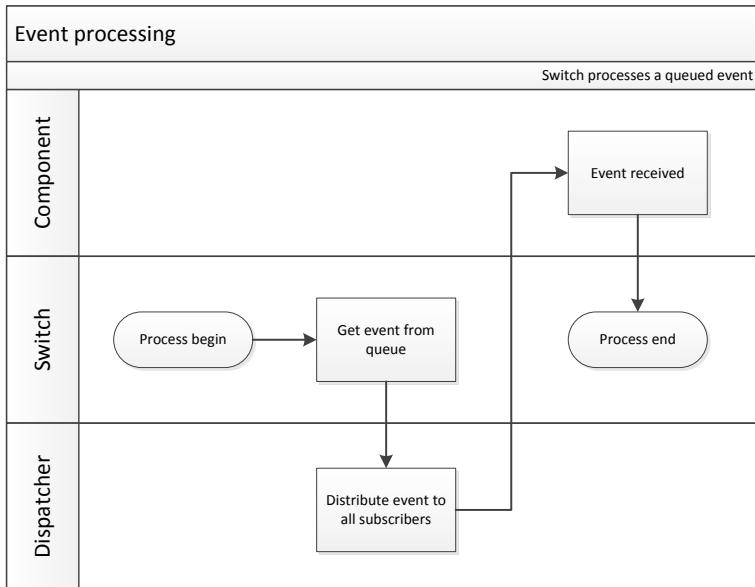


Figure 3.6: Flowchart event processing of asynchronous event

the storage using this event. The event carries a `MetaBundle` as reference to the origin bundle.

■ `BundleReceivedEvent`

If a bundle is received through the API or a CLA, this event carries the received Bundle to the processing component. To prevent an overflow by incoming bundles, this event is bounded and has a low priority. Here this mechanism is called *back-pressure* and may lead to a more deterministic processing flow.

■ `ConnectionEvent`

A CLA may hold a connection state with several peers. This event signals if a connection is initiated, up or down and carries the corresponding Node and connection state.

■ `CustodyEvent`

The `CustodyEvent` triggers the custody acceptance or the rejection mecha-

nism, raised by the storage. The event carries a `MetaBundle` as reference to the origin bundle and the action to indicate acceptance or rejection.

- **GlobalEvent**

If some state within the Core changes, a `GlobalEvent` is raised. This event can express various states related to the bundle node's life-cycle. It can indicate if the bundle node will shutdown, is in a suspended state, is busy or idle, and if the configuration has been reloaded. In general the events are used to control the bundle node from a management API or a platform-dependent integration layer.

- **NodeEvent**

If the state of a `Node` changes, this event signals if the `Node` is available as neighbor or has disappeared. Furthermore, this event is raised if additional data has been added to or removed from the `Node`.

- **NodeHandshakeEvent**

This event is raised if the routing exchange between the local node and a peer is completed, peer data was updated or the bundle node has replied to a response. The event carries the current routing exchange state and the EID of the peer. Details on this are discussed in Section 3.6.1.

- **P2PDialupEvent**

This event is raised if a new interface should be considered for peer connections or a previously announced interface is gone. It carries the action (up/down) and the name of the interface.

- **QueueBundleEvent**

If a bundle is stored in the storage and ready to get routed, this event is raised. Routing extensions and Registration instances react to this event and try to route or deliver the new bundle. Additional to the corresponding `MetaBundle` the event provides the origin EID.

- **RequeueBundleEvent**

If a transmission fails with a temporary error, the bundle is re-queued using the `RequeueBundleEvent` event. The event carries the corresponding `BundleID` and the peer EID.

- **StaticRouteChangeEvent**

The `StaticRouteChangeEvent` is used to alter static routes of the static routing component and is raised if such a static route expires. It carries the event-

type (add, delete, clear, expired) and the route. A static route consists of a next-hop EID and a pattern matching the bundles source EID.

- **TimeAdjustmentEvent**

Every time the internal clock is adjusted, this event is raised. The event carries the current measured clock offset and a clock rating value. For more details on time-synchronization, see Section 5.2.

- **TimeEvent**

Many mechanisms rest until some time has past. This event announces the current time-stamp to components with time-dependencies. The event carries the DTN time-stamp with second precision.

- **TransferAbortedEvent**

If a transmission of a bundle to another peer is aborted, this event is raised. The event carries the BundleID of the bundle and the peer EID of the transmission.

- **TransferCompletedEvent**

If a transmission of a bundle to another peer is completed, this event is raised. The event carries the `MetaBundle` of the bundle and the peer EID of the transmission.

As listed above, there are plenty of events. The events necessary to generate BSRs are summarized by the `BundleEvent` while others seem to similar but are expressed by dedicated types. The reason for this distinction is that these events carry different entities and are destined for different purposes.

3.5 Concurrency

Since the CPU clock cap is reached, modern systems are based on multi-processor respectively multi-core architectures to increase their performance. Even on mobile devices a dual-core CPU is the lower bound. Thus, processing multiple workflows in parallel is an important principle to utilize the existing resources efficiently.

As explained before, this architecture is partitioned into several components. Each component may run in an own thread or only react to incoming events. We distinguish between those types using *independent* (separate thread) and *integrated* components (event processing only). In addition to dedicated threads in the components, the event-processing may have several workers (see Section 3.4) depending

on the available resources. Thus, even integrated components may work concurrently.

Beside all the advantages of scalability when using multi-threading, the drawback of this approach is the concurrent data access. If multiple threads share the same data, a mechanism is required to protect the data against concurrent write access and being altered during a read operation. Therefore, if multiple threads require exclusive access to the same data, no performance gain is expected compared to single core processing. However, we expect to exchange multiple bundles at the same time and probably with multiple neighbors. The pipe-line approach where each of the components handle a part of the bundle processing chain would benefit in this case from multiple processors.

To protect data against concurrent access, *mutexes* [Tano1] are used for mutual exclusion. A *mutex* is a simplified version of a *semaphore* and allows only one process to enter a critical region at once. If a second process would like to enter the same region, the call will be blocked until the previous process has left the *mutex*. A *semaphore* has a counter which is incremented on a leave and decremented on each enter. If the counter is zero, the enter call will block until the counter is incremented again. By setting the counter initially to a value $n > 0$, the *semaphore* allows n processes to enter the critical region at once. This is used to limit available resources or to solve synchronization problems. Another important utility to synchronize multiple threads is called *conditional*. It allows a process to wait until a condition is met within a critical region without blocking the region for other processes. More implementation specific details on concurrency are explained in Section 4.5.

3.6 Work-flows

In this section, the major algorithms and work-flows will be illustrated. They define how the bundle node reacts under specific conditions and handles incoming as well as outgoing bundles.

3.6.1 Routing Data Exchange

Before any bundle is selected for forwarding (Section 3.6.2), most routing algorithms need to exchange some data first. Often, a summary vector representing all the bundles available in the neighbor's storage is required to select only bundles not already received by the neighbor. Other routing schemes need some specialized data as the Delivery Predictabilities of PProPHET [LDDG12]. To avoid implementing a dedicated data exchange mechanism in each routing module, a generic and extensible routing exchange approach is required.

An approach to realize such data exchange is to push relevant data to other peers on every contact and each time this information has changed. But this approach would introduce two issues. First, the bundle node has to know which peer is interested in which data and second, sending the data on every change would potentially be on each received bundle and is therefore unfeasible. Thus, instead of pro-actively pushing data to peers, we rely on a request-response approach to guarantee that the requesting peer is interested in the offered data and updates are only sent, if this is necessary for further processing.

The routing data exchange protocol introduced below, uses request and response messages encapsulated in bundles. This way, it is independent of the underlying CLA. Each routing exchange bundle has a priority level of *high* and is addressed to the application endpoint routing for the scheme dtn or application number 50 for the scheme ipn. Since this bundle is used for exchange between two peers only, the life-time is typically short (e.g. 60 seconds) and the scope should be limited to a single hop using the Scope Control using Hop Limits (SCHL) Extension Block [Fah0]. To avoid issues with not synchronized clocks as discussed in Section 4.13.1 of the next chapter, the bundle should also carry an AEB and a time-stamp set to zero. The disadvantages of the AEB (as shown in Section 4.12.1) are negligible due to the fact, that this bundle is limited to one hop and has a very short life-time.

Message Type (*)	Number of items (*)
Request Type (*)	
Request Type (*)	
...	
Request Type (*)	

(*) marked fields are encoded as SDNV

Figure 3.7: Routing Exchange Message – Request

The request message-format is shown in Figure 3.7 and the response message-format is shown in Figure 3.8. Both starts with the message type which is 0x01 for requests and 0x02 for responses. Then, the number of items is encoded followed by the requested types respectively the response items. Requests are just identified by an individual request code. Responses contain items for each request-type. Un-processable request-types are simply ignored. At this time, we define five different request codes.

■ BLOOM_FILTER_SUMMARY_VECTOR (0x01)

The summary vector Bloom-filter summarizes all the bundles known by

Message Type (*)	Lifetime (*)
Number of items (*)	
Response Type (*)	Response Data Length (*)
Repsonse Data	
Response Type (*)	Response Data Length (*)
Repsonse Data	
...	
Response Type (*)	Response Data Length (*)
Repsonse Data	

(*) marked fields are encoded as SDNV

Figure 3.8: Routing Exchange Message – Response

the peer. A Bloom-filter is a space-efficient and well compressible data-structure. Elements can be added and checked if they are part of the set. A check for inclusion has a false-positive probability which depends on the size of the Bloom-filter, its parameters, and the number of the elements already added to it. Using this filter it is possible to mask out all the bundles already known by the peer and just select those which are not already known. The advantage of this filter compared to a complete bundle list is the static size of the data structure. Moreover, the filter is well-compressible to an even smaller size if it is sparsely filled.

■ **BLOOM_FILTER_PURGE_VECTOR** (0x02)

The purge vector Bloom-filter contains all the bundles which are addresses to a singleton endpoint and already delivered to an application. Those bundles should be purged out of the network. Using this filter it is possible to select all bundles to purge in the storage and delete them.

■ **DELIVERY_PREDICTABILITY_MAP** (0x03)

The PROPHET routing scheme requires to exchange delivery predictabilities to the neighbors. These are used to decide which bundles should be forwarded to a peer and which not. This set contains several EIDs where each one has a float value assigned. See Section 4.9.5 for more details.

■ **PROPHET_ACKNOWLEDGEMENT_SET** (0x04)

The PROPHET routing scheme defines a set of acknowledgements which is distributed over the network. This set contains IDs of bundles which are addresses to a singleton endpoint and already delivered to an application. Sim-

ilar to the BLOOM_FILTER_PURGE_VECTOR, those bundles should be purged out of the network. See Section 4.9.5 for more details.

■ REQUEST_COMPRESSED_ANSWER (0xff)

The compressed answer request is a special flag which does not expect an explicit response. Instead the flag indicates that bundle compression is supported by the requesting peer and the size of the response can be reduced by applying bundle compression as specified in Section 3.9.3.

The received routing information is stored in a neighbor database within the routing component group and is accessible for all components. If some necessary data is not available in the database, a component can trigger the routing exchange. Then all registered components are asked to amend request-types to a new request message.

The exchange of the request-response messages is depicted in Figure 3.9. In this example, the Node A has generated an exchange request with types *a*, *b*, and *c*. This request is then transmitted to Node B where all registered components are asked to respond to the requests. Node B has components which know how to respond to *b* and *c*, but none react to request *a*. The responses of all components are collected and aggregated to one response message, which is transmitted back to Node A. Node A stores the received data into its neighbor database and marks the data with the life-time specified in the response message. If the life-time exceeds, any access to data related to Node B triggers the routing exchange again.

If a routing exchange request is sent to a peer which does not support the routing exchange procedure, the peer will simply not respond to the request. Each routing extension with a hard dependency on the requested data will abort its routing procedures for that peer. However, there may exist extensions which have specified the routing exchange as optional. In this case, the routing extension can access locally cached data which is certainly inconsistent with the remote data. It is up to the routing extension to specify, if this is acceptable or not.

3.6.2 Bundle Selection

Routing in DTNs differs from the classical routing. Very frequent changes of the topology are combined with routing in an additional dimension, the time. At every alteration, the bundle node has to consider all stored bundles to determine which of them can be forwarded under the new conditions. Using the event-system introduced in Section 3.4, all changes to the network topology as well as announcements of new bundles are distributed using events. Events relevant for the selection of bundles within the routing components are explained below.

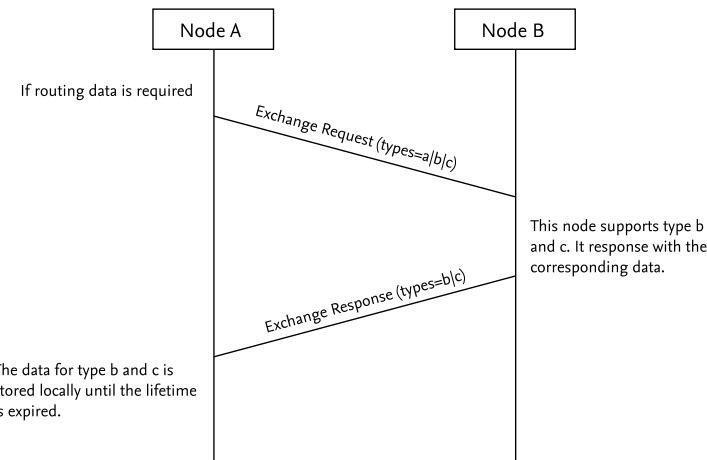


Figure 3.9: Request-Response Scheme of the Routing Exchange

■ ConnectionEvent

The event announces a change of the connection state between the local node and a peer. Every time a new connection is established, it has to be considered as new path and the routing components have to check if bundles could be forwarded to the peer using the new connection.

■ NodeEvent

This event is raised every time a new peer is discovered. Every new peer has to be considered as potential next-hop for any bundle in the storage.

■ NodeHandshakeEvent

If the routing components require unavailable or expired routing data, the routing process is aborted and an exchange is initiated. On completion of this exchange, the routing components have to check again if bundles in the storage could be forwarded. This event indicates completed exchanges or replies to exchange requests.

■ QueueBundleEvent

If a new bundle is received from an application or another peer, the bundle is forwarded to the routing components which generates a QueueBundleEvent, if the bundle is not a duplicate of an already received bundle and the processing of the bundle did not fail due to other reasons. The routing compo-

nents have to recognize this event to check if the bundle could be directly forwarded to another peer.

- **StaticRouteChangeEvent**

If an entry in the static routing table changes, this event is raised. Since the routing has to consider this topology change, it has to check all bundles if any of them can be forwarded using the new topology.

To limit the number of bundles selected by routing extensions, a *transfer capacity* is defined. For this purpose, the neighbor database holds a counter for the number of bundles in transit for each peer. On each new selected bundle, this counter is incremented and decremented if a transfer is completed or aborted. If the counter exceeds a configurable limit, the transfer capacity is exhausted and the routing components will stop their iteration through the storage until enough transfer capacity is available again. The events *TransferCompletedEvent* and *TransferAbortedEvent* indicate a change of the capacity to all routing extensions.

3.6.3 Bundle Forward

An example of a successful Bundle Forward procedure is shown in Figure 3.10. As explained in 3.6.2, the Router starts to select bundles from the storage if an event announces a change in the topology or new bundles appear in the storage. If a bundle is selected for forwarding, the Router first checks if the bundle is already queued for the selected peer or if the transfer capacity for that peer is exhausted. If not, a bundle transfer is generated and queued to the Core. The bundle transfer contains the EID of the next-hop and the BundleID.

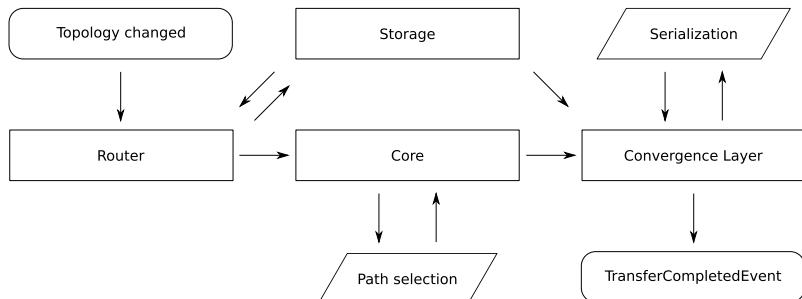


Figure 3.10: Bundle Forward

The Core selects a valid path (e.g. TCP via TCP-CLA) and queues the bundle transfer to the corresponding CLA. If no valid path was found, the transfer gets aborted and the TransferAbortedEvent is raised. If it is necessary to set-up a P2P connection first, the connection set-up is initiated and the bundle transfer is re-queued using a RequeueBundleEvent. See Section 3.6.7 for further discussion on that.

Finally, the bundle transfer is queued to the Convergence Layer which retrieves the complete bundle (with payload and all blocks) from the storage, apply security extensions (BAB), and serialize the bundle into the binary format defined by RFC 5050 [SBo07]. If the transfer was successful, a TransferCompletedEvent is raised. A TransferAbortedEvent is raised if the transfer fails for a permanent reason and a RequeueBundleEvent is raised if there was a temporary failure.

3.6.4 Bundle Reception

An incoming bundle is either received by a Convergence Layer or the API. As shown in Figure 3.11, an incoming bundle passes the Deserialization which does various checks for expiration, security and limiting policies. Due to performance reasons, these operations are superficial or at least very simple. More thorough checks are applied at a later stage. Additional to operations on all received bundles, those received by the API require further processing to assign a unique bundle ID, apply pro-active fragmentation, compress the payload if requested (see Section 3.9.3), and apply security policies (e.g. encrypt the payload or add signatures). Once a bundle is received completely, a BundleReceivedEvent is raised to forward the bundle to the Router.

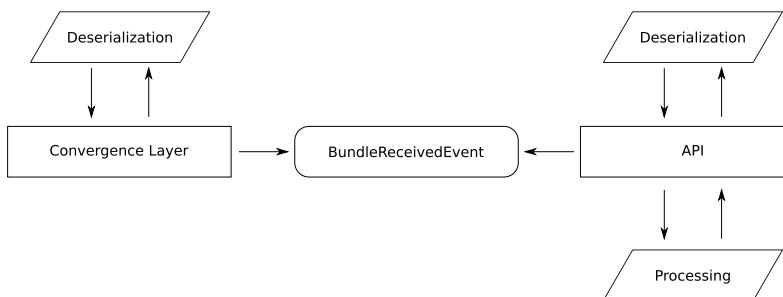


Figure 3.11: Bundle Reception

Afterwards, the router receives the BundleReceivedEvent as depicted in Figure 3.12 and does further processing. While bundles received by the API only need to

pass the duplicate check, bundles from a Convergence Layer has to pass a thorough security check for signatures, extension blocks may be altered (SCHL Block and Tracking Block) and the bundle is added to the stored neighbor entry in the neighbor database to prevent loops. Finally, the bundle is stored in the Storage and a QueueBundleEvent is raised.

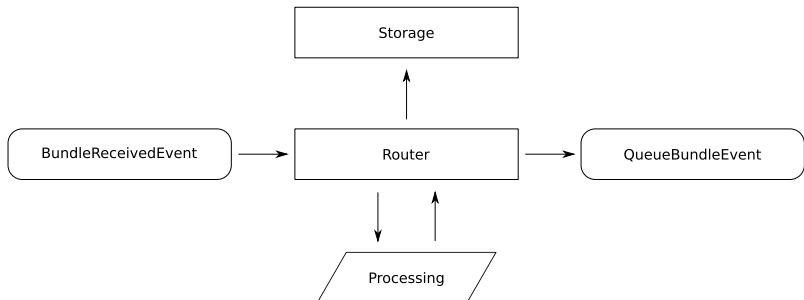


Figure 3.12: Bundle Processing

3.6.5 Bundle Delivery

The delivery of bundles by the API is done independently of the bundle processing of other components. As shown in Figure 3.13, the API is triggered by a QueueBundleEvent and then it starts to search for bundles to deliver. This task is similar to the Bundle Selection (Section 3.6.2) of the Router. Once a bundle is chosen, the API loads the whole bundle and applies final processing. That includes decrypting the payload and inflate compressed payload (see Section 3.9.3). Finally, the bundle is serialized for the transmission to the application and a BundleEvent is raised, as soon as the application confirms the reception.

3.6.6 Bundle Retransmission

Every time a bundle transmission is completed or stops with an error, one of three events will be raised. A TransferCompletedEvent is raised if the transfer was completed. If the transfer fails due to a permanent error, for example if the peer has disappeared, the TransferAbortedEvent is raised. Both events tell the Routing that more transfer capacity is available and that the Bundle Selection should be started again. Beside these two final states of the bundle transfer, the RequeueBundleEvent indicates a temporary failure. The bundle node should pause that transmission for a moment and then start a retransmission. Important

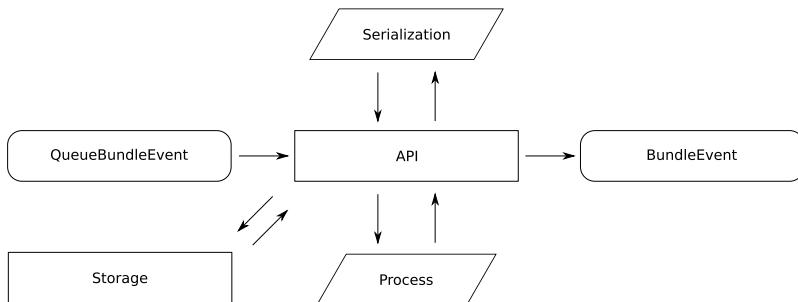


Figure 3.13: Bundle Delivery

here is that the allocated transfer capacity is not released in this case. Without this additional state, the transfer capacity would be released immediately on a failure, even if the path to the peer seems to be working. Thus, the routing would select the same bundle over and over again without any delay which leads to a exhausting processing loop.

A RequeueBundleEvent is handled by a dedicated component as part of the routing component group. It takes care about the exponential back-off between each retry and the maximum number of succeeding retries. If there are too many retries done, the transfer is aborted by raising an TransferAbortedEvent and the transfer capacity is released again.

3.6.7 Multi-stage Connection Set-up

In this architecture, we have several classes of connections. Some of them are available instantly while others need some sort of connection set-up first. Similar to the On-Demand Contacts mentioned in [CBH⁺07], there might be a discovered peer which requires a multi-staged connection set-up to get a valid path for a bundle transmission. For those peers, it is not clear whether a connection will succeed until a set-up is initiated.

For example, the Wi-Fi Direct [Wi-14] or more generic Wi-Fi P2P technology requires such a set-up procedure. By itself, the technology provides only a way to discover peers and to establish IEEE 802.11 connections between two or more devices. Typically, an IP stack is used on top of those connections to send and receive data.

To utilize this kind of connections using a compatible CLA, the bundle transfer will initiate the connection set-up and is then delayed as explained in Section 3.6.6. Once the connection set-up completes, the CLAs and the discovery related

components are notified to adapt to the new environment. If the discovery mechanism then reveals additional contact parameters using the new connection, they are added to the corresponding contact. Any further bundle transfer or retransmission will prefer this new path for subsequent transmissions.

3.7 Fragmentation

Since the potential size of one single bundle is almost arbitrary, it is allowed to split bundles into independent fragments with a portion of the original payload. The standard [CBH⁺07] permits two types of fragmentation. The pro-active approach divides a bundle into smaller pieces before the transmission starts. This may be convenient if the contact volume is known in advance. The reactive fragmentation allows partially transmitted bundles to be transformed into fragments in case of a disruption. This mechanism requires support of the CLA, because it is necessary to know how many bytes are already received by the next-hop. After a disruption, the receiver will subsequently forward the fragment of this bundle to other nodes and the sender may resume the transmission of the bundle with the remaining fragment on the next contact, even to different next-hops.

Not mentioned by the *Bundle Protocol Specification* is that this approach may be harmful in conjunction with multi-copy-routing schemes. But many proposed routing algorithms for DTNs distribute copies of bundles over several paths to increase the delivery probability. As soon as bundles are distributed over multiple paths, fragmentation might happen also multiple times for the same bundle. Without any further policy on that, a bundle may be fragmented on one path into three parts and on the other into two parts. Those sets contain the same payload, but have to be processed as unique bundles because they differ in data offset/length. Thus, the data volume to deliver to the final destination has doubled in this case. An option would be to reassemble the fragments on the path and discard redundant parts of the payload as soon as possible. But that still raises more questions for cases when a reassembling is reasonable and when not.

In this architecture, we propose to restrict fragmentation using a policy to avoid too many copies of the same bundle payload in the network. The most safe approach is the pro-active fragmentation where the generation of fragments is limited to the original sender. Before any bundle leaves the original sender, it gets fragmented using a predicted or configured maximum size. The bundle nodes on the path are not allowed to apply any further pro-active fragmentation. That policy avoids any redundant copies of the bundles due to pro-active fragmentation in the network.

Since contact times are not predictable in any kind, the reactive fragmentation would be very useful to resume a disrupted transmission. This is appropriate at least for terrestrial networks. But some issues have to be considered using this approach. The fragmentation may happen on arbitrary bundle nodes along the path and may lead to arbitrary fragment sizes. Further, it is not defined whether a bundle node should forward the new created fragment or the original bundle to other bundle nodes after a bundle has been fragmented. To forward both is not reasonable.

To mitigate the issues, we propose to limit the reactive fragmentation locally. Thus, if a transmission gets interrupted, the sender node remembers which next-hop has already received a bundle partially, but not generates a bundle fragment explicitly. If a contact to the same node occurs again, the transmission will be resumed by just transmitting the missing fragment. In case of a contact with a different node, the original bundle is transmitted. This approach may not completely eliminate redundant copies in the network, but effectively reduces the number of fragments while allowing interrupted transmissions to be resumed.

3.8 Neighbor Discovery

The discovery of neighbors is based on the IP Neighbor Discovery (IPND) specified in [EAGB12]. Contrary to the draft, this protocol is used here as a generic protocol for any underlying layer supporting multi-cast messages. For IP-based CLAs the protocol sends out UDP multi-cast beacons to announce its own presence to adjacent peers. If the IP-layer is not present, the CLA has to provide an interface to send multi-cast messages to all neighbors. A peer is considered as unavailable if the beacons are absent for a while.

Version	Flags	Beacon Sequence Number
EID Len (*) (optional)		EID String
Service Block (optional)		
NBF Length (*) (optional)		NBF Bloom Filter Bits

(*) marked fields are encoded as SDNV

Figure 3.14: IPND Beacon Message Format

The beacon message format in Figure 3.14 starts with a one byte version field set to 0x02 as proposed in [EAGB12]. As next, one byte of flags indicate the inclusion of optional parts and a two byte sequence number is incremented for each beacon. The EID is optional but strongly recommended and contains the endpoint of the peer which sent the beacon as SDNV encoded length and the EID as string of variable length. Service descriptions are stored in the service block using the structure shown in Figure 3.15 and as last the beacon can contain a Neighborhood Bloom Filter (NBF) to detect bidirectional links.

Number of services (*)	
Service Name Len (*)	Service Name (variable)
Service Param Len (*)	Service Parameters (variable)

(*) marked fields are encoded as SDNV

Figure 3.15: IPND Service Block Format

The service block can contain an almost unlimited number of service pairs consisting of a name and parameters. These are used to announce contact parameters and other service data. Since this version of the draft does not specify how the parameters are encoded, we use an intuitive and human readable string representation. A node with a TCP-CLA bound to the IP address 1.2.3.4 and port 4556 will add the service name `tcpcl` with `ip=1.2.3.4;port=4556`; as parameters. In this case, the parameter `ip` is optional because it is already included in the source address of the IPND beacon. Thus, it is allowed to send only the parameters `port=4556`; in our example.

Additional to CLA contact parameters, each peer may add other service parameters of interest like DTN-DHT parameters (see Section 3.9.1) and DT-NTP service parameters (see Section 5.2.6). Actually, each component may add and retrieve data by registering itself to the `DiscoveryAgent`.

Each discovered peer is represented by a `Node` object which is managed by the Core. These objects hold a merged set of the service entries discovered by all components. Each service entry is marked with an individual life-time. If not refreshed, an entry expires after a time-out and disappears. The assigned life-time is defined by the discovery component which refreshed the service entry as last.

3.9 Extensions

Just like the BP, the bundle node architecture presented here is extensible the same way. This allows developers to add new features without violating existing structures and processes. In this section, we specify nouveau extensions in order to provide new functionality or solve common issues.

3.9.1 DTN-DHT

In practice, DTN nodes are often connected using IP. Even if nodes are not immediately adjacent to each other, it is possible to use manually configured static routes and EID to IP mappings to link nodes and DTN regions through the Internet. Despite this is a quite simple approach, it does not scale well nor provide any mechanism to deploy topology changes dynamically.

As the Internet is ubiquitous and will remain so in the future, we presented an approach in [SLM⁺12] to obtain static routes and resolve EIDs to IP addresses via IP-connected nodes. Instead of discovering nodes using multi-cast beacons as presented in Section 3.8, we exploit the Distributed Hash Table (DHT) of the wide-spread Bit-Torrent (BT) network to store EIDs to IP mappings.

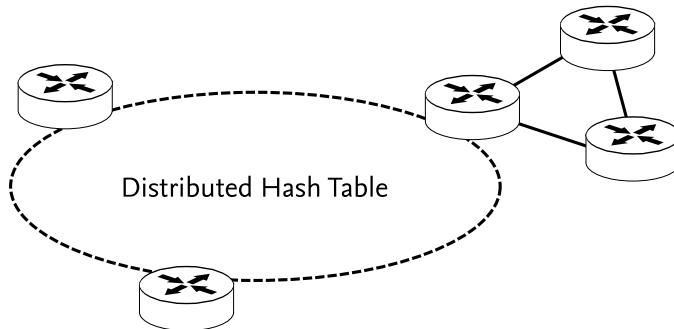


Figure 3.16: DTN-DHT architecture

In Figure 3.16, three nodes are participants of a DHT. One of the nodes is additionally connected to two other nodes forming a DTN region. The routing protocol within this region does not necessarily match the protocol used by the other nodes in the DHT. If nodes of a DTN region are not directly connected to the DHT, a proxy node can act as gateway by forwarding bundles from and to nodes within the DTN region. Nodes connected to the DHT can store and retrieve key-value pairs which are distributed evenly across all participating nodes while still providing

good lookup performance. In general, DHTs are resilient against node failures, have excellent scalability and support a flat name-space.

Without any modification, the BT-DHT is limited to store and retrieve an IP address / port value mapped to SHA-1 hashes, whereby the IP address is always assigned to the address of the publishing node. In order to publish arbitrary EIDs to IP mappings, the EID to publish is hashed and stored together with the UDP port of the local DTN-DHT service. This local service is used to obtain additional data about the found DTN node using a stateless query response protocol based on UDP.

If a node is configured to join the DTN-DHT, it tries to connect to multiple DHT peers. Once connected a node can gradually learn about the structure of the DHT by forwarding messages and populating its peer table. But first, the node has to connect to at least one peer to start from. This process is called bootstrapping.

The simplest approach to start is to configure static peers, but this is hard to maintain if the addresses of the peers are changed too often. A more flexible approach is to query the Domain Name System (DNS) using a pre-deployed address which returns at least two peers of the BT-DHT. To interconnect peers even if there are no Internet services, peers can discover each other using the standard discovery beacons as introduced in Section 3.8. For that purpose, a dedicated service block with the name dhtns is added to the beacons, which will offer the UDP port of the local DHT service. Once a node receives such a service block, it can extract the DHT service address and add the neighbor as DHT peer. As last option to bootstrap, the node may load peers from a file. This approach is typically used to restore known peers on a system reboot.

If a node is connected to the DHT and configured as discoverable, it publishes its own hashed EID together with the local UDP port of the DTN-DHT service. The node has to refresh its entry periodically to prevent it from expiring. If the node acts as a proxy for nodes within a DTN region, it may additionally publish its neighbors. If a node refuses to be announced by a proxy, it can add an opt-out parameter to its discovery beacons to signal this policy to all neighboring proxies.

The DHT is queried if a node generates or receives a bundle and has no valid route to the destination. For that purpose, the destination EID of the bundle is hashed using SHA-1 and a request is sent to the DHT. The responses to that request may contain several potential addresses, since there may exist entries of proxies or a node has moved to another IP address while the old entry is still valid. It is to mention here, that the hash table has no function to remove an entry. Entries are removed implicitly by not refreshing them.

Using the received addresses, the node queries the DTN-DHT services directly to collect more details about the found peers. The details include the exact un-

hashed EID of the peer, which is necessary to distinguish between proxy and direct entries, as well as addresses and ports for different CLAs.

Finally, the gathered data about nodes is announced the same way as in the Neighbor Discovery process (see Section 3.8). Found nodes are added as well as proxies plus a static route to forward data for the requested EID to the proxy. To remove gathered data out of the system, all entries are limited using a life-time. If this expires they are removed and the node will re-query the DHT if necessary.

Since the BT-DHT does not provide any security methods, anybody is allowed to assign a specific EID to its own host. However, the BP does not limit the structure of EIDs and thus it is legal to have multiple nodes with the same address. To provide some kind of trust in the identity of nodes, higher layer mechanisms are necessary. This can be done by using the Bundle Security Protocol Specification (BSP) (see Section 4.12.4) or the TLS Handshake in the TCP Convergence-Layer (TCP-CL) (see Section 3.9.5).

3.9.2 Streaming

The BP proposes to bundle all related data into a single bundle. That directive and the almost arbitrary size of bundles imply a considerable transmission delay while preparing a bundle before sending it. Moreover, it is impossible to store a continuously generated data stream without a predefined end in a single bundle. Thus, it is necessary in some cases to split-up data streams into multiple chunks, forward them successively, and reorder them at the receiver. This approach would effectively reduce the delay between the beginning of a transmission and the first data arriving at the receiver.

In [MPW11], we presented a proof-of-concept demonstration of a DTN streaming system. By extending the BP, the demo application was able to split large and continuous data streams into multiple chunks. These were augmented with metadata for reordering and sent to a single receiver as well as a group of endpoints. The set-up consisted of a surveillance camera, multiple intermediate nodes, and a central monitoring station. As shown in Figure 3.17, the video data was generated by the surveillance camera and transmitted by node A via ad-hoc Wi-Fi to nodes B and C. Since epidemic routing is used in this scenario, both nodes forward the streaming chunks to the node D using their direct Ethernet connection. Finally, node D is in charge to reorder received bundles and deliver them to the video application.

Additional to the basic streaming capability of this application, the fault-tolerance of this system was presented. If one of the Ethernet cables was unplugged, the video streaming was played without interruption. If the second

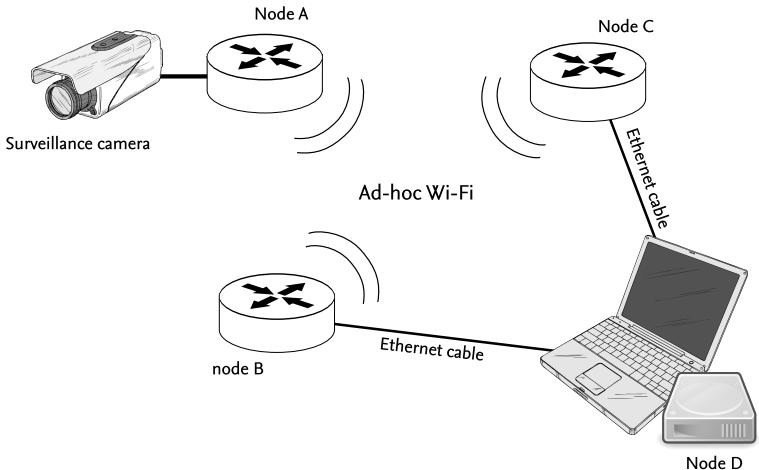


Figure 3.17: Surveillance system with DTN streaming

cable was unplugged too, the video was still playing until the buffer ran empty and then the video freezes. Meanwhile node A continued in producing more chunks and forwarded them to node B and C, so that if one Ethernet cable was plugged in again, the video playback was resumed without any missing frame.

Surveillance monitoring is just one application for DTN streaming. The same experiment was also done with a Digital Video Broadcasting (DVB) stream of public television. In combination with the ability to sent bundles to endpoint groups, an efficient and reliable video broadcasting to consumers would be possible.

The Figure 3.18 shows the extension block with the meta-data for each streaming chunk. The block starts with the experimental block type number 242 and adds streaming flags indicating the first/last chunk of the stream as well as a chunk number. Beside of reordering at the receiver, this block allows bundle nodes on the path to reorder streaming bundles. This would reduce the need of reordering at the receiver and may lead to a better utilization of the available transmission capacities, since nodes on the path can release their resources earlier. Chunks belong to the same stream if the source and destination EID is equal. Support for multiple streams in parallel is realized using an individual application part of the source EID. An additional identifier for each individual stream is not necessary.

The reordering on the receiver side may be done in the bundle node as part of the API or directly in the client. In the latter case, the bundle node does not need to support the streaming extension and the client gets more control over the

Block type	Block processing ctrl flags (*)
	Block length (*)
Streaming Flags (*)	Chunk Number (*)

(*) marked fields are encoded as SDNV

Figure 3.18: Streaming Extension Block Format

stream processing. In any case, the bundle node or the client has to pause the data delivery, if there is a missing chunk. As example, if chunk 1, 2, and 4 is received but 3 is missing, chunk 1 and 2 would be delivered directly but then chunk 4 has to be delayed until chunk 3 is received.

The sender of a stream has to decide how many bytes are stored in a single chunk. The simplest way to do this is to set a fixed amount of bytes. This works pretty fine if the sender produces a byte stream with a constant bitrate, but if the stream has a variable bit-rate this may lead to a long delay due to buffered data waiting until the byte limit is reached. Thus, a more reliable approach is to buffer data for a fixed amount of time to gain a fixed bundle rate where the transmission delay is predictable.

Another and more adaptive approach is to exploit receiver feedback to identify the right time when to flush and release the current chunk. Similar to the Nagle-Algorithm [Nag84], new data is appended to the current chunk as long as the last chunk is not acknowledged by at least one receiver. This procedure would lead to a transmission delay which is equal to the Round-Trip Time (RTT) between the sender and the »nearest« neighbor, where »nearest« is defined as the neighbor with the lowest RTT. If the stream duration is less than the lowest RTT, the whole data is stored into a single chunk. The acknowledgements for this purpose are generated using the status reporting capability of the bundle node. To request acknowledgements, each transmitted bundle is flagged with the bit indicating »Request reporting of bundle delivery«. As result, a receiver will generate a BSR for each delivered chunk and sent it back to the sender. Similar to the reordering approach, the buffering policy may also be implemented as part of the API or directly in the client.

3.9.3 Compression Extension Block

In cases where the delivery delay is less critical than the bundle size, e.g. due to limited contact durations, a data compression algorithm can significantly reduce

the size of bundles. This is particularly efficient if sparsely filled Bloom-filters are transmitted, as done during the routing exchange (Section 3.6.1). This section introduces an extension for bundle payload compression which deflates and inflates the payload transparently to the DTN application.

To deflate the payload of a bundle, an Compression Extension Block (CEB), shown in Figure 3.19, is placed in front of the payload. The block starts with the experimental block type number 202 and contains the used compression algorithm as well as the original payload size, both values are encoded as SDNV. As next, the payload in the Bundle Payload Block (BPB) is deflated in place and the block length of the BPB gets reduced. The inflate process just reverts the previous steps by inflating the payload in place using the compression algorithm defined in the CEB and removing the extension block if the operation was successful.

Block type	Block processing ctrl flags (*)	
	Block length (*)	
Compression Algorithm (*)	Original Payload Size (*)	

(*) marked fields are encoded as SDNV

Figure 3.19: Compressed Payload Extension Block Format

To provide the compression procedure transparently to all clients, it is implemented in the bundle node as part of the API. Since the bundle node can not decide, whether the destination of a bundle supports compressed bundles, the client can set an option which allows the bundle node to compress the payload. It is therefore necessary that the client knows about the capabilities of the receiver. An example where this is possible is the routing exchange of Section 3.6.1. The request-response procedure allows a bundle node to indicate the compression support as part of the request. The response, which is usually magnitudes larger, can then be transmitted compressed since the responding node knows the capabilities of the requesting node.

As part of this extension, only two compression algorithms are defined, but the extension block format is particularly designed to stay extensible for many other algorithms. The fast and very popular zlib algorithm specified in [DG96, Deu96a, Deu96b] is assigned to value 1. It is the default algorithm and should be considered as mandatory to all implementations. Value 2 is assigned to the bzip2 algorithm [Sew14] which combines several techniques to get a better compression rate as zlib but needs significantly more processing time.

Since fragmentation very often introduces restrictions, there are some limitations to consider if a bundle node handles CEBs while there are nodes with fragmentation support in the network. Splitting a compressed bundle into fragments is generally permitted, but the compression should then happen at the origin sender only to avoid any uncompressed copy of the same bundle in the network. If there would exist two copies of a bundle in the network, one compressed and the other not, it may happen that fragments of both variants arrive at the receiver. These fragments cannot be distinguished and if they are reassembled, the bundle payload would be corrupt. The same happens if compression on the bundles path would be allowed. Bundles distributed over multiple paths may get fragmented and some of them arrive compressed at the receiver while others arrive uncompressed. To avoid such an issue, it is recommended to never apply compression on a bundle fragment. Moreover, partial payload as part of a fragment can not be inflated. Thus, it is necessary to reassemble fragments first and then inflate the payload.

3.9.4 Tracking Extension Block

For administrative purposes and network monitoring it is sometimes important to track the path a bundles has been traveled. The *Bundle Protocol Specification* defines several BSRs which are generated when a bundle was received, forwarded, delivered, deleted, or accepted for custody, but those signals do not expose the edges of a path a bundle has taken.

Block type	Block processing ctrl flags (*)
	Block length (*)
	Number of items (*)
Entry Flags (*)	DTN time (optional)
EID Len (*)	EID String
Entry Flags (*)	DTN time (optional)
EID Len (*)	EID String
...	
Entry Flags (*)	DTN time (optional)
EID Len (*)	EID String

(*) marked fields are encoded as SDNV

Figure 3.20: Bundle Tracking Extension Block Format

Similar to the Record Route Option in IP [Pos81b], the extension presented here adds a Tracking Extension Block (TEB) to a bundle which gets complemented by each bundle node on the path. The block format is shown in Figure 3.20 and starts with the experimental block type number 193. To ensure that this block is copied to every bundle fragment, the bit 0 is set in the block processing ctrl flags. The Block body data contains a list of hops starting with the number of items encoded as SDNV. Each entry represents one hop including flags indicating inclusion of optional fields, an optional DTN time, and the EID of the passed hop. The DTN time definition is taken from the RFC 5050 and consists of two SDNVs (seconds and nanoseconds). If a bundle node receives a bundle with an TEB attached, it should increment the Number of items by one and append a new entry with the DTN time set to the time when the bundle was received at this hop. Since the block is altered on each hop, it can not get protected using the Extension Security Block (ESB) defined in the *Bundle Security Protocol Specification* [SFWL11].

3.9.5 Security Extensions for the TCP Convergence-Layer

The *Delay-Tolerant Networking TCP Convergence-Layer Protocol* presented in Section 4.10.3 is based on the reliable and connection oriented TCP protocol. In this section, we address vulnerabilities caused by missing security enhancements for authentication and encryption. Further, we will discuss possible attack scenarios and finally present security extensions to prevent those attacks.

Attack Scenarios

Although the *Bundle Security Protocol Specification* already defines a range of security features (see Section 4.12.4), there are still serious attack scenarios which are not covered. The following description of the attack scenarios assume that the attacker did not have hijacked any trusted node in the network. That kind of attacks are addressed by misbehavior detection systems [ZDG⁺14] and not part of this work.

Black-hole During the handshake between two peers, the attacker can pretend any EID, since there does not exist any authentication mechanism to verify that information. If the attacker has knowledge about the routing process of the bundle node and the destinations of the bundles in the storage of the victim, he can selectively choose an EID for himself, so that he gets passed as many of these bundles. Whether the attacker forwards the bundles can not be verified by the victim. The discard of bundles is called black-hole attack [HPJ02, RFdAGo8].

Denial-of-Service An attacker can initiate any number of connections to a node by using a counterfeit EID. Under certain circumstances, the bundle node of the victim allows only one connection per IP address. In this case the attacker has the

possibility to use several computers or can fake the IP address, if it is located at a point in the network where he can wiretap traffic and inject packets for other hosts. Alternatively, he can use proxy servers to get access to different IPs addresses. However, in cases of Network Address Translation (NAT) a node must accept multiple connections from a single IP address.

Once connected, the attacker can start to send very large bundles to the node. The attack is facilitated by the fact that the node does not know how large the bundle will be until the last block is received. The attacker can continuously send data segments without ever setting the flag for the last block. If the storage of the victim reaches a certain capacity, the bundle node is forced to reject bundles or even to terminate connections. However, since it can not distinguish between genuine nodes and an attacker, this can lead to rejection of legitimate bundles.

Man-in-the-Middle The highest potential for an attack provides the position between two communicating nodes. There exists several sophisticated techniques to intercept the communication of two peers. For example, an attacker can use Address Resolution Protocol (ARP) spoofing, if it is located together with one of the nodes on the same network segment, or establishes to both nodes a separate TCP connection and forwards the packets transparently between them. For this purpose, it can interrupt an existing connection between the nodes by performing a Denial-of-Service attack. In case that the nodes can verify EIDs by checking the source IP addresses, it may be necessary for the attacker to fake the sender's IP address and ensure by means of source routing that the replies return to him.

Once an attacker is in this position, he may discard any bundles and fake associated acknowledgments (black-hole attack). Additionally, he can generate shutdown messages with an almost infinite delay to disable any further communication between two nodes.

Transport Layer Security

The attacks presented before are feasible because none of the bundle nodes has proven its own identity to the other. But even if the initial exchange of the Contact Header would be authenticated, man-in-the-middle attacks would still work and a bundle node can not be sure that bundles have been acknowledged by the previously authenticated peer. Thus, it is necessary to authenticate each data segment transmitted between the bundle nodes.

TLS is a certificate based approach to provide security over the Internet. It uses X.509 certificates and hence asymmetric cryptography to authenticate servers and in some cases even clients. Once a connection is established, each data segment is authenticated using a Message Authentication Code (MAC) and optionally en-

crypted. If the TCP communication would be secured by TLS with mutual authentication, the mentioned attacks would not work because none of the peers can lie about its own identity and all traffic is cryptographically associated with the authenticated peer. For the security extension proposed in this section, we simply extend the TCP Convergence-Layer Protocol with TLS and initiate a TLS handshake with mutual authentication, if both peers indicate their support for this extension.

In order to do that, we utilize the last of the 4 unused bits in the contact header flags (Figure 4.2) to indicate the support of this extension. Once both peers are indicating their TLS support, they switch into the TLS mode and initiate an extended handshake to exchange their certificates. The certificates are verified by each bundle node against a known certificate authority, which is authorized to approve trusted peers and their EIDs. By offering the TLS extension as optional, the extension does not break any compatibility with the origin protocol and existing implementations.

The extension proposed here is based on certificates and thus has two requirements. First, the trusted nodes in the network have to be secure. If an attacker is able to overtake a trusted node, it can perform almost all the attacks mentioned before. In those cases, only a misbehavior detection system [ZDG⁺14] would be able to identify bad nodes and revoke their certificate. Second, certificates require some sort of key management. Trusted certificates of authorities must be distributed to the nodes and an approach is necessary to revoke certificates of compromised nodes. Furthermore, the validity of certificates is limited using time-stamps and they expire after some time. Thus, an approach for time-synchronization is necessary to verify if a certificate is valid or not.

4 Implementation

This chapter discusses details of the desired bundle node implementation based on the architecture presented in Chapter 3. We start with a summary of the objectives and as result we introduce a concept how the extensible structure and other proposed characteristics can be realized. Further, we explain performance relevant mechanisms and what we have to consider during the development of the software.

As next, the software structure of the implementation, named **IBR-DTN**, is presented. The name is a concatenation of the name of the Institute we are working for and simply DTN. In the following sections, details on the realization of concurrency, the data serialization, and the different components are presented. Bundled applications are presented in Section 4.11 and integrated protocol extensions are explained in Section 4.12. Finally, we talk about the challenges mastered during this work. In that section, we also explain the limitations we applied to gain a robust implementation.

4.1 Objectives

The implementation presented in this thesis has its roots in research on DTN related algorithms and applications. Thus our major goal is to keep the implementation open and extensible to make further research as easy as possible. Moreover, we chose to follow the *Delay-Tolerant Networking Architecture* [CBH⁺07] and the *Bundle Protocol Specification* [SB07] to keep the implementation inter-operable with existing implementations. The architecture discussed in Chapter 3 already follows these specifications. An additional objective of this work is to keep the memory footprint low to support even embedded systems with low resources. To be precise, we want to support hardware with at least 32 MB RAM and a processor capable in running at least a Linux kernel. Within the context of this work, we propose this as minimal specification for an *embedded system*. This allows us to target platforms like the Intel imote2 [Int06], Arduino Yún [Ard14], or even the hackable Wi-Fi SD card [Tra14] manufactured by Transcend.

Moreover, the bundle node implementation presented here should also be capable to utilize the available resources of more powerful systems. In general, all embedded as well as standard systems based on Linux belong to the primary target. Additional to those, we strive to achieve a high portability and design the software

as platform-independent as possible. Thus, we avoid external libraries if feasible and add an abstraction layer to important system calls. This way, it is possible to add support for Microsoft® Windows®, OS/X® as well as the Android™-platform and cover the most popular devices out there. However, we do not address micro-controllers like the ATMEL® ATMEGA328 and platforms based on it. Due to the very limited resources and the special needs of those nodes, the feasibility to realize a bundle node on such a platform is limited. uDTN [vZBPW12] is an implementation which focuses on those platforms and realizes a highly optimized but limited bundle node for sensor node applications.

As non-functional requirement, the system should be fault tolerant against protocol errors as well as bad behavior of other bundle nodes. Furthermore, the integrity and stability should not be affected by other connected bundle nodes. By introducing a strict modularization of the bundle node, it is possible to protect a single module against misbehavior of other modules. That way, a newly added module does not affect the existing modules if they are not explicitly designed to and this can avoid misleading behavior while testing experimental algorithms or applications.

To fit the needs of embedded systems with low resources, it is possible to exclude specific components or not needed parts with heavy-weighted dependencies at compile time. The result will be a more lightweight and specialized version of the bundle node which needs less resources but still suites the proposed application. If modules are not disabled during compilation, the bundle node should be configurable to enable or disable modules at runtime. Modules realizing specific features are only instantiated if they are explicitly enabled. Default modules may be disabled the same way. Even if disabling modules at run-time do not reduce the size of the binary, the resource requirements of the bundle node can be optimized for a specific use-case.

4.2 Concept

As discussed in Section 4.1, the strict modularization offers many advantages such as stability and robustness against failures. In this section, we like to explain the extensible structure of the bundle node presented in this work.

The bundle node is partitioned into components. Each of them fulfills a dedicated task and reacts to events raised by the event-system introduced in Section 3.4. All components implement methods to initialize and shutdown the component. In general, there exist two sorts of components. Integrated components are passive and only react to incoming calls. In case of an event, the processing is handled in a thread of the event-system. A long running operation would block

the event-system until the call is finished. To avoid such blocking procedures, independent components are introduced. Additional to the standard operations to initialize and shutdown, independent components have a run method which is handled by a dedicated thread. In this thread, the component can execute long running operations without affecting other components.

As mentioned in Section 3.2, the Core component group is the central hub within the bundle node. It contains the event distribution service which accepts and forwards events to other components using multiple dispatcher threads. Further, it manages all convergence-layers and decides which one is used to forward a bundle to a neighbor. The decision is made using the data gathered by the Discovery Agent which is also working as part of the core. This component provides abstract procedures for beaconing the local presence to other bundle nodes as well as the capability to process beacons. Generated beacons are forwarded to convergence-layers and other components capable in sending and receiving them. The WallClock periodically generates events for elapsed time with a resolution of one second. These events allow integrated components to react to elapsed time without the need for an additional thread within the component. The last purpose of the Core is the validation of bundles which is applied while a bundle is received from another peer or loaded from a persistent file-systems. This way a bundle can be rejected without the need to receive the potentially huge bundle if the size, life-time, or another property violates the configured policy.

Different to the components of the Core, components to store bundles are designed to be interchangeable. This is due to the fact that storage approaches largely depend on system features. Moreover, there exists a trade-off between high performance and low resource footprint. A bundle node gains its best performance if bundles are stored in fast but volatile memory. Additional caching strategies may increase the performance further. Such a configuration is applicable if the underlying system is reliable, has enough resources and a shutdown is unlikely respectively it is not necessary to recover bundles or state over reboots. If this is not the case, then a system maintainer should prefer to store bundles on disk instead of in volatile memory.

In addition to the option to use persistent storage or not, the user can decide to keep the working copy of the bundle payload on disk or in memory. The abstraction layer to realize that was already introduced in Section 3.3 and uses BLOB entities as generic data container. In any case, incoming payload of bundles is stored temporarily in those entities until the complete bundle with all its associated BLOBs is handed over to the storage component. By choosing to process working copies in memory instead of on disk, the bundle node consumes more resources, but achieves a higher throughput performance in return. To keep the

bundle node as flexible as possible there exist several interchangeable implementations for different storage approaches. These are instantiated during run-time dependent on the specific configuration.

Since routing in DTNs is still a hot research topic, the component group for routing is structured as flexible as possible. Several extensions collaborate to realize a single routing algorithm or a combination of different approaches. The standard set of extensions consists of a database for peer related state and a routing exchange handler which stores the result of the exchange in the peer database. Beside those basic parts, the routing decision engines are also encapsulated into extensions. Those realize direct delivery, static routing, flooding, epidemic, and PROPHET routing. All of them iterate over the storage to seek for bundles to forward, each time the neighbor topology changes. The set of extensions to use also depends on the configuration of the bundle node.

Another generalized approach is the discovery of neighbors using heterogeneous technologies. For adapting layers, we differ between those are capable in sending and receiving broadcast beacons and layers which do a technology specific discovery by itself. Examples for the latter case are the Wi-Fi P2P technology, Bluetooth, or the DTN-DHT approach mentioned in Section 3.9.1. All these announces found or disappeared peers to the bundle node. If the adapting technology supports broadcast datagrams, the corresponding layer just needs to send the beacons generated by the *DiscoveryAgent* component and pass back incoming beacons. The processing of beacons and monitoring of discovered peers is managed in a centralized component.

The convergence-layers are used to send and receive bundles using an underlay technology. Further they apply pro-active bundle fragmentation if necessary and reactive fragmentation if supported. By adapting an underlay technology for bundle transfers, this type of component does something similar to the discovery adaption layers. In fact, it makes sense to implement the discovery adaption layer as part of the convergence-layer, if the discovery approach is not shared among several convergence-layers. Many convergence-layers can be excluded during compilation to gain a more lightweight binary. If a convergence-layer is included, they can be attached and detached to the *ConnectionManager* during run-time dependent on the bundle node configuration.

The last type of components to mention here are integrated endpoints. These are in fact tiny applications embedded in the structure of the bundle node which send and receive bundles. They may interact with internal components to realize interaction between bundle nodes. The most well-known example for such an endpoint is the echo application. For debugging purposes, a bundle can be sent to this endpoint and it will get reflected back to its source. This way the RTT

to another bundle node can be estimated. Besides the echo endpoint, the bundle node creates integrated endpoints for the routing exchange, time-synchronization, and bundle-in-bundle encapsulation.

Each component is attached to a specific run-level. These are mainly used to re-configure the bundle node dynamically during runtime or to disable components to reduce the energy consumption. The existing run-levels are core, storage, routing, api, network, and routing extensions. Each of them depends on the preceding run-level, e.g. storage depends on core and api depends on routing. The core run-level sets up the basic configuration, logging facilities, and the node's identity. As central component, the event distribution is initialized to route events between all modules. The connection manager, the wall clock, and the discovery agent are also instantiated in this run-level. The configuration of storage components is applied in the run-level storage. The specific implementation is instantiated and assigned to the Core to make it globally available. Additional features like persistent bundle-sets or a custom bundle-index for scheduling is assigned here too. The routing run-level initializes the base components for routing. It processes received or generated bundles, put them into the storage, filter duplicates, and applies security checks. The application programming interface as well as integrated applications are added in run-level api. They require access to the bundle storage to receive bundles, but does not necessarily need a complete networking stack as long as they are allowed to create and queue new bundles. To reassemble received fragments and deliver them to applications, the FragmentManager is instantiated and integrated applications to reply to echo requests, process bundle-in-bundle encapsulation [SDSoI], and perform time-synchronization (see Section 5.2) are added by default. Finally, the ApiServer is added to allow external applications to send/received bundles and provides access to bundle node management functions. The run-level network initializes components responsible for discovery and interconnection between bundle nodes. To those belong all convergence-layers, the IPNDAgent component which is responsible for discovering neighbors using IP, the Wi-Fi P2P component, and the DTN-DHT component mentioned in Section 3.9.1. Moreover, static peers get announced and static routes are added to the static routing component. The routing extensions is the highest run-level and initializes specific extensions for routing bundles between bundle nodes. Each extension implements a single algorithm to decide if a bundle should get forwarded to a node or not. In general, extensions are combinable with each other but select and queue bundles independently for forwarding. The set of bundles being routed is the union of all bundles routed by all extensions. Additional to routing extensions for specific algorithms like flooding, epidemic or PRoPHET, there exist some dedicated extensions for standard operations. One of those extensions is the

retransmission extension which re-queues a bundle if a previously queued transmission failed due to a temporary failure. For that purpose, the extension keeps track of the number of retries made for a bundle and delays the re-queuing of it using an exponential back-off. Another standard extension is the static routing. If a static route is configured or added during run-time, the extension will match its routes against the bundles in the storage. The routing exchange approach explained in Section 3.6.1 is also realized as routing extension. This extension is available to all other extensions, provides access to routing data, and initiates the exchange if some required data is not available.

4.3 Performance Considerations

An efficient system design presupposes thin algorithms and a lean approach for data handling. Especially if data units can be of arbitrary size, as with bundles, an efficient approach to handle the potentially large amount of data is very important to achieve a good throughput performance. The pursued approach here is to store the payload in BLOBs and reflect the bundle block structure using objects. Further, if the complete structure or direct access to blocks is no longer necessary, we move bundles into the storage and use references like the entities `BundleID` and `MetaBundle` instead. References have a smaller memory footprint and require less processing.

4.3.1 Data Processing and Policy Checking

Handling data efficiently is not the only requirement to gain a good performance. In fact, even simple checks during the processing may reduce the achievable throughput. To explain where this is relevant, we will take a closer look at the transmission procedures.

The reception procedure for bundles starts in the convergence-layers where bundles are typically received as a set of frames. One way to accept those data for processing would be to store each bundle into a file or buffer. Once the complete bundle is received, it can be parsed and processed. This approach is easy to implement but also inefficient, because the CPU will idle during the data reception. Assuming that the processing of bundle data is much faster than the transmission speed, the processing time during the reception is wasted. Especially when dealing with large bundles, a more efficient way would be to assemble and parse a bundle while receiving it frame by frame. This way the combination of transmission and processing time can be shortened. Beside the more efficiently utilized processing time, early parsing allows the bundle node to reject the bundle as early as possible and the remaining transmission capacity is available for the next transmissions.

As result of the frame processing the received data is translated into processable structures. For that purpose, the primary bundle block and each extension block is stored in objects and attached to another object which represents the whole bundle. During parsing of incoming frames, the bundle node applies system policy checks. This way it is possible to reject already known bundles or those with unacceptable parameters such as large life-times or too big payload sizes as soon as possible. If those checks are not realized efficiently, they may slow down the whole transmission and the available link capacity is not fully utilized. Once a bundle is received completely, it is added to a set of known bundles and finally stored in the storage component. Due to the concurrent processing of incoming bundles, there is a chance that more than one copy of the same bundle passes the duplicate check during the reception. For that case, a second duplicate check after the reception of the complete bundle is necessary. If that check identifies a duplicate, the bundle will be dropped and processed no further.

The duplicate check is one of the most important checks in a DTN. Since there is no way to prevent loops another way, each bundle has to be checked for duplication using a list or a more efficient data structure. To realize a fast check, we use a Bloom-filter as first stage check and only if the filter results in a positive match, a more intensive search is done using a set of known bundles. This way the false-positives property of Bloom-filters is not relevant here. To keep the Bloom-filter in sync with the set, it has to be rebuilt every time a bundle is removed. Since it is not critical to detect a bundle as duplicate longer than its life-time, the rebuild operation is executed deferred so that multiple bundles are removed at once.

Despite the duplication checking, extended policies are applied depending on the configuration. The basic limits are checking each block for a maximum block size for foreign as well as locally received bundles, limit the life-time of bundles, and restrict predicated creation time-stamps. Further, there are security checks to reject non-encrypted bundles or those without a trusted signature or authentication. At last, there is an option to reject bundles not addressed directly to the local bundle node.

Before a received bundle is finally stored, it has to pass a procedure which may alter blocks of the bundle. The specification of the *Bundle Protocol* allows extensions with versatile extension blocks. An example of such an extension is presented in Section 3.9.4. Beside the Tracking extension block, there also exists an extension, presented in Section 4.12.3, to limit the number of hops a bundle is allowed to take. Those blocks are altered right after all reception checks and before the bundle is stored. This way all outgoing bundles consistently contain the same data.

4.3.2 Serialization and Parsing

Parsing bundles and preparing them for transmission is more complex as handling data structures of classical networking protocol such as IP. The data format defined by the *Bundle Protocol Specification* utilizes very dynamic data structures and the variable lengths of many data-types prevents a selective access to data required for routing. A basic data-type is the SDNV which represents a number of an unlimited size encoded in at least one byte. The most significant bit of the current byte indicates if the next byte belongs to the current value or not. For practical and efficiency reasons, the size of such a number is limited to 64-bit which is encoded in a maximum of 10 bytes.

Beside SDNVs, which are also used to encode processing flags, there are exist fields containing strings with a variable length. Those strings are primarily used to encode EID or parts of it. Basically, there are two ways to encode strings. One is the *Dictionary* which concatenates strings separated by a zero byte. The offsets of these strings within the *Dictionary* are referenced by a pair of SDNVs for each EID in the primary bundle block. Another way to encode a single string is to use a leading length encoded as SDNV which is followed by raw bytes containing the strings data.

Dynamic data structures make it necessary to parse and interpret almost each single byte instead of directly accessing the n -th bit or byte to apply checks or to match patterns using data masks. This makes an implementation more complicated than it is the case for protocols like IP. However, the benefit is that encoded data is in average much smaller than it would be if only fixed length types are used. If we assume once more that processing capacity is not limited compared to the available link bandwidth, it is a reasonable choice to encode bundles this way.

4.3.3 Routing

In DTNs, we have to route in time and space. Thus, routing decisions may change for each individual bundle every time the network topology is changed. Since this might happen frequently and is hard to predict in opportunistic networks, the bundle node does not queue bundles for transmission once they are received. Instead, the routing components iterate over all bundles in the storage and decide for each of them, if it should be forwarded to an available peer or not. This procedure is stopped once all bundles have been considered or the maximum number of concurrently queued bundles is reached.

Instead of actually placing bundles in a queue, some transfer capacity (introduced in Section 3.6.2) associated to a specific peer is acquired for forwarding procedure. Once the transfer is aborted or completed, the previously acquired transfer

capacity is released again and the routing components are triggered to search again for more bundles to forward. This flow control mechanism prevents the bundle node to queue too many bundles at once and allows even recently received bundles to overtake others with a lower priority.

The queuing decision of the routing components is made based on the EID of a specific peer. Which underlying technology is used to transfer the bundle to the peer, is not relevant here. If a bundle is queued for a specific peer, a connection manager is in charge of selecting the most appropriate convergence-layer to use.

As discussed in Section 3.1.2, the achievable throughput heavily depends on the iteration performance of the storage. Further, the routing decisions might be arbitrary complex but should be quite simple for most of the cases. As example, the epidemic routing decision is shown in Algorithm 1. The function is called for each bundle in the storage and returns True, if a bundle should be forwarded. It checks first, if the hop limit of a bundle is exhausted. Then it excludes all bundles which are addressed to the local node. As next, it skips all bundles addressed to a singleton EID and those with a hop limit less or equal than 1, because they can not be delivered by the peer, if the peer is not the final destination. The next condition exclude bundles handled by the extension for direct delivery. Since the epidemic routing extension assumes to work cooperatively with that extension, bundles addressed to the considered peer are skipped. A special check done by the epidemic routing is on line 14. It checks if the destination of the bundle is also available as neighbor. In that case, the bundle is not forwarded to any other peer, because it assumes that direct delivery is better suited for that case and thus avoids redundant multi-hop transmissions. Since this exclusion is problematic in some cases, this check can be disabled using a configuration option. Finally, the routing extension checks if the bundle is already part of the summary vector of the peer. Since this check is more complex than the others, it is done at last.

Other routing extensions search bundles to forward using the same principle as shown for epidemic routing. The bundles are considered over and over again on every topology change. If a new bundle is received the extensions are directly asked to figure out if the bundle might be forwarded to a peer without an iteration. This shortens the transmission delay in case of a continuous connected network. If the bundle could not be forwarded directly or if all transfer slots are occupied, the bundle is only considered again when the iteration reaches the bundle.

Since the iteration always starts at the beginning and searches as long as enough bundles are found or all bundles are considered, the order of the iteration determines the order in which bundles are transmitted. Thus, in general the storage is in charge to order the bundles as specified by the priorities defined in the *Bundle Protocol Specification*. For customized ordering and experimental purposes, a bun-

Algorithm 1 Epidemic Routing Decision

```

1: function SHOULDFORWARD(bundle, peer)
2:   if bundle.hopcount = 0 then
3:     return False
4:   end if
5:   if bundle.destination is singleton and bundle.destination is local
then
6:     return False
7:   end if
8:   if bundle.destination is singleton and bundle.hopcount <= 1 then
9:     return False
10:  end if
11:  if bundle.destination = peer then
12:    return False
13:  end if
14:  if bundle.destination is neighbor then
15:    return False
16:  end if
17:  if bundle is known by peer then
18:    return False
19:  end if
20:  return True
21: end function

```

dle index interface allows us to implement a custom order for all bundles in the storage. If such an implementation is configured, all iterating components will use that one instead of the default ordering provided by the storage component.

4.3.4 Application Delivery

As soon as an application initiates an API session and defines its EID and group registrations, the bundle node has to select those bundles in the storage which match the registration and deliver them to the application. Further, all newly received bundles must also be considered for delivery. Since the applications can change their EID and group registrations during an active session, it is necessary to repeat the consideration of all bundles in the storage at least on every change of the registration. For that reason, we use the same seek principle as of the routing components to deliver bundles to applications. The advantage of this approach is

that priorities for bundle delivery are automatically applied without any additional effort. To prevent local loop-back of sent bundles and double deliveries, each application has its own bundle-set to remember all already delivered bundles.

Before a bundle is delivered to an application, some additional processing is required depending on the extensions applied to each bundle. If the bundle is encrypted using the BSP, the bundle node tries to decrypt it before the payload is transferred to the application. Another extension to process on delivery is the CEB introduced in Section 3.9.3. The payload of such a bundle is inflated directly on delivery. Processing those blocks right before delivering it to the application has several advantages. Payload intended to be encrypted is never stored unencrypted in the storage. Even if the storage medium gets disclosed in some way, the data is kept secure. Furthermore, compressed payload in the storage saves space. A more practical reason is that it is hard to decide when this processing should be done in advance. The bundle node would have to decide which bundles are a candidate for local delivery. But applications might connect occasionally and the set of bundles for local delivery is not predictable under such a versatile environment.

Another necessary processing of bundles before they can be delivered to applications, is the reassembly of fragments. Different to the processing of specific extensions of a bundle, the fragments must be reassembled first before an application can request it for delivery. For the same reason as with processing of the CEB and the BSP, it is impossible to predict for all kinds of bundles whether applications will ever be interested in them. Thus, the reassembly of fragments is limited to the obvious case and applied only to bundles directly addressed to the primary EID of the local bundle node or to non-singleton endpoints. Once all necessary fragments are available they get reassembled and the complete bundle is announced as if it was received by another peer. Finally, all fragments are purged from the storage.

4.4 Software Structure

To guarantee a good portability, there are two things to take into account. First, avoid external libraries for the basic feature set and second, at least a minimal version of the software should compile without any additional libraries. If that is not possible, it is a good approach to rely on widely ported libraries or write an own abstraction for basic platform-dependent features.

As shown in Figure 4.1, IBR-DTN is organized in several packages: two libraries named `ibrcommon` and `ibrdtn`, a daemon package which contains the bundle node, and a tools package.

The library `ibrcommon` contains an abstraction of primitives for handling files, multi-threading and synchronization, network communication, link mon-

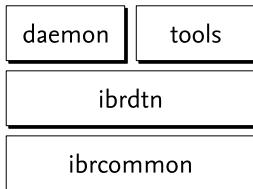


Figure 4.1: Package structure

itoring, cryptographic algorithms, logging, the BLOB abstraction mentioned in Section 3.3, and some minor utility classes. The feature set of this library is not specific to DTNs and encapsulates platform-specific implementations into system-independent object-oriented classes. This way the software built on top of this library does not need to implement any platform-specific code and is portable to all platforms supported by `ibrcommon`.

The second library is based on the `ibrcommon` library and implements DTN related data structures plus common algorithms for the bundle node as well as external applications. To handle the entities discussed in Section 3.3, classes are included to allow parsing, processing and serialization of the data-structures defined by the *Bundle Protocol Specification* and the *Bundle Security Protocol Specification*. Entities not part of this library are BLOBs, which are already included in `ibrcommon`, and entities like the Node and Registration classes which are specific for bundle node operations. The library also contains generalized classes for bundle transmissions using streams. The protocol of this approach is derived from the TCP-CL [DOP14] and is used for the implementation of the TCP-CL as well as for API connections between external applications and the bundle node. Additional API classes complete the set of functionality and allow external applications to connect to a bundle node using sockets.

Beside the libraries, there exists two packages with executables. The `daemon` package contains the binary which implements the full-featured bundle node including all previously presented components. Since the data abstraction is already encapsulated in the `ibrdttn` library, the code contained in the `daemon` defines the algorithms and procedures necessary to act as bundle node.

The `tools` package contains several command-line applications to maintain a bundle node or to set-up scripted applications. For diagnostic purposes, applications are included to measure the RTT or to track the path a bundle has taken. Additional applications are described in Section 2.1.

4.5 Concurrency

Although multi-processing brings in a better utilization of available resources, the down-side is the complex handling of concurrent access to data structures. If there is not enough locking, the software will just burst and if there is too much of it, the performance will be decreased. For that reason, each component and every shared object in this architecture must be thread-safe at least for public methods.

As explained in Section 3.5, the thread-safety is realized using mutexes, conditionals, and semaphores. To apply these patterns in platform-independent code, dedicated classes are implemented as part of the `ibrcommon` library to provide access to native threading capabilities of the system. Although the recent C++11 standard already provides a good abstraction for threading related functionality, the common abstraction done in the library is required to stay portable, since the C++11 standard is not yet supported on every platform.

4.5.1 Mutex

Mutexes are a fundamental mechanism for data sharing among multiple threads. They provide methods to enter a critical region and to leave it after some operations are done. While a thread is working within a critical region, the mutex guarantees that no other thread enters the same region at the same time. The class `Mutex` provides the abstraction of this mechanism. If such an object is instantiated, a system mutex is allocated and finally released if the object is destroyed. Further, it provides the methods `enter()`, `leave()`, and `trylock()`. Compared to the `enter()` method, the latter method does not block until the region is ready to enter. Instead it tries to enter the exclusive region and throws an exception, if that was not possible. A mutex can behave in three different ways. Besides the standard behavior, by blocking any further call of `enter`, the mutex may allow a single thread to enter a mutex more than once, e.g. for recursive functions. As third behavior, the mutex might be configured to do error checking. This way a second enter of the same thread returns with an error instead of leading to a dead-lock.

4.5.2 MutexLock

A frequently done mistake while working with mutexes is a missing call of `leave()`. Especially when dealing with exceptions or goto jumps, it is very likely that such a mistake slips in. To make a mutex more foolproof, a guard can be used to enter and leave the mutex. For that purpose a `MutexLock` is instantiated at the beginning of the critical region. The constructor of the guard calls `enter()` of a

given `Mutex` and `leave()` once the object is destroyed. This way it is guaranteed that every mutex is left as soon as the scope of the `MutexLock` declaration is left.

4.5.3 Conditional

Beside sharing data among multiple threads, it is necessary to synchronize threads waiting for input or state of another thread. Attempts to construct thread synchronization using mutexes only would lead in most of the cases to busy waiting, which is a waste of resources. Algorithm 2 shows such an example of thread synchronization using mutexes. Here, thread 1 sets a to a new value and thread 2 has to wait until a is set to something other than 0. Since it is not guaranteed that line 2 of thread 1 is processed before line 3 of thread 2, there is a chance that thread 2 enters the mutex as first. Then it needs to check, if the variable is already altered. If not, the thread have to leave the mutex to allow thread 1 to enter it. But even after thread 2 has left the mutex, it is also not guaranteed that thread 1 gets the focus. As result, thread 2 loops over and over again checking the variable.

Algorithm 2 Thread synchronization without condition variable

<pre> 1: function THREAD 1 2: Enter mutex 3: $a \leftarrow 1$ 4: Leave mutex 5: end function</pre>	<pre> 1: function THREAD 2 2: loop 3: Enter mutex 4: if $a \neq 0$ then 5: Process a 6: end if 7: Leave mutex 8: end loop 9: end function</pre>
---	--

To get rid of busy waiting in this example, condition variables are used. They provide a *wait* and a *signal* primitive to synchronize threads. Algorithm 3 solves the previous issue by using a condition variable. In this case, thread 1 enters the mutex directly and then check for its condition. If this is not satisfied, the thread calls the *wait* primitive which allows other threads to enter the mutex while thread 2 is paused until a *signal* is called. Thread 1 is now allowed to enter the mutex, alters variable a , and set a condition signal. Once thread 1 has left the mutex again, thread 2 gains back the exclusive access to the mutex and can check its condition again.

Algorithm 3 Thread synchronization with condition variable

1: function THREAD 1 2: Enter mutex 3: $a \leftarrow 1$ 4: Signal condition 5: Leave mutex 6: end function	1: function THREAD 2 2: Enter mutex 3: while $a = 0$ do 4: Wait condition 5: end while 6: Process a 7: Leave mutex 8: end function
---	--

To use the condition variable pattern the class `Conditional` exists in `ibrcommon`. Since a condition variable always needs to be combined with a mutex, the class is derived from `Mutex` and thus is a candidate for the `MutexLock` guard. A `Conditional` object can deliver a signal to a single or to all waiting threads. The single thread signaling is useful for worker approaches, where a new incoming object is delegated to one of many worker threads for processing. To complete the primitives, the `Conditional` also provides a `wait()` method which accepts a time-out value. If specified, `wait()` will throw an exception if the time-out expires before a signal was received. Additionally, there exists an abort-method which unblocks existing and succeeding `wait()` calls by throwing an exception.

4.5.4 Semaphore

Semaphores are a generalization of mutexes. Instead of allowing just one thread to enter a region, it can be entered n times. Most of the time, semaphores are used to manage countable resources. For that purpose the two primitives `post` and `wait` are provided. A `Semaphore` object is initialized using an initial resource counter. This counter represents the available resources and is incremented by one every time `post` is called. If a thread calls `wait`, the counter gets decremented by one if it is larger than zero. If not, the method will block the call until another thread releases a resource unit by calling `post`.

4.5.5 Thread

Threads allow code to run in parallel. The common approach is to start a specific function from the main routine of the program. To differentiate between several instances, each thread gets an individual pointer to some data. Threads can re-

ceive signals and share data with other threads. In general, there exist two types of threads. Detached threads act completely independently and must clean-up their own data structures once they are terminated. Joinable threads are associated to a caller and provide a result on termination. The thread data structures are not cleared until the result is received by another thread.

The classes `JoinableThread` and `DetachedThread` are both derived from the abstract class `Thread` and extend derived classes with threading ability. Since these are abstract classes, a derived class must implement at least two methods. In the `run()` method the class has to implement its own main routine. This method is called once the thread is started. If the method exits, the thread gets terminated. To terminate a thread while it is running, the `stop()` method is called which forwards the call to the `cancellation()` method, the second mandatory method to be implemented by each thread. It is responsible for the termination of the main routine, which should exit as soon as possible once `cancellation()` is called. Using this pattern, each thread can implement its individual termination without affecting other threads. Alternatively to this pattern, POSIX compliant systems provide a `cancel` method for threads. This method has been avoided here, because it uses signals to abort blocking operations which may lead to an unpredictable behavior. The abstraction made here provides a more intuitive and better portable way for thread cancellation.

Beside the mandatory methods, a thread implementation may implement some optional callbacks. The `setup()` method is called right before the thread is started during the call of `start()` is blocked. This allows a thread to initialize data structures which should be accessible as soon as the call of `start()` returns. The opposite of `setup()` is the method `finally()`. This is called right after the thread exits and leaves the running state. Previously initialized data structures should be cleared within this method.

For threads derived from `JoinableThread`, there exists a public `join()` method which blocks the callee until the main routine is terminated. This method is useful to stall the deletion of shared data structures until the termination of the thread is complete. While it is mandatory to call the `join()` method of a `JoinableThread` to finally clean-up its state, the `DetachedThread` deletes its own object once the main routine exits.

4.5.6 Queue

Queues are a good way to exchange data between threads. They provide primitives to queue an object to a specific thread for processing. The queued object can be some data-container with data to process or just a task triggered by some event.

The STL of C++ provides well-designed classes for many standard containers such as queue, list, vector, etc. Since the C++98 standard does not include any threading capabilities, protection for concurrent access on those containers is also missing. For that reason, the `ibrcommon` library contains an implementation of a thread-safe queue as generic C++ template. Those queues can be limited to a maximum number of elements and support thread-safe concurrent access, as well as exclusive locking and blocking methods to retrieve and insert elements.

4.5.7 Reference Counting

If it is not reasonable to copy an object multiple times, a pointer to the object can be shared among several threads. Besides the thread-safety of the object itself, someone needs to be responsible for the deletion of the object once it is no longer needed. Since the processing order of references is most of the time undefined, a concept called *reference counting* is necessary.

Reference counting between several threads requires locking mechanisms to increment and decrement a variable atomically. Thus, it is only available in the STL of C++11 which provides the necessary prerequisites. But C++11 is not supported by all platforms and once again we need an own implementation to keep the software portable. Luckily, all we need to implement for a reference counting mechanism is the previously introduced `Mutex` class. The template class `refcnt_ptr` is part of `ibrcommon` library and accepts pointers and atomically counts each copy of itself. If the counter drops below 1, the pointer is automatically deleted.

4.5.8 BLOB

The BLOB mechanism is based on the previously described mechanisms like mutexes and reference counting approaches. The intention behind this structure is to provide a clean way to handle large amounts of data, while the programmer should not be responsible to do extra work to keep concurrent access thread-safe and should be allowed to copy objects as many times as necessary, without any performance loss. Moreover, the usage pattern should be always the same, no matter where the data is actually stored.

To design the BLOB mechanism, we start with the definition of requirements. First, each access to the data should be thread-safe. That means, if a thread gains access to a BLOB, any further access on the same data should be blocked until the previous access has been completed. To improve performance, it would be possible to allow concurrent read-only access, but due to the complexity of such an approach this is intentionally omitted here. Instead, read and write access to the data is provided through a special `iostream` object which acts as a guard object

and locks the whole BLOB as long as the guard exists. Additional to locking, the guard resets the state of the underlying structures. In case of streams, this resets the put as well as get pointers and clears error flags such as EOF.

When dealing with potentially large amounts of data, it is not an option to copy these data during processing. Thus, BLOB objects make use of reference counting introduced in Section 4.5.7 and create just one instance of each data unit. This instance is wrapped into the `refcnt_ptr` object which allows the compiler to copy the reference instead.

Another requirement is that the approach should be agnostic with regard to the actual place where the data is stored. Further, there should be a way to provide a custom implementation to handle data units. For that purpose, a generic interface exists to set a provider for BLOB objects. By default, this is set to a provider for memory-based BLOB objects. These store all the data in a `std::stringstream` instance, which is part of the STL. If memory storage is not sufficient, a provider for file-based BLOB objects exists which requires a path where temporary files are stored. The BLOB objects created by this provider, are based on `std::fstream` instances and protected against too many concurrently opened files using a semaphore.

Additional to different providers for temporary storage, there exists a special BLOB implementation to further optimize data processing when existing files are used as data input. A special BLOB implementation accepts a file path and provides the same functionality as any other BLOB object except that it is read-only. This is useful to avoid additional copying if an existing file should be processes but not modified.

4.5.9 Concurrent Read-only Locking

Even if not part of the BLOB concept, it is generally a good idea to consider a distinction of read-only and read-write locks. In some scenarios, it is reasonable to permit several threads concurrent read-only access to data structures. For these cases an additional mutex exists as class `RWMutex`. This mutex implements the same interface as the `Mutex` class does, thus it can be used in conjunction with `MutexLock` too. The difference here is that `MutexLock` will only acquire a read-only lock. Thus, multiple threads are allowed to acquire this lock simultaneously. If a thread requires exclusive access to the resources protected by the mutex, it may call the methods `enter_wr()` to wait until an exclusive access is granted or `trylock_wr()` to probe for it. Comparable to `MutexLock`, the specialized guard `RWLock` acquires the exclusive read-write lock of an `RWMutex` instance and release it if the declaration scope was left.

4.6 Data Serialization

Bundles are internally handled as objects with attached data encapsulated in BLOBs. Since object instances are not exchangeable with other bundle nodes, the bundle structure has to be encoded into a binary representation. The programming pattern utilized in IBR-DTN to convert bundles into binary data and vice versa is called serialization. Using classes providing serialization and deserialization functionalities, bundles can be written to and read from streams using the C++ shift operators. Listing 4.1 shows how a bundle is written to a stream. Line 1 instantiates the `DefaultSerializer` which is capable in encoding bundles according to the *Bundle Protocol Specification*. The instance accepts `std::ostream` instances as constructor parameter to write its data to. In line 2, a bundle is constructed and gets serialized to the standard output stream in line 3.

Listing 4.1: Write bundle data to standard output using the `DefaultSerializer`

```
1 DefaultSerializer serializer(std::cout);
2 Bundle bundle;
3 serializer << bundle;
```

The serialization procedure is utilized in every component where a bundle leaves the bundle node. This is the case in each convergence-layer, in the client API connections, and even if bundles are written to disk to get persistently stored. To read bundles from disk or incoming data-streams, the deserialization procedure is executed as shown in listing 4.2. Line 1 instantiate the `DefaultDeserializer` which is capable in reading bundle data according to the *Bundle Protocol Specification* [SB07]. The instance accepts `std::istream` instances as constructor parameter to read its data from this stream. In line 2, an empty bundle is constructed and filled in line 3 with the deserialized content of the standard input stream.

Listing 4.2: Read bundle data from standard input using the `DefaultDeserializer`

```
1 DefaultDeserializer deserializer(std::cin);
2 Bundle bundle;
3 deserializer >> bundle;
```

The design pattern using `Serializer` and `Deserializer` is very flexible if the data format changes, since all the routines for parsing and serialization are encapsu-

lated in very few classes. It also allows a developer to change the underlying format without the need to adapt those parts of the code, where these classes are used.

Moreover, the generic interface allows a developer to write additional serializer implementations with a specialized bundle encoding. This approach is used to gain the canonicalized form of bundles which is necessary for bundle security protocol operations as described in Section 4.12.4. The plain-text API also utilizes specialized implementation to encode bundle into a plain-text sequence and parse them from such data.

4.7 Event-system

The event-system introduced in Section 3.4 is part of the `daemon` package. It distributes raised events to components which are registered to the corresponding event type. The implementation is protected against concurrent access and distributes events using multiple workers. To achieve a good resource utilization, these workers process multiple events in parallel, while the number of workers scales with the number of available Central Processing Units (CPUs). Since it is possible that an event overtakes another one, it is required that the method which processes raised events of each registered component must be thread-safe.

To manage registered receivers efficiently, a dispatcher template provides methods to register components as receiver as well as methods for event distribution. This way, each event-type is associated with a dedicated list of receivers. A component which wants to register for a specific event-type, just registers directly at the corresponding dispatcher instance as shown in listing 4.3.

Listing 4.3: Register to an event

```
1 void componentUp() throw ()
2 {
3     EventDispatcher<BundleEvent>::add(this);
4 }
```

Once created, each event is passed to the corresponding dispatcher. In order to distribute the event, there are two options: Raise the event directly or queue it for deferred distribution. By raising the event directly, the thread which calls the dispatcher will block until the event is delivered to all components. If an event gets queued, the call to the dispatcher will return as soon as the event is queued to the event-switch. The listing 4.4 shows an example where a `BundleEvent` is created and queued afterwards. According to the event priority, a queued event is inserted into

one of three queues of the event-switch. As soon as there is an idle event worker, it pulls out the queued event and distributes it to the registered components.

Listing 4.4: Raise an event

```
1 void raise(...)  
2 {  
3     Event *evt = new BundleEvent(...);  
4     EventDispatcher<BundleEvent>::queue( evt );  
5 }
```

To process raised events, each component has to implement the `EventReceiver` template interface which requires to implement a `raiseEvent()` method for each event-type a component wants to receive. Listing 4.5 shows an example where a `BundleEvent` is received and processed. The event is passed as constant reference to the component. Thus, any mutable operation on the object is denied. This restriction is necessary since the object is passed to multiple components and each of them should receive the unmodified object.

Listing 4.5: Reception of an event

```
1 void raiseEvent(const BundleEvent &evt) throw()  
2 {  
3     // print out the bundles action  
4     std::cout << event.getAction() << std::endl;  
5 }
```

Each event-type is represented by its own class while each of them is derived from the `Event` class. The listing 4.6 shows the public methods of the class. A derived event implements at least the methods `getName()` and `getMessage()`. These describe the event instance for debugging and logging purposes. Additionally, every event can assign a custom priority using the `Event` constructor and may turn off the logging flag.

4.8 Data Storage

Classical network protocols are designed in that way that a single data packet fits into the volatile memory of a router. Considering the *Bundle Protocol*, a single bundle can be arbitrary large and thus it is not efficient to hold one or more of them in the volatile memory of a router. Especially if the router has limited resources as expected on embedded systems. Moreover, bundles are not necessarily forwarded

Listing 4.6: Abstract Event class

```
1  class Event
2  {
3      public:
4          // virtual destructor
5          virtual ~Event() = 0;
6
7          // get the name of this event.
8          virtual const string getName() const = 0;
9
10         // get a describing message for this event
11         virtual string getMessage() const = 0;
12
13         // get a string representation of this event.
14         virtual string toString() const;
15
16         // if this event should be logged, this method returns true
17         bool isLoggable() const;
18
19         // contains the priority of this event.
20         const int prio;
21
22     protected:
23         // constructor of Event
24         // Accepts a priority lower or higher than zero.
25         Event(int prio = 0);
26
27         // specify if this event should be logged or not
28         void setLoggable(bool val);
29     };
```

at the moment they are received. Instead bundles are kept until there is an opportunity to forward them, they expire, or another policy decides to purge them from the node. Thus it is very important to handle incoming bundles using a small memory footprint.

The most important concept to handle large amounts of data was already presented in Section 3.3. The BLOB objects provide a generic interface to data streams using the standard shift-operators of C++. This is a very common pattern also used by standard IO streams available in the C++ Standard Template Library. Furthermore, the internal storage principle is configured globally and completely hidden by a simple access pattern.

By default, the data of all BLOBS are stored in volatile memory using the standard C++ class `std::stringstream` as base. Since the program does not know any path to store data files to, this is the only way to handle data in this case. Once the library is initialized using a temporary path for files, the data is stored in files using the `std::fstream` class of C++. For very specific needs, a BLOB provider may be instantiated to create BLOB objects with a custom implementation.

As introduced in Section 3.2, the storage system is one essential part of a bundle node. Since the requirements and conditions of heterogeneous platforms are quite diverse, the storage system is separated using an interface to provide all necessary methods to manage bundles within the storage.

- **void store(Bundle)**

Accepts a Bundle and stores it until the life-time expires or a policy purges it. For performance reasons, it is allowed to defer the request to a background thread and return directly. In such a case, the storage should retain the acceptance signal for a requested custody until the store procedure is completed.

- **bool contains(BundleID)**

This method accepts a BundleID and returns true if there is already a bundle with the same source EID, creation time-stamp, creation time-stamp sequence number, and life-time. In case of a bundle fragment, the payload length and fragment offset is also considered.

- **MetaBundle info(BundleID)**

This method accepts a BundleID and returns the corresponding MetaBundle with additional data.

- **Bundle get(BundleID)**

Using this method a copy of the Bundle specified by the given BundleID is retrieved from the storage.

- **void get(BundleSelector, BundleResult)**

With this method bundles are selected using a given `BundleSelector`. The instance of `BundleSelector` provides a maximum number of bundles to select and a function which checks if a single bundle meets the requirements to be selected. Once the `get()` method is called, the storage will iterate over all bundles and put selected bundles into the `BundleResult` until either the maximum number of bundles is reached or all bundles in the storage are considered.

- **set<EID> getDistinctDestinations()**

This method returns a set with all destination EIDs of all bundles in the storage.

- **void remove(BundleID)**

Removes the bundle corresponding to the given `BundleID` from the storage.

- **void clear()**

Clears all bundles from the storage.

- **bool empty()**

Returns true if the storage is empty and false if not.

- **size_t count()**

Returns the number of bundles in the storage.

For different purposes, the implementation of the bundle node presented here provides three different storage engines.

- The `MemoryBundleStorage` is the default engine and uses volatile memory to organize stored bundles. Since all objects are available instantly and no disk access is necessary, this is potentially the fastest variant but needs more memory than the others.

- The `SimpleBundleStorage` stores the bundles into files. Each file is named using the unique primary key of the bundle and contains a complete bundle in the same binary format as it is sent over the wire. Different to other storage variants, this one implements a performance optimization for store and remove operations. Both operations queue a deferred job and return immediately. While the job is processed by a background task, the storage processes further operations as if the deferred jobs were already done. Moreover, the storage keeps an index to iterate through all bundles in memory which is

constructed from the bundle files on each start-up. Each bundle in the index is referenced by a `MetaBundle` object to provide fast access to `MetaBundle` objects. This way, the storage is almost as fast as the `MemoryBundleStorage` while storing, removing or iterating over bundles.

- The `SQLiteBundleStorage` depends on the SQLite library [Con14] and thus is only available on platforms supported by SQLite. This variant realizes a small memory footprint by keeping even bundle indexes on the file-system. All meta-data is stored in a Structured Query Language (SQL) database and each block of a bundle is stored to a dedicated file. The iteration through the list of bundles is done using SQL statements which only keep the necessary amount of data in memory. Furthermore, this implementation adds a special BLOB provider to associate existing payload files to BLOB instances using file-systems hard-links. This way it is not necessary to copy payload data to get a bundle copy.

4.9 Routing

The routing is realized using multiple extensions which, combined with each other, realize a specific behavior. The central routing component of the bundle node is the `BaseRouter`. This class manages all extensions and reacts to events such as incoming bundles and topology changes. It contains a set of already received (summary vector) and another set for purged bundles (purge vector). Both sets utilize a Bloom-filter to check if a bundle is part of the set. The summary vector is used for duplicate checking as already explained in Section 4.3.1, the second set contains bundles that should get purged out of the network, because they are finally delivered or no longer necessary due to another reason. To avoid these bundle-sets from growing infinitely, bundles are removed from the set as soon as the life-time expires. Since the Bloom-filter has to be rebuilt every time a bundle is removed, the expiration on these sets is executed deferred in an interval of 60 seconds. The Bloom-filters of both sets are transmitted to other peers using the routing data exchange. The summary vector allows a node to check if the considered peer already knows a bundle before it is selected for transmission. Using the purge vector, a node can iterate through its own storage and purge bundles matching the Bloom-filter. Due to the false-positive property of Bloom-filters, a peer might come to the assumption that a bundle is already known by a peer while that is not the case. Since such a probability is small when using a well-dimensioned data-structure, this drawback is negligible compared to

the data-amount that would be necessary to exchange a complete list of all known bundles on each encounter.

Also part of the `BaseRouter` is the `NeighborDatabase`. It provides a generic way to store neighbor associated data using a dedicated `NeighborEntry` for each neighbor. For example, this component is used to store the summary vector received during the routing exchange. Each bundle queued by a routing extensions has to pass this component which effectively limits the number of concurrently queued bundles per peer by throwing an instance of the `NoMoreTransfersAvailable` exception once the queuing capacity is exhausted. The `NeighborDatabase` also expires outdated data which may provoke a new routing exchange.

The routing exchange introduced in Section 3.6.1 is handled by an extension called `NodeHandshakeExtension`. This extension is a mandatory part of the `BaseRouter` and is always present no matter how the bundle node is configured. It provides methods to initiate an exchange and prevents this mechanism from being executed too often. On incoming exchange requests, the extension responses with the summary vector as well as the purge vector, if that is requested. Moreover, it adds a request for the purge vector to every outgoing exchange request message.

Retransmissions are handled by a dedicated routing extension which is also always part of the `BaseRouter`. This extension listens to events of type `RequeueBundleEvent` and delays the retransmission of this bundle using the exponential back-off algorithm with an upper bound. A retransmission is necessary if a bundle is rejected due to a temporary failure. This could be the case, if a peer seems to be reachable, but a necessary connection could not be established. The extension will re-queue the bundle for a maximum number of retries and finally raise an event of type `TransferAbortedEvent` if the limit is reached or if the bundle is expires beforehand.

Beside the standard extensions, an unlimited number of additional extensions may be attached to the `BaseRouter` to realize different routing approaches. All extensions receive multiple callbacks defined by the interface shown in listing 4.7. The method `eventDataChanged()` defined in line 11 indicates that the topology or the queuing capacity has changed. In any case, the routing extension should search again for bundles to deliver to the given peer. A successful transfer of a bundle to a peer is indicated by the `eventTransferCompleted()` method (line 15). In most of the cases, a call of this method is handled the same way as the method `eventDataChanged()`. For some routing approaches it might be useful to know how many times a bundle has been forwarded and who has received it. Every time the bundle node has received a new bundle, the `eventBundleQueued()` method (line 19) is called. Using this method the routing extension can directly decide,

Listing 4.7: Abstract Routing Extension class

```
1  class RoutingExtension {
2      public:
3          RoutingExtension();
4          virtual ~RoutingExtension() = 0;
5
6          virtual void componentUp() throw () = 0;
7          virtual void componentDown() throw () = 0;
8
9          // This method is called every time something has changed. The module
10         // should search again for bundles to transfer to the given peer.
11         virtual void eventDataChanged(const EID &peer) throw () { };
12
13        // This method is called every time a bundle has been
14        // completed successfully
15        virtual void eventTransferCompleted(const EID &peer,
16            const MetaBundle &meta) throw () { };
17
18        // This method is called every time a bundle was queued
19        virtual void eventBundleQueued(const EID &peer,
20            const MetaBundle &meta) throw () { };
21
22        // If some data of another node is required. These method is called
23        // to collect all necessary identifier of data items.
24        virtual void requestHandshake(const EID &peer,
25            NodeHandshake &handshake) const { };
26
27        // If a handshake message is received , this method is called to collect
28        // the different data items generated by the router extensions.
29        virtual void responseHandshake(const EID &peer,
30            const NodeHandshake &request, NodeHandshake &response) { };
31
32        // After a handshake has been completed every module can
33        // process the handshake response.
34        virtual void processHandshake(const EID &peer,
35            NodeHandshake &response) { };
36
37    protected:
38        // Transfer one bundle to another node.
39        void transferTo(const EID &destination, const MetaBundle &meta);
40
41        BaseRouter& operator*();
42    };
```

if the bundle should get forwarded directly without the need to iterate through the whole storage. By reacting directly to this signal, the bundle delay is shortened given that the queuing capacity is not exhausted. The callbacks defined in line 24, 29, and 34 are used to prepare and react to routing exchange requests and responses. The `requestHandshake()` method indicates that a request message gets prepared and each extension has the opportunity to add further request items within this call. In the method `responseHandshake()`, an extension can response to one or more request items contained in the `request` parameter. The generated response items should be attached to the object in the `response` parameter. If the response is finally received, it is passed to all extensions using the `processHandshake()` callback. Beside callbacks to implement, the interface includes the asterisk operator to provide access to the `BaseRouter` instance and a method to initiate a transfer of a bundle with a given destination.

The approach of concurrently running routing extensions has a lot of advantages. Multiple routing approaches with different complexity can select independently bundle for transmission. Since each of them is separated in a dedicated thread, the resources of multi-core architectures are better utilized as with a single procedure loop. Further, a slow algorithm can not slow-down other ones. The down-side of the design is that several extensions might choose the same bundle for transmission to the same peer. For that reason, the `NeighborDatabase` keeps track of all queued bundles and rejects further queuing of the same bundle with an instance of the `AlreadyInTransitException` exception. Since queuing operations are done concurrently, the `NeighborDatabase` needs to be locked while working with its data.

4.9.1 Neighbor Routing

The neighbor routing extension follows a routing scheme also known as direct delivery. This principle forwards bundles to its final destination if it is directly reachable using a convergence-layer. The corresponding extension implements the default behavior of the bundle node, if no other routing approach is configured and reacts to calls of `eventDataChanged()` and `eventBundleQueued()`. Since the extension has nothing to do if a bundle has been successfully forwarded, calls of `eventTransferCompleted()` are ignored.

On every call of `eventDataChanged()`, the extension will execute a search procedure looking for bundles to deliver to the considered neighbor. The listing 4.8 shows the processing of the `SearchNextBundleTask` which is generated on each call of `eventDataChanged()`. First, the generic task is checked if it is of the right instance (line 4). This approach is similar to the processing of events described in

Listing 4.8: Search for bundles to transfer in the Neighbor Routing Extension

```
1 BundleResultList list;
2
3 try {
4     SearchNextBundleTask &task =
5         dynamic_cast<SearchNextBundleTask&>(*t );
6
7     // lock the neighbor database while searching for bundles
8     {
9         // this destination is not handled by any static route
10        MutexLock l(db);
11        NeighborEntry &entry = db.get(task.eid, true);
12
13        // check if enough transfer slots are available
14        // (threshold reached)
15        if (!entry.isTransferThresholdReached())
16            throw NoMoreTransfersAvailable();
17
18        // create a new bundle filter
19        BundleFilter filter(*this, entry);
20
21        // query an unknown bundle from the storage
22        list.clear();
23        (**this).getSeeker().get(filter, list);
24    }
25
26    // send the bundles as long as we have resources
27    for (list<MetaBundle>::const_iterator
28        iter = list.begin(); iter != list.end(); ++iter)
29    {
30        try {
31            // transfer the bundle to the neighbor
32            transferTo(task.eid, *iter);
33        } catch (const AlreadyInTransitException&) { };
34    }
35 } catch (...) { };
```

Section 4.7. In case this task is of another instance the try-catch-block is executed and the routine is left. Otherwise, we proceed with the locking of the neighbor database using a guard (line 10) in a separate scope from line 8 to 25. This lock is required because the `BundleFilter` (line 19) will access the neighbor entry retrieved from the neighbor database in line 11. Since this entry is accessed directly instead by copying it, we must ensure that it is not deleted while we are working with it. This is accomplished by locking the neighbor database (line 10). Before we start searching for bundles to transfer, we check in line 15 the available transfer capacity of the neighbor. If this is already exhausted, the routine will stop by throwing an exception.

The bundle selection (line 23) is done using an implementation of the `BundleSelector` shown in listing 4.9. As explained in Section 4.8, instances of `BundleSelector` are used to select a limited set of bundles from the storage. In this case, the limit is set accordingly to the currently available transfer capacity (line 10). If the selection process is executed, the method `shouldAdd()` is called for each bundle in the storage. The process stops if enough bundles have been selected or if all bundles have been considered. The `shouldAdd()` method returns with a boolean which indicates whether the considered bundle should be added to the result-set. To achieve a good performance during this selection process, the method checks each proposed bundle for disqualifying attributes. Thus, the method starts with excluding bundles with a remaining hop limit of zero (line 17). Those bundles are no longer qualified for forwarding to any other node. Only the delivery to an application is acceptable in this case. In line 19, the extension differentiate the destination address of bundles into singleton and non-singleton endpoints. The latter case is not handled at all by the neighbor routing extension. Bundles address to a singleton endpoint will be handled if they are not addressed the local EID (line 21) and addressed to the proposed next-hop (line 25). Finally, the method checks if the bundle is not already in the summary vector of the neighbor (line 33). If all checks are passed, the method returns true and the bundle will be added to the `BundleResult` which was passed to the storage during the call of `get()`. Once the selection of bundles is finished, the selected bundles are queued for a transmission using the `transferTo()` method as shown in listing 4.8, line 33.

Additional to the call of `eventDataChanged()`, the neighbor routing extension also executes one `ProcessBundleTask` for each existing neighbor, every time the `eventBundleQueued()` method is called. Different to the `SearchNextBundleTask`, this task does not iterate through the storage to select bundles. Instead it just checks if the neighbor is a potential receiver of the given bundle. Listing 4.10 shows the procedure beginning with the checking of the task type (line 2) and locking of

Listing 4.9: Bundle Filter implementation used in the Neighbor Routing Extension

```
1 class BundleFilter : public BundleSelector {
2     public:
3         BundleFilter(NeighborRoutingExtension &e,
4             const NeighborEntry &entry) : _extension(e), _entry(entry)
5         { };
6
7         virtual ~BundleFilter() { };
8
9         virtual Size limit() const throw () {
10             return _entry.getFreeTransferSlots();
11         };
12
13         virtual bool shouldAdd(const MetaBundle &meta) const
14             throw (BundleSelectorException) {
15                 // check Scope Control Block
16                 // do not forward bundles with hop limit == 0
17                 if (meta.hopcount == 0) return false;
18
19                 if (meta.get(PrimaryBlock::DESTINATION_IS_SINGLETON)) {
20                     // do not forward local bundles
21                     if (meta.destination.sameHost(BundleCore::local))
22                         return false;
23
24                     // do not forward bundles for other nodes
25                     if (!meta.destination.sameHost(n.eid))
26                         return false;
27                 } else {
28                     // do not forward non-singleton bundles
29                     return false;
30                 }
31
32                 // do not forward bundles already known by peer
33                 if (n.has(meta)) return false;
34
35                 return true;
36             };
37
38     private:
39         NeighborRoutingExtension &_extension;
40         const NeighborEntry &_entry;
41     };
}
```

Listing 4.10: Processing a recently queued bundle in the Neighbor Routing Extension

```
1  try {
2      const ProcessBundleTask &task =
3          dynamic_cast<ProcessBundleTask&>(*t);
4
5      // lock the neighbor database while searching for bundles
6      {
7          // this destination is not handled by any static route
8          MutexLock l(db);
9          NeighborEntry &entry = db.get(task.nexthop, true);
10
11         // check Scope Control Block
12         // do not forward bundles with hop limit == 0
13         if (meta.hopcount == 0) throw NoRouteKnownException();
14
15         if (meta.get(PrimaryBlock::DESTINATION_IS_SINGLETON)) {
16             // do not forward local bundles
17             if (meta.destination.sameHost(BundleCore::local))
18                 throw NoRouteKnownException();
19
20             // do not forward bundles for other nodes
21             if (!meta.destination.sameHost(n.eid))
22                 throw NoRouteKnownException();
23         } else {
24             // do not forward non-singleton bundles
25             throw NoRouteKnownException();
26         }
27
28         // do not forward bundles already known by peer
29         if (n.has(meta)) throw NoRouteKnownException();
30     }
31
32     // transfer the bundle to the neighbor
33     transferTo(task.nexthop, task.bundle);
34 } catch (...);
```

the neighbor database (line 8). The code to check if the bundle should get forwarded to the neighbor (line 13 to 29) is similar to that code used for selecting bundles in the `BundleSelector` of the other task. But instead of returning true or false, an exception is thrown if the bundle is identified to be excluded from forwarding to that specific neighbor. If all checks are passed, the `transferTo()` call (line 33) initiates the transfer of the bundle to the neighbor.

4.9.2 Static Routing

In addition to neighbor routing, the bundle node accepts static routes which are handled by a dedicated routing extension. These routes are expressed as pair of a destination pattern and a neighbor. If the specified neighbor is recognized as available peer, due to the opportunistic discovery or statically configured, the associated routes will be considered to check if there are bundles in the storage that match the destination pattern. The pattern is expressed as POSIX regular expression and allows very flexible patterns. E.g. `^dtn://[[[:alpha:]] .moon.dtn/[[[:alpha:]]]` matches all EIDs with the scheme dtn and a host part which ends with .moon.dtn.

To gather bundles to forward, the extension uses the same principle as the neighbor routing. All calls of `eventDataChanged()` and `eventBundleQueued()` create a dedicated task which is processed by a worker thread. To select bundles to forward, the extension uses an implementation of a `BundleSelector` similar to the implementation shown in listing 4.9. But instead of simply checking if the bundle is directly addressed to the peer, the implementation checks if one of the routes for the currently considered neighbor matches the destination of the bundles. Further, the extension considers even bundles addressed to non-singleton EIDs.

4.9.3 Flooding

Besides the direct delivery approach and static routing, the flooding routing scheme is one of the simplest routing approaches. It simply forwards all bundles to all neighbors and realizes multi-hop routing even of bundles addressed to non-singleton endpoints. A common known statement for such a routing scheme is that it always utilizes the shortest path for a bundle, but each bundle consumes a fraction of the capacity of all other paths too.

To keep this approach simple, there is no exchange of routing data required. Thus, a bundle node does not receive the summary vector of a peer before starting the transfer of bundles. That means a bundle node would forward all bundles, even if the peer already received all of them by another peer. At least, a node will remember which bundles have already been forwarded to a peer and will not transfer again all bundles on the next contact.

As implemented in all other routing extensions, this extension also processes tasks in a worker thread and reacts to events signaled by the `BaseRouter`. Different to the implementation of the `BundleSelector` shown in listing 4.9, this extension processes even bundles addressed to non-singleton endpoints to realize some sort of group routing and excludes bundles handled by the neighbor routing. The latter property is an optimization and avoids bundles to get queued twice or even forwarded to all neighbors if the destination of a bundle is directly reachable.

4.9.4 Epidemic Routing

The epidemic routing approach is explained in [VBoo]. In contrast to the flooding routing scheme, epidemic routing exchanges a summary vector before bundles are forwarded. Using the summary vector, a bundle node can decide if a bundle is already known by the peer without wasting any transfer capacity. In the implementation presented here, the summary vector exchange, which is done during the routing data exchange, is merged into the `NeighborEntry` objects of the neighbor database. This way, each routing scheme automatically uses the received summary vector to check if a bundle is already known by the peer.

4.9.5 PRoPHET Routing

Another routing extension implements the Probabilistic Routing Protocol using History of Encounters and Transitivity (PRoPHET) [LDDG12]. This approach is based on the epidemic routing approach, but prunes the epidemic tree by removing paths which look less likely to provide an effective route for delivery of bundles to its destination. In order to achieve that, each encounter with other nodes is measured to gain a table of delivery predictabilities, while intervals of connectivity increment and periods of disruption decrease the values associated to an EID. The delivery predictability is used as routing metric to decide if a node should forward bundles to another peer or not. The authors also mention that it would be common that only a subset of the whole network uses PRoPHET. At the edges of such a subnet, called PRoPHET zone, the nodes may act as gateway between PRoPHET and other routing schemes. Such gateways are very easily to realize by using TBR-DTN. The ability to run multiple routing schemes simultaneously is an inherent part of the architecture.

To exchange necessary routing data, an out-of-band channel based on TCP is proposed. Other underlay protocols like UDP or even direct access to IP are mentioned but an exact specification is left out for future revisions. The channel is used to exchange the delivery predictability tables, a list of acknowledged bun-

dles, and to offer as well as request bundles during the exchange phase. The actual transmission of the bundle should be done by the bundle node.

Since the authors of PRoPHET isolate the Routing Protocol Agent (RPA) which contains the routing protocol implementation from the BPA, they specify the necessary functionality which must be provided by the BPA to the RPA in Section 2.2 of [LDDG12]. However, the RPA of IBR-DTN is not strictly separated from the rest of the bundle node. Thus, the interface is different in some parts compared to the specification.

■ Get Bundle List

The RPA of PRoPHET needs to access the complete list of bundles and their attributes stored in the BPA to offer bundles to the neighbors. In IBR-DTN the bundles are not offered. Instead, each neighbor requests a summary vector in addition to the delivery predictability table using the routing exchange as explained in Section 3.6.1. Using these data, a bundle node can decide locally which bundle should get forwarded to the neighbor.

■ Send Bundle

In PRoPHET a neighbor requests bundles from other RPA using the out-of-band channel. Since this channel is not available in the BPA, the RPA has to forward the request to the BPA to initiate the bundle transfer. In IBR-DTN each routing extension has access to a `transferTo()` method to initiate a bundle transmission.

■ Accept Bundle

If a RPA is responsible in forwarding and receiving bundles instead of the BPA, this method forwards a received bundle to the BPA. Otherwise, this method is implemented as notification to tell the RPA that another bundle has been accepted. In IBR-DTN, the routing extension gets notified if a new bundle is received.

■ Bundle Delivered

If a RPA is responsible in forwarding and receiving bundles instead of the BPA, this method tells the BPA that a bundle has been delivered to its final destination. Otherwise, this method implements a notification to the RPA which will add the bundle to the acknowledgment set. In IBR-DTN, the routing extension gets notified if a bundle transfer is completed and if the receiver was the final destination.

■ Drop Bundle Advice

If a bundle was acknowledged within the PRoPHET zone, all bundle nodes

are free to purge the bundles from the storage. This call is a notification by the RPA to tell the BPA that the bundle should get purged and no longer offered to other nodes. The `BundlePurgeEvent` of IBR-DTN introduced in Section 3.4 is equivalent to this call.

■ Route Import

The PRoPHET interface of the RPA supports to import routing tables from RPAs not supporting PRoPHET. By importing external routing tables into the PRoPHET zone, it is possible to realize a gateway between two different routing zones. IBR-DTN does not explicitly implement such a mechanism. Instead, a routing extension may directly access another one to query all the required data.

■ Route Export

The PRoPHET interface of the RPA supports to export routing tables to RPAs not supporting PRoPHET. By exporting the delivery predictability table to other RPAs, it is possible to realize a gateway between two different routing zones. IBR-DTN does not explicitly implement such a mechanism. Instead, a routing extension may directly access another one to query all the required data.

The implementation of PRoPHET routing as part of IBR-DTN not only differs in the way of integration compared to the PRoPHET specification. A significant detail is the lack of an out-of-band channel. All routing related data is requested and exchanged using the *Bundle Protocol* and the convergence-layers of the bundle node. This is possible since the neighbor routing extension for direct delivery is always active and delivers bundles addressed to directly connected nodes even if there is no previous exchange of routing data. Thus, the implementation does not rely on a functional IP stack.

Another detail is the way how bundles are selected for forwarding. The specification proposes an offer-request scheme, which requires to send a list of bundles first. The receiver of the list selects bundles and explicitly requests them from the other peer. Since this approach requires a lot of processing and additional out-of-band signaling during the transmission, we decided to realize that a different way. IBR-DTN just exchanges a summary vector of all known bundles encoded as Bloom-filter on each encounter. Using the summary vector and the delivery predictability table, the routing extensions can decide locally which bundles should get forwarded to the neighbor.

The implementation is configurable using a bunch of parameters.

- **prophet_p_encounter_max** (Default: 0.7)
Affects how strong the predictability is increased on each encounter.
- **prophet_p_encounter_first** (Default: 0.5)
The predictability of a neighbor on the first encounter.
- **prophet_p_first_threshold** (Default: 0.1)
Lower bound of predictabilities to keep. If a predictability falls below this value, the entry is dropped.
- **prophet_beta** (Default: 0.9)
Weight of the transitive property.
- **prophet_gamma** (Default: 0.999)
Determines how quickly predictabilities age.
- **prophet_delta** (Default: 0.01)
(1-delta) is the maximum predictability.
- **prophet_time_unit** (Default: 1)
Time unit in seconds.
- **prophet_i_typ** (Default: 300)
Typical time interval between two node encounters.
- **prophet_next_exchange_timeout** (Default: 60)
Timeout how often handshakes should be executed.
- **prophet_forwarding_strategy** (Default: GRTR)
The forwarding strategy used GRTR | GTMX.
- **prophet_gtmx_nf_max** (Default: 30)
Maximum times to forward in the GTMX strategy.

The *Bundle Protocol* does not specify any limitation of EIDs in regard to unicast or multicast addressing. Thus, the string representation of an EID does not necessarily indicate if there are just one or multiple receivers. For that reason, bundles carry a dedicated bit in the processing flags that indicates if the destination is a singleton or not. The PRoPHET approach is designed to handle singleton bundles only. If there are multiple peers announcing the same EID within a PRoPHET zone, the routing might choose only the shortest path which leads to some sort of anycast behavior. To route even bundles addressed to a group of nodes, a reasonable approach is to fall-back to the epidemic principle and deliver the bundles to all nodes.

Since that approach would distribute those bundles to all nodes in the PRoPHET zone, we can truncate the paths by exploiting the delivery predictabilities. Similar to the reverse path forwarding approach [TW10], a bundle gets only forwarded if the potential next-hop has a delivery predictability above a given threshold for the source EID of the bundle. In fact, this approach is not limited to PRoPHET, but would work with all routing protocols that provide a table of destinations to its peers.

4.10 Convergence-Layers

As explained in Section 3.2 a bundle node always inter-connects to other bundle nodes using convergence-layers in order to exchange bundles. Considering the ISO/OSI networking stack, such a layer connects the bundle node with the Transport Layer of the underlay network. In this section, we will mention that the transport layer and even the network layer is not necessary for all convergence-layers, because required features such as error detection and correction is intentionally left out or implemented directly in the convergence-layer. Moreover, the end-to-end principle and routing features of the network layer is not necessary in DTNs, since this is already an essential part of the bundle node architecture.

Convergence-layer implementations are designed as optional part of IBR-DTN. They may be enabled or not, depending on the specific configuration or even excluded during compilation due to a missing dependency. Since the *Bundle Protocol Specification* does not name any convergence-layer as requirement, it is even possible to run the bundle node without any convergence-layer. In this case, it only exchanges bundles between applications directly connected to the bundle node.

Basically, there exist two types of convergence-layers. Connection-oriented convergence-layers need to initiate a connection to a peer before bundle transfer is possible. Most of the time, connections are bi-directional and allow immediate acknowledgments for transmitted bundles. Moreover, they may give feedback, if the connection set-up was unsuccessful. Connection-less convergence-layers are able to send bundles directly without a connection set-up which may require several round-trips. This type of convergence-layer is well-suited if a path allows only unidirectional transmissions or if the propagation delay is very high.

Based on the previous explanation, we can conclude that convergence-layers should be able to receive bundles from and transfer bundles to other bundle nodes in range. Additionally, connection-orientated ones should be able to initiate a connection if the bundle node requests that. The listing 4.11 shows the interface for each instance of `ConvergenceLayer`. The method returns in line 8 the specific protocol identifier for this convergence-layer. Based on the discovered node pa-

Listing 4.11: Interface for Convergence-Layers

```
1 class ConvergenceLayer
2 {
3     public:
4         // virtual destructor
5         virtual ~ConvergenceLayer() = 0;
6
7         // returns the protocol specification used for discovery
8         virtual Protocol getDiscoveryProtocol() const = 0;
9
10        // queues a job for transmission to a node
11        virtual void queue(const Node &n, const BundleTransfer &job) = 0;
12
13        // initiates a connection to a node
14        virtual void open(const Node&) {};
15    };
```

rameters, this identifier is used to queue bundles to the right convergence-layer once they are queued for transmission to a node. The queuing itself is done using the method in line 11. It requires an instance of `Node` which holds all the discovered parameters of the targeted bundle node. The class `BundleTransfer` encapsulates an instance of `EID` and a reference to the bundle to transfer as `BundleID`. The last method (line 14) of the interface is optional and initiates a connection set-up to another peer without queuing a bundle. In environments where discovery is not applicable or connection set-ups are only permitted in one direction, such a method is used to initiate a persistent connection between two bundle nodes even if there are no bundles to exchange.

4.10.1 Neighbor Discovery

Beside the transmission of bundles, a convergence-layer is optionally responsible in discovering neighbors as soon as they enter the communication range. Such a mechanism is specified in [EAGB12] for IP-based convergence-layers such as the TCP-CL or the UDP-CL mentioned in later sections. All other convergence-layers have to provide a way to detect neighbors on their own. Typically, neighbor discovery is done using beacons which are broadcasted to all neighbors. If such a beacon is received, then the node assumes that there is an opportunity to transfer bundles between itself and the peer. Since that assumption is only true for bidirectional communication channels, the IPND specification [EAGB12] proposes a

Bloom-filter within the discovery beacon which contains all neighbors visible to the bundle node. If a bundle node receives a beacon and finds itself within the Bloom-filter, it assumes that the connection allows bidirectional communication.

Beacons received by a convergence-layer are forwarded to the *DiscoveryAgent* component, mentioned in Section 4.2, which does further data processing and invokes corresponding events to notify other components about the change of the network topology. This component merges several beacon sources and recognizes if a neighbor disappears. In order to achieve that, each beacon has a time-out value attached and if all beacons corresponding to a neighbor expire, the neighbor is considered as gone.

For each interface of the bundle node an outgoing beacon is generated. The *DiscoveryAgent* asks all registered components to add their specific discovery data to the beacon. Based on the interface, the components can decide if they want to be visible in that beacon or not. The complete beacon gets forwarded again to all registered components which are responsible in broadcasting it to all reachable neighbors. How exactly this is done depends on the specific implementation.

4.10.2 UDP Convergence-Layer

The *RFC 7122* [KJO14] defines a convergence-layer using UDP or other connection-less datagram protocols to transfer bundles between bundle nodes. UDP as underlying transport protocol has the advantage that even unidirectional links are sufficient to transmit bundles to a bundle node. The downside is that this approach does not provide reliability or any kind of flow control. Except of a checksum for each datagram, there is no mechanism for error detection or even correction. For that reasons, the applicability in conjunction with the BP is questionable. To mitigate the issues of the unreliable transport protocol, the *RFC 7122* suggests to implement DCCP or LTP as part of the convergence-layer to gain a more reliable channel.

The UDP convergence-layer of IBR-DTN does not include an implementation of DCCP or LTP. Due to the complexity of those protocols, we decided to follow a less complex approach which makes an implementation even feasible on sensor-nodes. That approach is part of the Datagram Convergence-Layer and introduced in Section 4.10.4.

In IBR-DTN, the UDP convergence-layer simply encapsulates a single bundle into a UDP datagram without any additional header. Since the Maximum Transmission Unit (MTU) of the link limits the maximum bundle size, the convergence-layer applies pro-active fragmentation to bundles with a size larger than the MTU (see Sec-

tion 3.7). Incoming bundles are decoded and forwarded to the routing component using an `BundleReceivedEvent`, see Section 3.6.4 for details.

As explained in Section 4.10.1, the convergence-layers are responsible for discovery of neighbors. The IPND approach specified in [EAGB12] defines a generic protocol for all convergence-layers based on IP. This protocol is implemented as the dedicated `IPNDAgent` component which does this job in place of all IP-based convergence-layers. Thus, the UDP convergence-layer does not include an additional mechanism for neighbor discovery.

4.10.3 TCP Convergence-Layer

The convergence-layer presented in this section is based on the reliable and connection oriented TCP protocol. This transport protocol provides error correction and flow control to applications built on top of it. The *Delay-Tolerant Networking TCP Convergence-Layer Protocol* [DOP14] is designed to work on top of TCP connections with disruptions and provides additional support to acknowledge or reject bundles. The protocol also specifies optional keep-alive frames which give an application the ability to detect disrupted connections more reliable than the system reflects to applications.

magic='dtn!'		
version	flags	keepalive_interval
local EID length (SDNV)		
local EID (variable)		

Figure 4.2: TCP convergence-layer Contact Header Format

Using this protocol, each connection between two bundle nodes starts with an exchange of a contact header. The structure of this header is shown in Figure 4.2 and includes the protocol version, flags, a keep-alive interval, and the EID of the peer. The flags of the header are used to signal if optional features are supported or not. These features are explained below.

■ Acknowledgments

The first bit of the flags indicates that the peer requests acknowledgments. Since bundles are of arbitrary size, the encoded format of the bundle is segmented during the transmission. If acknowledgments are requested and the peer supports this feature, each segment is acknowledged by a short message. Only this way, a convergence-layer gets knowledge about how much of the

bundles has arrived at the peer, because the standard socket API does not provide feedback about how much data actually arrived at the peer.

■ **Reactive Fragmentation**

The second bit requests reactive fragmentation (see Section 3.7). If both peers support this feature and the connection gets interrupted during a transmission, the convergence-layer on the receiver side can generate a bundle fragment using the already received data. The sender may also generate a fragment of the not transmitted part of the bundle and forward that remaining bundle fragment on a later contact or over a different path. In order to determine how much of the data has been received by the peer, support for acknowledgments is necessary to realize this feature.

■ **Bundle Refusal**

Bit three indicates support for bundle refusal. Since a transmission of a bundle with an arbitrary size also takes an arbitrary amount of time to transfer, the bundle refusal feature allows a convergence-layer to stop an ongoing transmission if it detects that the incoming bundle does not match the local policies. Usually, the primary bundle block is part of the first received segment. If the convergence-layer decodes that data on-the-fly, then it can easily check if the bundle has been already received from another peer or if other parameters contradict the local policies. In such a case, a message for each rejected segment is sent to the sender of the bundle. This feature is only enabled if acknowledgments are available.

■ **Length Messages**

By setting the fourth bit of the header flags, the peer requests messages containing the length of the bundle before it is actually transmitted. Since the actual size of an incoming bundle is only available after reading the header of the last block, these messages provide a way to indicate the actual size of the next bundle before the transmission starts. Using this hint, a peer can refuse too large bundles without wasting precious transmission capacity.

In IBR-DTN, there exist two classes to realize the convergence-layer implementation. The TCP server is represented by the `TCPConvergenceLayer` class and manages incoming connections. The connection to a peer is handled by the `TCPConnection` class. It contains a sender and a receiver part. Both operate mostly independent of each other. The sender accepts bundles queued by the `TCPConvergenceLayer` and serializes them into an instance of the `StreamConnection` class. This is a TCP independent implementation of the

Delay-Tolerant Networking TCP Convergence-Layer Protocol with a STL stream interface. The receiver part of the `TCPConnection`, deserializes data received through the `StreamConnection` and passes the received bundles to the `Router` component using a `BundleReceivedEvent`, see Section 3.6.4 for details.

Since this protocol is not designed to provide any security or authentication, a protocol extension was developed to signal TLS support to a peer and initiate an encrypted and authenticated connection between two bundle nodes. The details are presented in Section 3.9.5.

As explained in Section 4.10.1, the convergence-layers are responsible for discovery of neighbors. The IPND approach specified in [EAGB12] defines a generic protocol for all convergence-layers based on IP. This protocol is implemented as the dedicated `IPNDAgent` component which does this job in place of all IP-based convergence-layers. Thus, the TCP convergence-layer does not include an additional mechanism for neighbor discovery.

4.10.4 Datagram-based Convergence-Layers

IBR-DTN includes a generic convergence-layer implementation for datagram oriented transport layers. It allows an adaptation of new transport layers such as Ethernet, IEEE 802.15.4, or even serial links without much effort. Since datagram-based transport layers typically follow the best-effort principle and are known as unreliable, the datagram convergence-layer adds mechanisms for error correction and flow control.

To integrate new transport layers, the abstract class `DatagramService` shown in listing 4.12 has to be implemented. Such a *datagram service* provides the following configuration elements by implementing the methods shown in line 6-19.

■ Service Tag

A service tag is necessary to create unique service entries in the discovery beacons based on the IPND protocol. Using such a service entry, the convergence-layer instance is announced to the neighbors.

■ Service Description

A service description is a data-array which contains contact parameters of the local service. This data is linked with the service tag and attached to discovery beacons.

■ Interface

Each *datagram service* is assigned to an interface which is actually a unique string identifier representing an interface, even if it is virtual.

■ Protocol

For each adapted protocol, a unique protocol identifier is specified which is used to queue bundles to the right convergence-layers.

■ Parameters

Additional to the parameters presented above, each convergence-layer must define its own transport layer specification and requirements. The *datagram service* can choose between several flow control modes. Beside the option to disable flow control, which also disables the error correction, it is possible to select stop-and-wait or a sliding-window approach. Additionally, the parameters include a range of sequence numbers, the maximum datagram length, and how many retries should be done until the retransmission mechanism aborts the transmission.

Beside the parameters, the class `DatagramService` requires that each service implementation provides a `bind()` method, which gets called to initialize the *datagram service* and bind to an interface, if necessary. In the `shutdown()` method, the service should terminate and release its resources. Further, two send methods are required, one for datagrams directed to a neighbor and one for broadcast datagrams. The `recvfrom` method is provided to poll the *datagram service* for incoming datagrams. This method must block the call while there is no new data.

To transmit potentially large bundles using datagrams, it is necessary to split them up into segments. Pro-active fragmentation is not feasible here, since it would require to replicate the primary bundle block in each datagram and the available payload capacity might be even smaller than the encoded primary bundle block. The header of each message includes a type, flags, a sequence number, and the payload length. These parts are not encoded in the payload buffer during the call of the `send()` methods. Instead it is specific to the implementation of the *datagram service* how the message header gets encoded.

Bundle data is transmitted in messages marked with the type *segment* while discovery beacons are sent via *broadcast* messages. Two additional types for *Acknowledgements (ACKs)* and *Negative Acknowledgments (NACKs)* are required to provide flow control and error correction. Since there are only four message types, it is possible to encode them in just 2 bit. Additional to the type, each message may carry several flags. Basically, only two bits are required to mark a data segment that carries the first or the last segment of a bundle. If a whole bundle fits into a single segment, then both flags are set.

The minimal number of required bits to cover the whole feature-set of the *datagram convergence-layer* is five bits. Two bits for the message type, two bits for

Listing 4.12: Interface for Datagram Services

```
1 class DatagramService {
2     public:
3         // Virtual destructor.
4         virtual ~DatagramService() = 0;
5
6         // Returns the tag for this service used in discovery messages.
7         virtual const string getServiceTag() const;
8
9         // Returns the service description as string.
10        virtual const string getServiceDescription() const = 0;
11
12        // Return the used interface as vinterface object.
13        virtual const vinterface& getInterface() const = 0;
14
15        // Returns the protocol identifier for this type of service.
16        virtual Protocol getProtocol() const = 0;
17
18        // Returns the parameter for the connection.
19        virtual const Parameter& getParameter() const = 0;
20
21        // Bind to the local socket.
22        virtual void bind() throw (DatagramException) = 0;
23
24        // Shutdown the socket.
25        virtual void shutdown() = 0;
26
27        // Send the payload as datagram to a defined destination
28        virtual void send(const char &type, const char &flags,
29                          const unsigned int &seqno, const string &address,
30                          const char *buf, size_t length) throw (DatagramException) = 0;
31
32        // Send the payload as datagram to all neighbors (broadcast)
33        virtual void send(const char &type, const char &flags,
34                          const unsigned int &seqno, const char *buf,
35                          size_t length) throw (DatagramException) = 0;
36
37        // Receive an incoming datagram.
38        virtual size_t recvfrom(char *buf, size_t length, char &type,
39                               char &flags, unsigned int &seqno, string &address)
40                               throw (DatagramException) = 0;
41    };
```

first/last segment, and one bit to encode the sequence numbers 0 and 1. If a *datagram service* decides to disable flow control, the sequence number and the types for ACKs and NACKs are not required.

Similar to the implementation of the TCP convergence-layer, the datagram-based convergence-layer is separated into a `DatagramConvergenceLayer` and several instances of `DatagramConnection`. The `DatagramConvergenceLayer` is the receiver for all incoming datagrams, decodes the header, forwards discovery beacons to the `DiscoveryAgent`, and queues datagrams of other types to an instances of `DatagramConnection`. If there is no instance that matches the source address of the datagram, the component will instantiate and implicitly establishes a connection. The `DatagramConnection` represents the connection to a peer and will be terminated if there are no more incoming datagrams of the corresponding peer. Each instance handles flow control as well as error correction, segments outgoing bundles, and reassembles incoming data to bundles.

Flow Control & Error Correction

The generic convergence-layer implementation has only two requirements to lower layers. Bytes must be arranged as frames and a frame is either correct or not received at all. A channel with the latter feature is called packet erasure channel and generally assures this by adding Cyclic Redundancy Checks (CRCs) in the lower layers. If this feature is not provided by lower layers, it is necessary to guarantee that with an appropriate mechanism implemented in the *datagram service*. For example, this is necessary in the Serial-Link Service.

Essential for reliable transmissions, based on unreliable datagram-based channels, is an approach to compensate errors. If enabled, the datagram convergence-layer detects lost datagrams by expecting an ACK for each segment. If the retransmission timer expires before the ACK is received, the transmission of the previous segment is repeated up to maximum retry limit. The time-out value is calculated using the Exponential Weighted Moving Average (EWMA) algorithm based on the measured RTT between a segment and the corresponding ACK and increased on each retransmission using an exponential back-off.

To detect a lost segment, the sequence number range contains at least two numbers. Even if stop-and-wait is used as flow control, a sequence number range with at least two elements is mandatory for reliable detection of lost segments. Imaging a situation where the retransmission timer expires before the ACK for a packet arrives, but the ACK is not lost. The error correction algorithm would retransmit the previous segment a second time, which will result in a second ACK. Then the ACK related to the initial packet arrives, is accepted as confirmation of the retransmission, and triggers the transmission of the next segment. Now, the ACK of the

retransmission is received and is wrongly assigned to the second segment. Additionally, the required measurement of the RTT will fail accordingly. The same principle is true for the sliding-window approach. In general, the sequence number range must be twice the size of the number of not acknowledged segments, or the sliding window size.

Additional to confirming a received segment, a bundle node can reject a bundle by sending a NACK for any of its segments. A rejection of a bundle is considered as permanent action and indicates that the bundle node is not interested in the bundle due to any existing policy. Using an additional flag for a temporary NACK, the bundle node signals that it is not capable at the moment to process the incoming data, but that this may change in the future. Bundles rejected by such a NACK are queued for a later retransmission.

UDP Datagram Service

UDP provides a packet erasure channel using an attached checksum at the end of the header. Additionally, it encapsulates up to 65 535 bytes into datagrams restricted by the MTU of the underlying layer. The UDP Datagram Service utilizes the standard UDP/IP stack to send and receive datagrams. Those which are addressed to all neighbors are transmitted using the multi-cast feature of IP. By confining to IP version 6, we can benefit from automatically assigned link-local addresses. These are available without a central entity, such as a Dynamic Host Configuration Protocol (DHCP) server, and are thus best suited for ad-hoc communication. This datagram service sets the flow control option to sliding window and defines a sequence number range with 16 elements. This way it is possible to encode the sequence number of a segment into only 4 bits and allows 8 unacknowledged segments at the same time. The initial time-out for an ACK is set to 50 ms, because we expect that all communication partners are directly connected to each other and the expected average RTT should be really small in such a case. Finally, the retry limit is set to five attempts. The MTU respectively the maximum payload size of each segment is defined using a runtime parameter. Since UDP/IP is used on top of different layers, it is necessary that the MTU is specified during the system set-up.

If the generic datagram convergence-layer calls one of the `send()` methods of the datagram service, the parameters and payload must be encoded into a single data frame. The format is shown in Figure 4.3 and starts with 1 byte for the type indicator. The different types and the corresponding values are shown in table 4.1. Flags are encoded in 4 bits, while each of them indicates a property of the current message. Table 4.2 shows the assigned bits to indicated the first or last segment. The *Temporary* flag is used in combination with the NACK message type to indicate

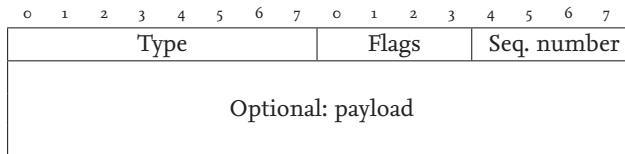


Figure 4.3: Message Format of the UDP Datagram Service

that the bundle node should retransmit the bundle after some delay. The parameters define a sequence number range of 16 elements. These are encoded into 4 bits in the same byte as the flags. The optional payload is only present if the message type is set to *Segment*. The length is not explicitly encoded as part of the message header. Instead the underlying UDP layer provides a payload length which is used to determine the remaining payload length of the segment.

Message type	Value
Broadcast	0x01
Segment	0x02
ACK	0x04
NACK	0x08
Reserved	(other)

Table 4.1: Message type encoding in UDP Datagram Service

Message flag	Flagged bit
Last segment	0x01
First segment	0x02
Temporary	0x04
Reserved	(other)

Table 4.2: Message flags encoding in UDP Datagram Service

IEEE 802.15.4 Datagram Service

Since a complete UDP/IP stack is not very efficient on low-power devices, a datagram convergence-layer may skip the transport as well as the network layer and uses the data-link layer directly to connect with other bundle nodes. An implementation using that strategy is the datagram service implementation based on

IEEE 802.15.4. This technology is a sub-layer of ZigBee [Zig14] and is primarily deployed as low-power radio of sensor nodes.

Each IEEE 802.15.4 frame consists of a maximum of 127 bytes. The header is typically 12 bytes long and identify stations as well as subnets also known as Personal Area Networks (PANs). A Frame Check Sequence (FCS) at the end of the frame adds the characteristics of a packet erasure channel. The standard covers addressing of unicast destinations as well as broadcasts to all stations in the same PAN. The optional layer 2 acknowledgments, which must return within 864 µs, make the channel more reliable for unicast transmissions. Considering the available payload of 115 bytes, the need for segmentation of bundles is even more clearer.

This datagram service sets the flow control to sliding window and defines a sequence number range with 4 elements. This way, it is possible to encode the sequence number of a segment into only 2 bits and allows 2 not acknowledged segments at the same time. The initial time-out for an ACK is set to 2000 ms, because we expect communication with sensor nodes which are potentially slow in processing the data frames. Finally, the retry limit is set to five attempts and the MTU respectively the maximum payload size of each segment is set to 115 bytes.

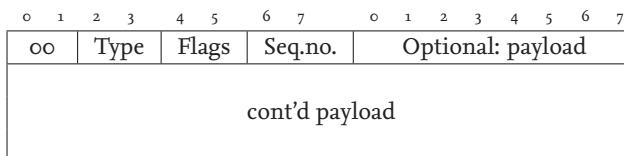


Figure 4.4: Message Format of the IEEE 802.15.4 Datagram Service

If the generic datagram convergence-layer calls one of the `send()` methods of the datagram service, the parameters and payload must be encoded into a single data frame. The format is shown in Figure 4.4 and starts with 2 bit compatibility flags. These bits are used to allow co-existence with other protocols based on 6LoWPAN such as IPHC [HT11] and HC1 [MKHC07]. Both set the first bits of each frame to 01. To avoid misinterpreting 6LoWPAN datagrams as bundles, nodes implementing this convergence-layer set the first two bits to oo. Receivers have to check for this bit combination to dispatch datagrams to the proper stack. The next two bits are used for the type indicator. The different types and the corresponding values are shown in table 4.3. Flags are encoded in 2 bits. Table 4.4 shows all valid combinations to indicated the first or last segment. The combination to mark a first segment is also used to indicate the *Temporary* property of a NACK message. The parameters define a sequence number range of 4 elements. These are encoded into

the bits 6 and 7. The optional payload is only present if the message type is set to *Segment*. The length is not explicitly encoded as part of the message header. Instead the underlying IEEE 802.15.4 layer provides a payload length which is used to determine the remaining payload length of the segment.

Message type	Bits
Broadcast	10
Segment	01
ACK	11
NACK	00

Table 4.3: Message type encoding in IEEE 802.15.4 Datagram Service

Message flag	Bits
First segment / Temporary	10
Last segment	01
Middle segment	00
Single segment bundle	11

Table 4.4: Message flags encoding in IEEE 802.15.4 Datagram Service

This encoding scheme allows us to reduce the necessary header and thus the overhead to 1 byte per frame. The drawback of this approach is the quite limited sliding window with only two frames in-flight. In regard of the limited MTU, this is a reasonable design decision.

4.10.5 File Convergence-Layer

A non-traditional kind of network, but in fact the best example for a terrestrial delay- and disruption tolerant networking, is known as »Sneaker-net«. This kind of network uses removable media such as magnetic tape, floppy disks, compact discs, USB flash drives (thumb drives, USB stick), or external hard drives to move data from one computer to another. The convergence-layer presented in this section is based on such an approach and considers a removable medium as passive bundle node. The necessary logic to manage the data on this media is actually implemented in the active bundle node connected to the medium. Figure 4.5 shows how the medium, named *node D*, moves between the other nodes A, B, and C. Each time the node D is connected to another node, the active node scans all files on the medium and exchanges bundles accordingly to the configured routing protocol.

In this set-up, node A has the opportunity to deliver bundles addressed to node B by using node D as intermediate hop.

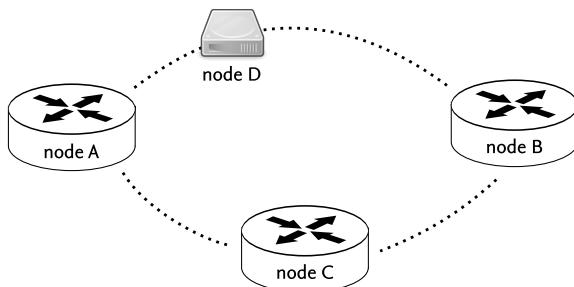


Figure 4.5: Sneakernet scenario using a DTN

As in each convergence-layer not based on IP, some sort of discovery to recognize the passive bundle node instances is necessary. We use the notification capabilities of the Linux kernel to get informed as soon as the file-system changes and a connected medium is ready to get discovered. The convergence-layer scans for a configuration file with the name `.dtndrive` on the medium which contains the EID of the passive bundle node and a relative path on the medium's file-system which is dedicated for DTN related files. This way, it is possible to keep the medium operational for any standard use-case but extend it by DTN capabilities. Finally, the discovered credentials of the passive bundle node are added to the internal list of neighboring nodes.

The routing exchange mentioned in Section 3.6.1 is required for many dynamic routing protocols and usually performed by exchanging bundles between two bundle nodes. Since the medium is passive and can not reply to received bundles, the convergence-layer intercepts the transmission of those special bundles and generates a reply bundle as if the exchange has been done with a remote peer. The summary vector returned in the reply is generated on demand by iterating over all bundle files on the medium.

Outgoing bundles queued to the convergence-layer are simply written to unique files within the dedicated path on the medium. The receive procedure is not based on the routing extensions mentioned in Section 4.9. Instead the convergence-layer considers all bundles on the medium as received, except those already known.

4.11 Applications

Compared to traditional networking applications, a DTN-aware application needs to tolerate message delivery delays. Those delays do not have to be, but may be present while exchanging messages with other endpoints. The best way to handle such delays is to use asynchronous operations as much as possible and avoid multiple message round-trips for a single operation. Thus, all data for a transaction should be packed into a single bundle. If that design recommendation has been followed, an application can expect that the message will be definitely delivered if there exists a path and the resources on the path are not exhausted. In any case, the application does not have to worry about disconnections in the network.

While an application is connected to a bundle node, its state is stored within a registration. If the application disconnects, the registration gets destroyed. To keep a registration alive and resumable while the application is closed or at least disconnected, it can be marked as permanent. Those registrations allow applications to poll for new bundles instead of being active all the time. If the application has processed all waiting bundles, it can exit and free its resources until the next poll interval.

The bundle node provides several primitives to applications via an API. The basic primitives allow applications to register to endpoints, send bundles, and receive bundles. Using extended primitives an application can acknowledge the delivery of a specific bundle, dynamically unregister from endpoints and register to another one, switch a registration into a persistent mode, and resume a persistent registration.

In the architecture of IBR-DTN, there exist several ways to attach applications to the bundle node. In this section, we are going to explain how applications can interact with the bundle node.

4.11.1 Embedded Applications

The first category of applications to mention here are the embedded services. These applications are derived from the `AbstractWorker` class and listen persistently to incoming bundles after they are instantiated together with the bundle node. This kind of integration has a tiny processing overhead and gives an application direct access to internal structures of the bundle node which is useful for management functionalities. The standard set of the embedded services consists of the following applications.

- `CapsuleWorker`

This application is an endpoint for Bundle-In-Bundle Tunneling explained

in Section 4.12.2. Encapsulated bundles are extracted and handled as they were received through a convergence-layer.

- **Debugger**

The debugger application just prints meta-data of incoming bundles to the standard output.

- **DevNull**

This application is a Null-endpoint application for debugging purposes. Bundles received by this endpoint will be immediately delivered, but no further action takes place.

- **DTNTPWorker**

The implementation of the time-synchronization protocol presented in Section 5.2.6 uses this application to exchange bundles related to clock synchronization between two bundle nodes.

- **EchoWorker**

The echo application replies to incoming bundles with an echo of the same payload. Further, it copies specific extension blocks to support network management tools as dtntracepath.

- **HandshakeEndpoint**

The component HandshakeEndpoint is the endpoint application for the routing exchange presented in Section 3.6.1. This application initiates the routing exchange and responds to incoming requests.

4.11.2 Application Interface

Applications not integrated into the bundle node need to access the application primitives. The library `iBrdtn` mentioned in Section 4.4 provides classes to create and manage an API connection based on TCP or unix domain sockets. The API is generally based on human readable plain-text, but allows a client to switch to different protocols which may follow another encoding scheme. When connecting to the bundle node using telnet, the API shows a welcome message which contains the software and API version. Listing 4.13 shows the output of a telnet session connected to the API. The bundle node prints out the welcome message (line 5) and the client changes the protocol to »extended« which is confirmed by the bundle node.

Each connection has an attached `Registration` object. It allows the client to manage endpoint subscriptions, send and receive bundles, and provides a filter to

Listing 4.13: Connecting to the API using telnet

```
1 $ telnet localhost 4550
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^J'.
5 IBR-DTN 0.13.0-346 (build 0afc778) API 1.0
6 protocol extended
7 200 SWITCHED TO EXTENDED
```

prevent bundles from being delivered twice. By default, a registration is volatile and gets destroyed if the API connection is terminated. To prevent this, the client can set the registration to persistent. This way, the registration stay active even if the client disconnects. Once the client is connected again, the client can resume the suspended registration using its unique identifier.

As explained before, the API lets the connecting client choose between different protocols. Each protocol implementation is encapsulated into a class derived from `ProtocolHandler`. As shown in listing 4.13, the client selects the protocol to switch to by sending the keyword »protocol« followed by the specific protocol keyword. The following list describes the available protocols.

■ `BinaryStreamClient`

This protocol is based on the TCP-CL protocol defined in [DOP14] and explained in Section 4.10.3. Using the `ibrtdn` library, the client can construct bundles the same way as within the daemon implementation and push them to the bundle node. The bundle node processes the incoming bundles similar to if they were received from another bundle node, except that the unique bundle identifier and the source endpoint is replaced on reception. Other on the bundle node applied features like compression, confidentiality, and signatures are requested using extended processing flags of each bundle. A client connected using this protocol, registers to endpoints before the protocol is switched. Once the connection is established, the functionality is limited to send and receive bundles. The protocol does not provide any other operation to change a session setting or even endpoint registrations. If that is required, the `ExtendedApiHandler` is a better candidate. Beside that drawback, the streaming approach has almost no overhead and processing received bundles as well as delivering them to applications is pretty fast.

■ ExtendedApiHandler

Since the binary TCP-CL protocol is quite complex to implement, this protocol provides a simple but powerful API based on a plain-text syntax. Using commands terminated by line-break characters, a client can manage its registered endpoints, change the bundle encoding, list all neighbors visible to the bundle node, suspend and resume a registration, query the EID of the bundle node, and manage bundles. A bundle queued for delivery is notified to the client using a small message instead of directly pushing the whole bundle to the client. After the notification, the client can load the bundle into a register which allows the client to analyze the bundle without receiving it completely. For example, the client may receive the header first and then decides that the bundle is not of interest. Then it can simply issue the command to drop the bundle and declare it as delivered even if the bundle content was never transmitted to the client. Bundles to send are also pushed into the register first and once the bundle is complete, the client can issue the command to send it.

■ OrderedStreamHandler

The reliable and ordered data-streaming approach presented in Section 3.9.2 is implemented as API protocol. As alternative way to handle the data-streaming in a library, this protocol implementation handles the necessary segmentation and re-ordering. A connected client simply specifies the parameters of the stream and initiates the virtual connection. Once the protocol is in the *connected* mode, all data written to the connection is transparently transmitted to the specified destination. Since the streaming is bi-directional the receiver does the same as the sender to initiate the connection.

■ ManagementConnection

The management API provides access to management routines of the bundle node. These allow a program to monitor the bundle node and change its configuration during its run-time. Available calls are:

- list all visible neighbors
- list and clear all bundles in the storage
- add or remove static connections
- add or remove interface bindings of convergence-layers
- add or remove static routes
- enable or disable neighbor discovery

- query routing data (e.g. PRoPHET's probability map)
- query statistic data
- query recent logging data

- **EventConnection**

This is a passive protocol and does not accept any commands. Instead a connected client gets notified as soon as an event is raised. These messages include all details of the event and allow a client to react to them without the need to poll the state of the bundle node.

- **ApiP2PExtensionHandler**

As explained in Section 3.6.7, there exist convergence-layers which need to set-up some kind of channel between two bundle nodes before any transmission is possible. Using this protocol a program, which controls the underlying technology, can announce discovered peers and receives connection set-up request from the bundle node, if a set-up is required.

4.11.3 Tools

As mentioned in Section 4.4, IBR-DTN ships with a `tools` package. It includes several command-line programs to manage and test the bundle node instance. Additionally, there exist some programs to script small applications. For performance reasons, all applications use the client library contained in the `ibrdtm` package. This library utilizes the binary protocol provided by `BinaryStreamClient`. In this section, we will introduce and explain each application of this package.

dtnping

This tool is used for network maintenance and testing purposes. It sends a bundle to a given endpoint and waits for a reply with the exact same payload. The payload contains a sequence number encoded in the first four bytes and fills the remaining part of the payload with a string pattern. The tool measures the RTT of each probe, i.e. the delay between the transmitted and the received bundle. Listing 4.14 shows an example output of the tool. By default all probes carry 64 bytes payload and have a life-time of 30 s. Thus, if the response is not received within twice the life-time, the bundle is considered as lost and the next probe will be transmitted. The short life-time is well suited for statically connected networks and testing of direct connections. If the expected delivery delay is larger than 30 s, this parameter must be increased by the user. Another issue is related to the dependency on synchronized clocks mentioned in Section 4.13.1. If the clocks of two peers are more

than 30 s apart, either the probe request or the response will be dropped due to expiration policies. To test if missing clock synchronization is the reason for a failed probe, it is reasonable to increase the life-time to a higher value depending on the expected offset between the node's clocks.

Listing 4.14: Example usage of dtncping

```

1 $ dtncping dtn://syrah/echo
2 ECHO dtn://syrah/echo 64 bytes of data.
3 64 bytes from dtn://syrah/echo: seq=1 ttl=30 time=2.51 ms
4 64 bytes from dtn://syrah/echo: seq=2 ttl=30 time=3.01 ms
5 64 bytes from dtn://syrah/echo: seq=3 ttl=30 time=3.04 ms
6 64 bytes from dtn://syrah/echo: seq=4 ttl=30 time=3.37 ms
7 ^C
8 --- dtn://syrah/echo echo statistics ---
9 4 bundles transmitted, 4 received, 0.00% bundle loss, time 3.27 s
10 rtt min/avg/max = 2.51/2.98/3.37 ms

```

dtntracepath

This tool is used for network maintaining and testing purposes. It sends a bundle to a given endpoint and requests a delivery report. The delay between the transmitted bundle and the received reports is displayed as a list. If the bundle is sent to an echoing endpoint, the tool will also display the received response and its RTT. Additional to the delivery report, the tool optionally request reports for deletion, receptions, and forwards as defined in [SB07]. Using the TEB introduced in Section 3.9.4, this tool can also record the path of the bundle. For that purpose, a special feature of the embedded echo application is required. The echo application copies the TEB of the request to the response. This way the path of the request is merged with the path of the response. Listing 4.15 shows an example trace with enabled path tracking and requesting a BSR for each received bundle. The path is displayed in the lines 9-20. The destination is reached in line 14 and takes a slightly different back-route over dtn://android-d3b56b84.dtn instead of dtn://blueberry.

Listing 4.15: Example usage of dtntracepath

```

1 $ dtntracepath -pr dtn://android-8f98e4f5.dtn/echo
2 TRACE TO dtn://android-8f98e4f5.dtn/echo
3   1: dtn://syrah                                reception  1.12 ms
4   2: dtn://matrix                               reception  10.46 ms
5   3: dtn://cloud.dtnbone.dtn                  reception 135.65 ms
6   4: dtn://quorra.ibr.cs.tu-bs.de            reception 139.67 ms

```

```

7   5: dtn://blueberry           reception  465.92 ms
8   6: dtn://android-8f98e4f5.dtn/echo    ECHO      712.47 ms
9     # dtn://syrah
10    # dtn://matrix
11    # dtn://cloud.dtnbone.dtn
12    # dtn://quorra.ibr.cs.tu-bs.de
13    # dtn://blueberry
14    # dtn://android-8f98e4f5.dtn
15    # dtn://android-d3b56b84.dtn
16    # dtn://blueberry
17    # dtn://quorra.ibr.cs.tu-bs.de
18    # dtn://cloud.dtnbone.dtn
19    # dtn://matrix
20    # dtn://syrah
21  7: dtn://android-8f98e4f5.dtn           delivery  907.54 ms
22  8: dtn://android-e231d43c.dtn           reception  7.48 s

```

dtnconvert

This tool is capable in reading and writing bundles using the binary encoding specified in [SB07]. Basically, it is used to analyze traffic or bundles stored in files. Additionally, it can be used to inject bundles into a storage for experiments. Listing 4.16 shows an example output of the tool. The file bundle.dat is actually a file of the standard file-based storage and piped into the standard input of the tool. The tool is started with the parameter »r« to indicate that it should read encoded input data and print out the header information.

Listing 4.16: Example usage of dtnconvert

```

1 $ cat bundle.dat | dtnconvert -r
2 flags: 144
3 source: dtn://syrah-1/xxudbXjLNcTQGHBQ
4 destination: dtn://test/test
5 timestamp: 450256166
6 sequence number: 1
7 lifetime: 3600
8 payload size: 12

```

dtnsend

The tool dtnsend can send bundles with a given payload to an endpoint. This is useful for testing as well as for scripted applications. In combination with dtnrecv or dtnttrigger it is possible to build simple applications.

The listing 4.17 shows an example usage of this tool. Here, the tool is used to transmit the data in the file `payload-64k.dat` to the endpoint `dtn://syrah/app`. Additionally, the command-line parameter `sign` is set. It requests a signature for this bundle which will be attached by the bundle node.

Listing 4.17: Example usage of `dtnsend`

```
1 $ dtnsend --sign dtn://syrah/app payload-64k.dat
2 Transfer file "payload-64k.dat" to dtn://syrah/app
```

dtnrecv

`dtnrecv` is used to receive the payload of a single or multiple bundles. It writes the data into files or to the standard output. The listing 4.17 shows the output of the tool after receiving a single bundle. By setting the command-line parameter `name`, the tool registers to the bundle node's local endpoint with the application name as suffix. In this case, the bundle node's local endpoint is `dtn://syrah`. Therefore, the endpoint of the application will be `dtn://syrah/app`. The payload of the bundle is written into the file `received.dat`. If the command-line parameter `file` is not specified, the tools writes the payload to the standard output. In combination with `dtnsend`, it is possible to transfer data between two or more hosts through a DTN.

Listing 4.18: Example usage of `dtnrecv`

```
1 $ dtnrecv --name app --file received.dat
2 Wait for incoming bundle...
3 Bundle received (1).
4 done.
```

dtnstream

This tool accepts continuous data input by reading the standard input and generates bundle chunks as described in Section 3.9.2. In the example, the sender (listing 4.19) retrieves a continuous mp3 radio stream using `wget`. The output is piped through `pv` to show the progress and finally into `dtnstream`. The option `--c` specifies the minimal size of each chunk. The tool will wait in this example until at least 64 kBByte are ready to transmit. The option `-f` enables the adaptive approach to determine when the next bundle should be released by requesting a delivery report for each chunk. As last, the option `-d` defines the destination endpoint for the stream.

Listing 4.19: Example usage of `dtnstream` (sender)

```

1 $ wget -q -O - http://player.ffn.de/ffn.mp3 | pv | \
2   dtnstream -c 64000 -f -d dtn://syrah/radio
3 1,42MB 0:01:17 [22,9kB/s] [           <=> ]
```

In this case, the receiver is an instance of `dtnstream` (listing 4.20) which registers to the singleton endpoint of the bundle node with the suffix specified by the »`s`« parameter. The received data stream is written to the standard output of the application and piped through `pv` into `mpg123` which decodes and plays the mp3 audio stream.

Listing 4.20: Example usage of `dtnstream` (receiver)

```

1 $ dtnstream -s radio | pv | mpg123 -q -
2 1,04MB 0:01:05 [15,8kB/s] [           <=> ]
```

`dtntrigger`

The tool `dtntrigger` offers a simple way to script applications without the need to compile any code. It reduces the API offered to applications to the reception of the bundle's source and payload. In return, the application does not have to deal with the connection to the bundle node. Instead, `dtntrigger` connects to the bundle node, registers to an endpoint, and writes the payload of each received bundle into a temporary file. After the file is written, the tool executes an external program or script with the bundle's source and the temporary payload file as parameters.

To clarify how easy it is to build applications using `dtntrigger`, we want to present a chat bot example. As text source we use a stateless Eliza program [Wei66] written in python. It accepts any text via the standard input and response to it with a sentence written to the standard output. Listing 4.21 shows how that program reacts to a »Hello« message.

Listing 4.21: Example usage of the eliza program

```

1 $ echo "Hello" | python ~/eliza.py
2 Hello... I'm glad you could drop by today.
```

To make that bot reachable via the bundle protocol, we create a simple script (listing 4.22) which accepts two parameters. The first parameter is the sender of the incoming bundle. The second one is a file containing the payload of the bundle. To see what happens, we print the bundle's source and the message to the console (line 7). In line 8, we pipe the content of the received file into the Eliza script and store the response in the variable `ANSWER`. The actual answer is sent in line 11. In order to do that, the content of the variable `ANSWER` is piped into the `dtnsend` tool with

the source endpoint set to »chat«, the life-time set to 2 hours, and the destination set to the source of the origin bundle.

Listing 4.22: dtntrigger scripts for the eliza chat bot

```
1 #!/bin/bash
2 #
3
4 SENDER=$1
5 FILE=$2
6
7 echo "${SENDER}: `cat ${FILE}`"
8 ANSWER='`cat ${FILE} | /usr/bin/python ~/eliza.py`'
9 echo "Eliza: ${ANSWER}"
10
11 echo -n ${ANSWER} | /usr/bin/dtnsend --src chat --lifetime 7200 ${SENDER}
```

Assuming the script is named `bot.sh`, we can run the `dtntrigger` program as shown in listing 4.23. It requires three parameters: The singleton application endpoint, the script interpreter, and finally the script to execute on every received bundle.

Listing 4.23: Execution of the chatbot application based on dtntrigger

```
1 $ /usr/bin/dtntrigger chat /bin/bash bot.sh
```

dtnoutbox

The tool `dtnoutbox` observes a given folder for file changes. If new files appear or a file has been changed, the program will put these files into a tar archive and send it to a pre-defined destination. The application is started with three parameters: The application endpoint, a folder to observe, and the destination endpoint. Optionally, an interval and a number of rounds can be specified which are otherwise set to default values. Once started, the tool observes a given directory by periodically checking the file-names and the file-sizes. If new files are not changed for a number of rounds, then these are sent to the destination encapsulated in a tar archive.

The example in listing 4.24 shows a sample output of this application. Here, the tool recognizes the file »file1.png« and waits for some time to check if the file gets

altered. In line 4, a tar archive is created with the as stable considered file and sent away. As next, another file »file2.png« appears in the folder and is sent away.

Listing 4.24: Example usage of dtnoutbox

```
1 $ dtnoutbox outbox outbox-folder dtn://syrah/inbox
2 -- dtnoutbox --
3 file found: file1.pdf
4 files sent: file1.pdf
5 file found: file2.png
6 files sent: file2.png
```

dtninbox

dtninbox is the counter-part to **dtnoutbox**. It receives bundles which contain a tar archive and extracts the content to a given directory. Both tools in combination allow the user to set-up a folder which is mirrored to a remote one. As shown in the listing 4.25, the tool requires two parameters: The application endpoint and the folder to extract the archives to. Each received bundle is displayed using its unique identifier consisting of the creation time-stamp, creation time-stamp sequence number and the source EID.

Listing 4.25: Example usage of dtninbox

```
1 $ dtninbox inbox inbox-folder
2 received bundle: [455025198.3] dtn://syrah/outbox
3 received bundle: [455025253.1] dtn://syrah/outbox
```

dtntunnel

Using **dtntunnel**, a user can set-up an IP tunnel between two endpoints. Each IP packet is encapsulated into a single bundle and then forwarded to the remote peer. That allows legacy IP-based applications to profit from a DTN. While TCP is not designed to tolerate really long delays, the reasonable scenarios for this kind of tool are quite limited. If the network is well-connected or the application can tolerate long delays, this kind of tunnel allows applications to profit from the overlay network and utilize heterogeneous links which are not capable to transport IP packets.

In listing 4.26, a tunnel endpoint is set-up. The tool requires several parameters: »s« specifies the application endpoint, »d« the name of the local interface, and »t«

the remote tunnel endpoint. Once the application is started, the user has to set-up the IP stack. `dtntunnel` displays a help text to ask the user to do that (line 7-9).

Listing 4.26: Example usage of `dtntunnel`

```

1 $ sudo dtntunnel -s tunnel0 -d tun0 -t dtn://syrah/tunnel1
2 IBR-DTN IP <-> Bundle Tunnel
3 Local: tunnel0
4 Peer: dtn://syrah/tunnel1
5 Device: tun0
6
7 Now you need to set-up the ip tunnel. You can use commands like this:
8 # sudo ip link set tun0 up mtu 65535
9 # sudo ip addr add 10.0.0.1/24 dev tun0
10
11      up:          0.00 kB/s    down:          0.00 kB/s

```

4.12 Protocol Extensions

In this section, we will present a bunch of important extensions we have been integrated into IBR-DTN in order to deal with the challenges discovered during the development. In contrast to Section 3.9, the extensions presented here are proposed and specified by other researchers and engineers.

4.12.1 Age Extension Block

As later discussed in Section 4.13.1, the dependency on synchronized clocks is one of the major drawbacks of the BP. To mitigate the dependency on this, an IRTF draft [BFB10] defines an AEB which is able to track the age of each individual bundle. This allows a bundle node to expire bundles, even if the clock of the source is considered as invalid and it is not possible to set the creation time-stamp to a correct value.

To indicate that the AEB should be considered to expire a bundle, the creation time-stamp in the primary bundle block is set to zero and an AEB is placed before the payload block. To ensure correct expiration of every bundle fragment, this block must be replicated in every fragment, thus the corresponding flag is set in the block processing control flags. The block format is quite simple and adds only a single SDNV containing the age of the bundle in microseconds (see Figure 4.6).

To figure out how to track the age correctly, we have to consider the life-cycle of a bundle instance. A new bundle, whether generated on the sender or received from

Block type	Block proc. ctrl. flags (*)	Block length (*)
Age in microseconds (*)		

(*) marked fields are encoded as SDNV

Figure 4.6: Age Extension Block Format

another peer, leads to an instance of a `Bundle` object in the first step. The instance is then forwarded through the event-system to the router which processes the new bundle and stores it in the storage system. Later, a routing extension may select a bundle and queues it to a convergence-layer for transmission to another bundle node. The convergence-layer loads the bundle from the storage system, applies additional processing if necessary (e.g. authentication extension of the BSP), and finally writes the encoded bundle data to the underlying networking layer. Additionally, a bundle may spend a lot of time while it is in-flight between two bundle nodes. This time is typically defined as propagation delay. Based on this life-cycle, there exist three phases to track. The time the bundle is stored in the storage, the duration it takes to process the bundle in the convergence-layer, and the time while the bundle is transmitted but not received by another bundle node. All these phases should increment the age of a bundle.

As any other extension block, the AEB is implemented as specialization of the `Block` entity. The AEB instance is created during the Deserialization procedure or if a new bundle with an AEB is created. Using the constructor and the serialization methods of the `Block`, it is possible to track at least the time a bundle is being processed by simply storing the monotonic time-stamp of the system in a variable and calculating differences. If the storage is memory-based only, this approach also covers the time a bundle is stored, because this kind of storage only manages the `Bundle` instances in lists without destroying them. Other storages have to implement additional age tracking approaches. For example, the file-based storage uses the time-stamps of the files to alter the value of the age block when a bundle is restored from a file. The latter phase, while the bundle is in flight, is hard to determinate and depends on the specific underlying network. Thus, it is the task of the convergence-layer implementation to alter the AEB, if a bundle was received over a channel with a known transmission and propagation delay. All the convergence-layers presented in Section 4.10 do not implement such a mechanism, because the required delay values are not determined.

The drawback of the approach using the AEB is that the bundle node loses its ability to assign unique identifiers to bundles without holding a persistent state. Since the creation time-stamp is always set to zero, bundles are only distinguished by their creation time-stamp sequence number. Thus, it is required to store the highest assigned creation time-stamp sequence number in a persistent storage to always know which is the next number to assign. If such a persistent state is not feasible, a reset of the register with the last creation time-stamp sequence number may lead to new bundles being rejected by other bundle nodes due to their duplicate checking. Since there is no known solution for that issue, we propose to add a time-synchronization scheme as described in Section 5.2, but fall-back to AEB augmented bundles if the synchronization fails or if the bundle node has been restarted and a synchronization did not happen so far. To keep basic mechanisms working even if there are two diverging clocks, the routing exchange and the embedded application for time-synchronization always utilize AEB augmented bundles for their approach. Since the life-time of these bundles is quite small and the distribution is limited to one hop only, the risk of a wrong duplicate is relatively small.

4.12.2 Bundle Encapsulation

The Internet-Draft *Delay-Tolerant Networking Bundle-in-Bundle Encapsulation* [SDS09] specifies an approach to encapsulate multiple bundles into an enveloping bundle. Doing this has several use-cases as specified in the Internet-Draft.

Obviously, this approach allows to set-up a tunnel through a DTN which is useful if intermediate nodes are not able to route a bundle because the destination is not known in the routing domain. Additionally, it is possible to encrypt bundles transmitted through such a tunnel. Different to the BSP, the entire bundle including the source and destination address is handled confidential and not exposed to intermediate nodes.

The second use-case is related to custodial retransmissions. These may traverse multiple hops in a DTN and are necessary if a node reports a custody failure. In case the bundle affected by the failure is addressed to a singleton destination, it can simply get resent. But if the bundle is addressed to a group endpoint, the retransmission would lead to a redistribution of the bundle to many nodes which may already received it. Bundle encapsulation helps to avoid such redundant transmissions by tunneling the origin bundle using an encapsulating bundle addressed directly to the node which reported the custody failure.

The last use-case mentioned in the Internet-Draft describes a scenario where a node requests some content which is already present in the storage of a caching

bundle node. Since the content was previously addressed to another endpoint, the destination of the bundle containing the requested content differs from the requesting node. Thus, the caching node has to alter the destination of the origin bundle or encapsulate the bundle into another one with the requesting node as destination. In case, the bundle's integrity is protected using a Payload Integrity Block (PIB), altering the destination EID is not an option and the encapsulation is the only way to forward the bundle to the requesting node.

Block type	Block proc. ctrl. flags (*)	Block length (*)
Number of Encapsulated Bundles (*)		
Bundle Offsets List (opt.)		
Encapsulated Bundle 1		
Encapsulated Bundle 2		
...		
Encapsulated Bundle n		

(*) marked fields are encoded as SDNV

Figure 4.7: Payload Block Format of an encapsulating bundle

Bundles encapsulated into another are encoded using the common BP format as they were transmitted using a convergence-layer. The encoded data is placed as usual in the Payload Block of the bundle. There is no need for an additional extension block. Figure 4.7 shows the Payload Block Format of an encapsulating bundle. As usual the Payload Block starts with a type which is set to 1. The following processing flags only indicate if the Payload Block is the last block of the block chain or not. The Block length contains the remaining number of bytes of this block. The payload encapsulating bundles starts with the number of bundles contained in this block, encoded as SDNV, and followed by a list of bundle offsets. This list is only present if more than a single bundle is encapsulated and contains the byte positions of each single bundle except the first one which always starts at the first byte. Finally, all encapsulated bundles are concatenated in the encoded byte representation.

IBR-DTN implements the Bundle-In-Bundle Tunneling [SDS09] approach as exit-node only. By default, an integrated endpoint application registers to the suffix `bundle-in-bundle` and unwrap all enveloped bundles. Those bundles are handled as they were received by one of the convergence-layers. Not part of the IBR-DTN is the encapsulation procedure. There is no component which

encapsulates bundles and forwards them to another bundle nodes. Instead, the library IBR-DTN provides methods to fulfill such a task and the encapsulation itself is left up to applications.

4.12.3 DTN Scope Control using Hop Limits

Under certain conditions it is reasonable to limit the number of hops a bundle may travel. For example, the routing exchange (presented in Section 3.6.1) generates request bundles on each contact with another bundle node. These messages are destined for direct deliver only and a delivery over multiple hops is not reasonable. If the recipient is no longer a neighbor, the requested routing data becomes useless. By adding a hop-limit of 1 hop to the bundles, it is possible to avoid distribution of these through the entire network. Another use-case is the clock comparison of two bundle nodes as done in Section 5.2. To avoid long delays due to multi-hop forwarding, the hop-limit for delay sensitive bundles is also limited to 1 hop. Thus, those bundles are only forwarded if two bundle nodes are directly connected.

Block type	Block proc. ctrl. flags (*)	Block length (*)
Hop count (*)		Hop limit (*)

(*) marked fields are encoded as SDNV

Figure 4.8: SCHL Extension Block Format

In order to track and limit the number of hops a bundle is allowed to travel, the Internet-Draft *DTN Scope Control using Hop Limits (SCHL)* [Fali0] specifies an extension block with the format definition shown in Figure 4.8. The block type is set to 9 and the processing control flags instruct all bundle nodes that this block must be replicated in every fragment. This is necessary to apply the same hop-limit to every fragment made out of this bundle.

In contrast to the Time to Live (TTL) value in IPv4 or the hop-limit in IPv6, the extension presented here does not decrement a value until zero has been reached. Instead, there is a hop-count value incremented on each hop and an additional hop-limit. This allows forwarding nodes to ascertain how many hops bundles have traveled so far. In other words, it indicates how much effort already has been spent to forward the bundle to the current bundle node. That information might be used to affect resource allocation decisions and/or congestion controls.

In IBR-DTN, this extension was integrated into the standard feature set. If an incoming bundle carries a SCHL extension block, its hop-count is incremented

every time it passes the BaseRouter (see Section 4.9). Each routing extension will consider the remaining number of hops to determine if it is sensible to forward those bundles to another bundle node. E.g. a routing extension will skip any bundle with a remaining hop-count of zero and each multi-hop routing scheme which does not deliver bundles to its final destination will skip bundles with a remaining hop-count of 1, since it would be impossible to forward a bundle from there to its destination, if the bundle is not allowed to get further forwarded.

In any case, it is up to the applications to add a SCHL extension block in order to restrict the number of hops a bundle is allowed to travel. IBR-DTN utilizes that feature to prevent multi-hop forwarding of bundles generated by the routing exchange (Section 3.6.1) and the time-synchronization approach (Section 5.2).

4.12.4 Bundle Security Protocol

The *Bundle Security Protocol Specification* [SFWL11] defines additional extension blocks for the BP to augment standard bundles with security extensions and introduces procedures to generate and processes them.

Since the BP is build on top of convergence-layers which adopt a wide variety of underlying technologies to communicate with peer nodes, it is important to stay independent from the underlying network and the associated convergence-layer. For that reason, bundles are only extended or altered according to the base BP to achieve confidentiality, authenticity, integrity, and non-repudiation. The second constrain for the security design is to achieve security with a minimal number of round-trips. Due to the potential high transmission delay, it is not reasonable in DTNs to set-up a secure session using several round-trips at the beginning of each communication. Instead, security enhanced bundles must be self-contained and should carry all the necessary data to apply security procedures.

Figure 4.9 shows the general data-flow of a security enhanced multi-hop transmission through a DTN. The example starts at the »sender« application which simply forwards a plain bundle (1) addressed to the »receiver« application on the bundle node C to its local BPA on node A. As soon as the BPA receives the bundle, it will apply security enhancements according to the local policy. In this case, the bundle is encrypted and signed (2). To apply encryption, the content of the Payload Block is replaced by the encrypted data and a Payload Confidential Block (PCB) is placed in front of the Payload Block. In addition to the secret key, the PCB contains the necessary credentials to decrypt the payload. Additionally, the bundle is signed by placing a PIB in front of all other extension blocks. This block typically contains a cryptographically signed hash value of the bundle.

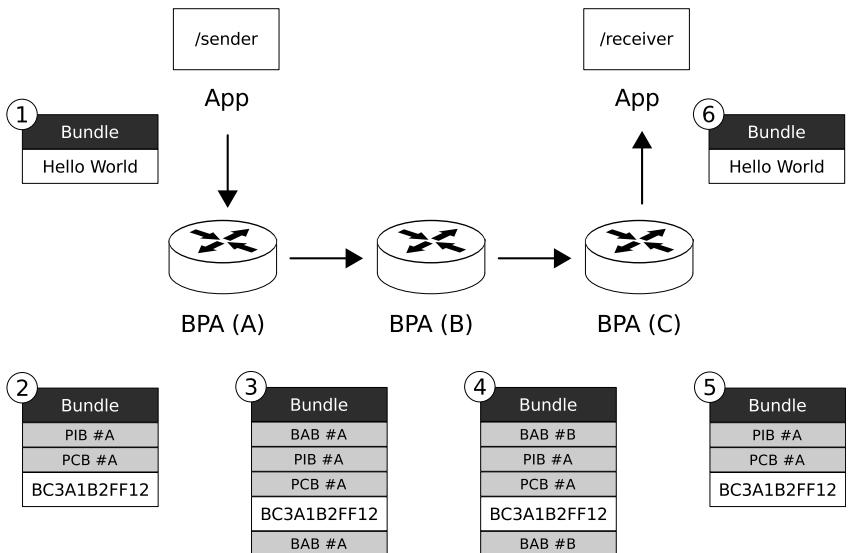


Figure 4.9: General data-flow of the Bundle Security Protocol

In the next step, bundle node A wants to forward the bundle 2 to bundle node B. Since the bundle node B only accepts bundles from authenticated peers, the bundle node A adds a pair of Bundle Authentication Blocks (BABs) to the bundle. The first block announces the authentication and contains the security parameters. The authenticated hash of the bundle is placed in the second BAB. On reception of the bundle 3, the bundle node B verifies the authentication and removes the BABs. Since bundle node B is an intermediate node, it needs to forward the bundle to bundle node C in the next step. In order to do that, the bundle 4 is augmented again with BABs and forwarded to bundle node C which strips and verifies the BABs. Now, the bundle 5 has reached the final destination and has to be delivered to the registered application. Since the security extensions are implemented as part of the BPA, the signature is verified and the payload is decrypted before the bundle 6 is finally delivered to the application.

Each security block has a security-source and a security-destination which may be different to the source and destination of the corresponding primary block. The security-source always refers to the bundle node which added the security extension block and the security-destination points to the bundle node which should be capable in processing it. Referring to the example from above, the PIB and

the PCB of the bundle 2 have bundle node *A* set as security-source and security-destination set to bundle node *C*. In this case there is no difference between these sources respectively destination points of the primary block. Another example is the BAB pair in the bundle 3. The security-source is also set to bundle node *A*, but the security destination may be set to bundle node *B*, because it is in charge of processing the security extension attached by bundle node *A*.

Canonicalization

As seen in the previous example, the security blocks of bundles are altered on its way through a DTN. It is even common that values of the primary block are changed. Depending on the type of protection, such changes are considered as valid or not. Authentication and integrity protection always work the same way. An algorithm generates a hash value out of the data to protect and signs it with a key. To allow bundles to get changed on their path, but still protect non-mutable parts of the bundle, mutable data has to be skipped during the hash generation.

The bundle security protocol defines two types of canonicalization which encodes a bundle into a fixed data-structure. The strict canonicalization protects the whole bundle-data and permits no changes at all to the bundle between its security-source and the security-destination. The mutable canonicalization skips mutable parts of the bundle such as the custodian of the primary block and all reserved processing flags. Additionally, it ensures that the data-structure is always generated exactly the same way with exactly the same order of values, no matter how often the bundle was recomposed on its way.

To implement several types of canonicalization, we make use of the serialization pattern introduced in Section 4.6 and implement the `StrictSerializer` as well as the `MutableSerializer`. Each of them is derived from the `DefaultSerializer` and modifies the way a bundle is serialized according to the *Bundle Security Protocol Specification*. This way, it is easy to pipe the canonicalized form in a very efficient way into an output stream implementation which generates a hash out of the data.

Security Extension Blocks

The *Bundle Security Protocol Specification* adds four different types of security extension blocks. The Bundle Authentication Block (BAB) is typically used between two adjacent bundle nodes to mark a bundle as valid in the network. The Payload Integrity Block (PIB) is primarily used to protect the integrity of a bundle, but since it also contains a signature it is capable in authenticating the bundle and validating its source. To encrypt the payload of a bundle, a Payload Confidential Block (PCB) is added in front of the payload block and the original payload is replaced by the encrypted data. The last block to present is the Extension Security Block (ESB),

which is used to encrypt non-security related extension blocks. To protect an extension block using an ESB, the extension block is encapsulated into an ESB which replaces the original extension block.

Block type	Block proc. ctrl. flags (*)	EID-ref list (opt.)
Block length (*)	Cipher-suite (*)	
Cipher-suite flags (*)	Correlator (*)	
Params length (*)	Cipher-suite params data	
Result length (*)	Security-result data	

(*) marked fields are encoded as SDNV

Figure 4.10: Abstract Security Block Format

Figure 4.10 shows the data structure of the Abstract Security Block (ASB) which is the base for all security related extension blocks. Beside the generic fields for the type of the block, processing flags, an optional EID-reference list, and the block length, each ASB contains a cipher-suite number and additional flags to indicate which parts of the ASB are present. The correlator value is used to indicate that several ASBs correspond together. For example, if a bundle is protected by a pair of BABs, the correlator value in both BABs must be equal. The data-structure of the cipher-suite parameters and the security-result data depends on the cipher-suite specification used to generate the individual ASB. In the ASB format, both fields are specified as generic data container, but both data fields contain a Type Length Value (TLV) set to hold several items of different types.

Cipher-suites

A cipher-suite defines the exact procedure to encrypt, sign, or authenticate a bundle. Typically, the definition includes a cryptographic algorithm and the content of the corresponding security extension blocks. For each use-case mentioned in the *Bundle Security Protocol Specification* one standard cipher-suite is defined. In the following, all standard cipher-suites are summarized.

■ BAB-HMAC

This cipher-suite is used for hop-by-hop authentication. The strict canonized bundle-data is hashed using an Keyed-Hash Message Authentication Code (HMAC) and placed into a BAB at the end of the block-chain. Another BAB is placed in front of the block-chain and defines the parameters used for this authentication.

■ PIB-RSA-SHA256

To protect integrity of the bundle payload, this cipher-suite defines how to sign a SHA-256 hash value made out of the mutable canonicalized bundle-data using the RSA algorithm. The result is placed as PIB right after the primary bundle block. This approach protects parts of the primary bundle block and the payload from being altered on its way from the security-source to the security-destination.

■ PCB-RSA-AES128-PAYLOAD-PIB-PCB

Using this cipher-suite, a bundle node can encrypt the payload and security-related blocks. First, the payload is encrypted using Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) with a key-length of 128-bit and replaced by the encrypted data. GCM ensures that the encrypted payload has the same size as the unencrypted data and avoids this way issues with fragmentation/reassembly and custody transfer mechanisms.

Beside the payload block, existing PIBs and PCBs are also encrypted and replaced by encapsulating ESBs. The AES key is encrypted using the RSA public key of the security-destination and stored in a PCB which is placed right after the primary bundle block.

■ ESB-RSA-AES128-EXT

Blocks other than payload blocks, PIBs and PCBs are encrypted using this cipher-suite. Blocks to encrypt are encapsulated into ESBs and encrypted using the AES algorithm in the Galois/Counter Mode (GCM) with a key-length of 128-bit. The AES key is encrypted using the RSA public key of the security-destination and placed in the security parameters of the replacing ESB.

4.13 Challenges

While implementing the BP and all its previously mentioned extensions, several challenges have been revealed. In this section, we want to discuss some important aspects in more detail.

4.13.1 Time-dependencies

Different to many other network protocols, the BP heavily depends on synchronized clocks. Bundles are time-stamped and expire after some amount of seconds defined by its own header. The reason for this is that bundles are stored on each bundle node until there is an opportunity to forward them. In conventional networks, packets are forwarded or discarded if there is no continuous path. Such ap-

proach would not work in DTNs and for that reason we need another way to purge data which probably never reaches its destination. Another argument is that many applications also depend on some sort of knowledge about time. So it is practical to utilize this also for the networking protocol.

However, the dependency on synchronized clocks is not a trivial assumption. Especially in disconnected environments it is very hard to get distributed clocks synchronized. For that reason, there exist several approaches to mitigate the dependency. The Age Extension Block (AEB) presented in Section 4.12.1 is such an approach. It allows the bundle node to expire bundles even if there are no synchronized clocks, but introduces some restrictions. The AEB reduces the unique identifier of a bundle to the increasing creation time-stamp sequence number. If the bundle node does not remember its last value between reboots, it is not possible to assign unique identifiers. Moreover, tracking of the relative age may be very complicated and is not even feasible in some scenarios. As explained in Section 4.12.1, the bundle is processed, stored, restored, sent, and received. The duration of all these phases have to be tracked in order to determine the relative age correctly. If this is accurate enough, then the introduced error will sum up on the path of the bundle. A case when a relative age tracking is definitely not possible, is a data transmission using a sneaker-net [GCB⁺02]. The *File Convergence-Layer* (Section 4.10.5) implements such an approach which is based on passive storage media. Without time-synchronization among the bundle nodes, it would not be possible to determine the duration spent by a bundle on such a device. For that reason, the AEB approach is not a solution for all cases.

Since two bundle nodes simply can not communicate if they try to exchange bundles with a life-time shorter as the delta between their clocks, IBR-DTN mixes AEB augmented bundles and standard bundles to provide basic functionality, even if the clocks are drifted apart.

4.13.2 Platform-independence

One of the development goals of IBR-DTN is to run on as many platforms as possible. Actually, the software was addressed to an embedded platform with a limited amount of memory and processing power. Since porting software to other platforms is in most of the cases a quite complicated thing, we decide to avoid additional dependencies to libraries, which would reduce the portability, as much as possible. On the other hand, platform-independent development requires some sort of abstraction layer which is usually provided by a library.

Existing libraries for platform-independent programming (e.g. boost [DAR14] or Qt [Hos14]) are quite comfortable and provide support for the three major

operation systems, but they are also heavy in code-base and do not provide sufficient support for the embedded platform OpenWrt. Thus, a specialized implementation is necessary. In IBR-DTN the library package `ibrcommon` hides system-differences by encapsulating common functions into classes with system-independent implementations. It includes all necessary abstractions to build a bundle node which is portable to multiple platforms including Debian/Ubuntu™, OpenWrt, Android™, OS/X®, and Microsoft® Windows® as well as other Linux™ derived platforms.

The heterogeneous platform support of IBR-DTN entails the need for a flexible configuration of the bundle node. IBR-DTN is able to work with or without a persistent file-system, it adopts many networking opportunities using its convergence-layers, and react tolerant to failures. Moreover, the code-architecture allows removal of components which are not supported by a specific platform (e.g. due to a missing port of a library) without any impact to the remaining functionality of the bundle node. Additionally, the library `ibrcommon` has been tested and adopted to work with the `uclibc` library [And12], which is a standard C library for embedded platforms and utilized by OpenWrt.

Beside embedded systems supported by OpenWrt, there are a couple of hardware out there for which it is not feasible to obtain or set-up a dedicated toolchain to build IBR-DTN. In those cases, it is possible to compile statically linked binaries as long as the processor is supported by buildroot and the Linux™ kernel is not too old. These binaries include all necessary dependencies and are able to run on all Linux™-derived systems. This way, we also ported IBR-DTN successfully on devices like the Buffalo Link-Station and the Transcend Wi-Fi SD Card.

4.13.3 Performance

As with all networking stacks, performance matters and the election of the programming language dictates the potential for a good performance. For IBR-DTN, we chose C++ as the language of choice. Beside good modularization of components and data-structures, C++ is close to the hardware, adds only a small footprint, and provides full control over the memory-management. We also made strong use of C++ STL and the `iostream` concept to make the software as flexible as possible for future extensions.

While opening a software for unpredictable extensions, it is important to consider any additional feature as potential impact to the performance. The architecture of IBR-DTN allows a developer to easily extend the software by new features. To avoid performance impacts, each component should process a received event as fast as possible. If the processing may take some time, then the component should

defer the execution using a worker thread. This way, the event-system and other components are not effected by the additional processing time a component may need.

Another performance trap is introduced by the large payload support of the BP. A single bundle can carry up to 2^{64} bytes payload and processing of that data may take some time. For better handling and integration into C++, payload processing is encapsulated into standard input/output stream containers. Those containers implement buffers and provide a blocking interface which restrict the usage to a linear pattern, but also provide a high performance.

The development was accompanied by a continuous integration procedure which performs throughput tests. This way, performance impacts are easily to recognize and since the test are done even after a small set of changes, it is also relative easy to locate the reason for an unveiled performance impact. More details on performance testing are presented in Section 6.3.3.

4.13.4 Fragmentation

As already mentioned in Section 3.7, the ability to split bundles into fragments can lead to unsolicited data duplication. Due to this fact, the set of features regarding fragmentation capabilities are limited or even completely disabled if requested by the corresponding configuration option.

In general, there are two types of fragmentation. Reactive fragmentation is applied if the transmission of a bundle is interrupted. Assuming that the convergence-layer is able to determine how many bytes of the bundle have been successfully delivered in a reliably way, the receiver and the sender can generate two fragments with an adjacent or over-lapping fragments of the payload. In a network with a routing scheme that distributes a lot of bundle copies, this fragmentation approach would potentially generate additional copies on each disruption. In IBR-DTN, only the TCP-CL implements reactive fragmentation. But different to the specification, it only creates a fragment on the receiver side. The fragment at the sender exists virtually, but is never distributed to other nodes. Instead the bundle node will resume the transmission by generating the remaining fragment on demand if there is another contact with the same peer.

Proactive fragmentation splits bundles into fragments before the actual transmission starts. According to the *Bundle Protocol Specification*, a bundle node may generate fragments if a bundle does not fit into a predicted contact time or into the MTU of a convergence-layer. If that approach is applied at each hop, we potentially duplicate the data in the network by distributing complete bundles as well as

fragments. To avoid that, IBR-DTN splits bundles into fragments only at the sender. This way, it is possible to ensure that there is no unnecessary data duplication.

In addition to the splitting policy, we also restrict the reassembly of bundles. The BP permits to merge fragments on the path, but that would lead to redundant data in the network because of the distribution of complete bundles and related fragments at the same time. IBR-DTN only reassembles a bundle if it is mentioned for local delivery.

4.13.5 Security Policy

The *Bundle Security Protocol Specification* provides a set of structures to augment standard bundles with security related data. Additionally, some vague specifications are made to explain how and when security extensions are added. Since the standard permits very complex combinations in all imaginable kinds, a policy is necessary to define the exact behavior of the bundle node. According to the default policy of the *Bundle Security Protocol Specification*, IBR-DTN also specifies its own configurable policy as follows.

- Bundles that are not well-formed do not meet the security policy criteria.
- The mandatory cipher-suites are used to authenticate, sign, and encrypt bundles.
- An outgoing bundle is authenticated using a shared secret if the bundle node has enabled authentication as mandatory.
- A bundle is signed using the private key of the local bundle node if the application requests a bundle signature.
- A bundle is encrypted using the public key of the destination if the application requests encryption and the public key is available in the key-store.
- Optional: All received bundles must have a BAB.
- Optional: All received bundles must have a PIB.
- Optional: All received bundles must have a PCB.
- If a received bundle contains a BAB, it is verified using the default shared key or a key dedicated for that security-source. A bundle with an invalid authentication is discarded and a local warning is printed. BSRs are not generated for this case.

- If a received bundle contains a PIB, it is verified using the public key of the security-source if the key is available in the key-store. A bundle with an invalid signature is discarded and a local warning is printed. BSRs are not generated for this case.
- An encrypted bundle is decrypted on-demand, if an application on the security-destination requests that bundle. A bundle with an invalid encryption is discarded and a local warning is printed. BSRs are not generated for this case.

The policy defines that bundles are only encrypted or signed at the source endpoint. This ensures that each bundle is encrypted and signed only once. That implies, a signature can always be verified before a bundle is decrypted. In general, IBR-DTN is able to verify nested signatures. But since verification is done at reception and decryption right before delivery, the extra validation of nested signatures is omitted. The second benefit of the policy is that fragmentation is always applied afterwards. This allows verification of the payload as a whole and partial verification of fragments is not necessary. However, IBR-DTN also supports verification of PIBs related to fragments.

Since PIBs and PCBs are always applied at the source bundle node, the security-source and the security-destination of bundles generated by IBR-DTN always match the source and destination of the primary bundle block. The ability to route bundles using a different security-destination is not part of this implementation. The forwarding node behaves in those cases as a bundle node without BSP support and always routes bundles to its destination endpoint defined in the primary bundle block.

Still an open question is how the public-keys are distributed. There exist approaches like the identity-based cryptography [BM07], but those have limitations. For example, a revocation of a single key in a domain with identity-based cryptography would imply a revocation of all keys of the network. To gain a functional DTN with an easy key distribution method, we implemented an approach where each bundle node sends its own public key on request to other peers. Since those keys are not authenticated at all, we used a side-channel (e.g. QR codes or hash commitment [BCC88]) to verify the received key.

5 Time-synchronization

In general, »time« is considered as something that can not be mathematically defined. Instead, it is referred to as a physical quantity that can be identified solely by a reproducible procedure. An example would be the interval between a sunrise and the next. The most accurate measurable event takes place in atomic clocks. Here, the timing is derived from the characteristic frequency of radiative transitions of electrons of free atoms. The apparatus, to determine a cycle from a measurable event, is called »clock«.

Both the atomic clock and the alternation of day and night, are used to define time. The problem here is the different accuracy. The deviation of an atomic clock is so small that the uncertainty of time is just one second in 20 million years. The Earth's rotation, which is responsible for the day and night cycles, is much less precise. Depending on the event used for the determination of time, different intervals are the result.

In everyday life, times are used to divide the day, to make arrangements, or just to do something at the same time. Therefore, a common time base for all involved parties is necessary. In computer networks, it can also be advantageous to have a common understanding of time. Moments in time can be assigned to avoid simultaneous speaking, augment data with a validity, or sort messages according to their chronological order. If multiple computers share a common time base, we speak generally of »synchronicity«.

In this chapter, we will investigate the problem of time-synchronization in DTNs. We are going to refine the reasons for the dependency on a global time and identify possible alternatives with their potential limitations. Finally, we will answer the original question of *Joseph Ishac* (Section 2.5.2), when the dependency on an absolute time-stamp is required and when not. Furthermore, mechanisms will be designed to perform a synchronization of clocks within a DTN. In order to get a working approach, the maximum relative deviation must be defined and we need to check whether the designed approach satisfies this requirement.

5.1 Requirements

As a first step, the actual problem, *Synchronicity in Delay- and Disruption-Tolerant Networks*, must be refined. As already mentioned briefly in Section 2.5.2, the BP

depends on global synchronized time-stamps for various mechanisms. In this section, we explain the reasons for that design decision in more detail and clarify their indispensability.

5.1.1 Identification and Referencing

Each generated bundle gets a unique identifier assigned. This consists of the EID of the sender, the creation time-stamp and the creation time-stamp sequence number which must not be assigned twice in the same second. The advantage in using time-stamps is that a bundle node does not need to store already assigned values persistently. This unique identifier is used to detect duplicates and to avoid forwarding loops. Furthermore, fragments are correlated using this identifier and administrative records contain them to reference related bundles.

5.1.2 Validity and Expiry

In a network with dynamic routing, it is important to release capacities, occupied by no longer needed data packets, as early as possible. In traditional networks, data is either passed or dropped immediately. Thus, data gets removed automatically if no route to the destination is known and no loop exists. In the case of a loop, an additional mechanism is needed to resolve this issue. The common variant is to provide the data packet with a maximum number of hops, which is decremented at each transmission. The packet will be void if the hop-limit reaches zero.

In DTNs, the removal of unnecessary data is more complicated. Bundles can linger for a very long time on a node without ever being forwarded because there is no suitable contact. In order to avoid unnecessarily occupied capacities, it must be ensured that bundles are deleted if a delivery is no longer useful. Furthermore, it makes sense to consider the remaining validity of the data for routing decisions. For example, an algorithm may decide that the path to a destination takes too long for a bundle with a short validity. Likewise, a similar decision could cause that certain bundles are discarded to gain more space for others.

Such a decision on the validity of data must be taken autonomously in a DTN, since a further contact with the sender or the receiver, which could make such a decision instead, is not guaranteed. The *Bundle Protocol Specification* defines for this purpose a life-time of each individual bundle. Each bundle carries a creation time-stamp and a maximum life-time in seconds. If the life-time expires, the associated bundle can be deleted. Assuming there exists an accurate global time-synchronization, this method would perfectly fit its task.

For the pretty common case that a time-synchronization is not present, the approach mentioned in Section 2.5.2 was presented to track the aging of bundles with

relative timing. It determines the validity of bundles by local aging and marks the corresponding bundle on each forward. Problems arise with this method, if the clocks of a local system run too slow or too fast and thus a bundle on one host is aging faster or slower than on another one. Until the bundle reaches its destination or finally expires, the errors of the individual systems sum up to a large inaccuracy. Moreover, the approach does not work if the transmission delay between two bundle nodes is unknown or residence time of a bundle on an intermediate system can not be determined. This is the case if a system goes to sleep, is disconnected from the power supply, or is merely a passive transport device such as a USB storage medium.

5.1.3 Routing

The routing in a DTN is a very challenging task. Due to the intermittent and usually high-dynamic topology, known reactive methods that perform a search for directions on demand are not suited to make decisions about the next-hop a bundle should pass. To solve the problem, usually predicting [LDS03] or generally applicable [VBoo] methods are proposed in the literature.

A simple approach is the static specification of scheduled contacts, which is easy to perform in the interplanetary communication. By calculating the trajectories of the planet's rotation, each node knows exactly when each node has contact with another node, and how long this contact lasts. This information can be predicted for many years in advance and stored on the node. A different approach pursues the RUTS routing algorithm [DPW10], which is specialized in using predefined routes of public transport and the corresponding time-tables to find multiple paths to a destination. To compensate potential interferences (e.g. schedule deviations), several alternative paths are calculated. A mobility model especially developed for the research of DTN routing algorithms is also based on the recurring meetings typically for DTNs [EKKO08]. That approach simulates the habits of people, which organizes their activities, and therefore also their personal mobility, in daily and weekly rhythms.

The processes mentioned above have one thing in common, they use recurring patterns which are defined by a specific timing. In order to use these patterns, a DTN node needs to make decisions based on future encounters. If the clocks of two nodes are too far apart, a routing with the procedures referred to in this section may not be carried out.

5.1.4 Energy-saving

As mentioned in the previous section, the encounters in DTNs can usually be predicted. Therefore, a node, which has temporarily no neighbors, can save a large portion of its energy by going into a sleep mode until the next encounter is expected. In particular, the wireless communication device offers a high potential for savings when switched off. The prerequisite for energy-saving using this approach is also a precise knowledge of the time that can be slept.

If the node has no further duties beside to store and forward messages, then it can be put into sleep mode as soon as there are no neighbors available. However, the node should be awakened as soon as a neighbor appears and is ready to communicate. With a sufficiently good time-synchronization and scheduled contacts, as present in the interplanetary communication scenario, nodes can go to sleep with the help of an external timer for a long time and thereby save energy. However, if it is not known when the next contact appears, a mechanism is needed which wakes the node in case of an opportunistic contact [DRW11].

5.1.5 Security

Another more general dependency on synchronized clocks is motivated by security-related mechanism to secure the communication between peers in an untrusted environment. The *Bundle Security Protocol* contains approaches to prevent man-in-the-middle attacks, replay attacks, data injection as well as manipulation using symmetric and asymmetric cryptography. For the latter technique each node must own a public/private key-pair which is used to sign outgoing bundles. To encrypt a bundle for another peer, the public key of the destination must be known in advance and, more important, evaluated as trustworthy by the sender. This requirement is one of the most challenging tasks of the open key-distribution issue mentioned in Section 4.13.5.

In traditional networking scenarios a Public-Key-Infrastructure (PKI) is often considered to realize a trust chain. In order to evaluate received public keys of foreign nodes, each node owns a pre-deployed set of Certificate Authority (CA) certificates. Instead of distributing node-specific public keys directly, a node must offer a certificate attached to its public key which must be signed with a certificate of a trusted CA. The complete approach is based on the assumption that private keys are always kept secret. Thus, a lot of effort is spent to guarantee confidentiality for CA keys. Nodes in a DTN are vulnerable due to their exposed position and it may happen that a node get compromised in some way. For those cases, a CA publishes a revocation list which contains invalidated keys which are prospectively excluded from the network.

Publishing a revocation list in a DTN is not as easy as in continuously connected environments. Moreover, compromised keys are seldom recognized immediately. With enough effort and time it is possible to compute the private key from a low-bit public key. Even if the corresponding node already uses a new and larger key, the old one would be still valid as long it is not revoked. To limit the available time to compute the private key and avoid a constantly growing revocation list, certificates are limited to an epoch and contain an issuer as well as an expiration date. The revocation list only have to contain certificates valid in the current epoch and if an attacker can not compute the private key within the epoch the result would be useless. The limitation to an epoch introduces the dependency on synchronized clocks. In order to evaluate a certificate, a node needs a clock to check if the epoch of the certificate is valid.

Certificates are not the only security-related approach which introduces the dependency on synchronized clocks. For example, the *Kerberos Network Authentication Service* [KN93] is a protocol to authenticate users in a network. It uses time-stamps to detect and prevent reply attacks. For this purpose, each request contains a signed time-stamp and the respective receiver determines the age of a request using the local time. If the request is too old, it will be discarded.

5.1.6 Limitations

The preceding sections show that many mechanisms of a DTN are already dependent on a common sense of time. Many of the duties could be solved with local time tracking, but local clocks are notoriously inaccurate and will never match any clock on another system. Thus, it is impossible to determine the exact life-time of a bundle with such an alternative.

In practice, the dependence on global time has been recognized as problematic. Clocks in computers are usually very inaccurate and run at different speeds. Which in turn leads to issues, even if the clocks were perfectly adjusted on initialization. Apart from inaccuracies, wrong configured time-zones as well as the initial clock state of a system are sources of issues. In the latter case, embedded systems often start with a predefined time (e.g. 01/01/1970), because a clock with an attached battery is not available.

At this point, we want to soften the requirements of synchronicity further. It can be assumed that predictions about upcoming encounters are rarely more accurate than a second and thus variations can be tolerated within a precision of one second. The potential of energy-savings for wireless devices is at best if the phase while the device is active without any ongoing transmission can be completely eliminated. Assuming that the prediction for encounters does not provide a resolution lower

than one second, mechanisms for energy-saving must work with the same resolution. As defined by the *Bundle Protocol Specification*, the life-time of each bundle is also restricted to a resolution of one second. A time-synchronization in a DTN would therefore be acceptable if the error is less than a single time-unit, hence one second.

Furthermore, in a system which is used to work with predictions based on global time-schedules, at least one node needs to be defined as reference. If there are multiple nodes that have precisely synchronized clocks, it should be possible to utilize all of them to support the whole system. In addition, it is assumed that each node has a local clock with an unspecified but approximately constant inaccuracy.

5.1.7 Alternatives

The BP has a time-stamp in each bundle indicating when a bundle was created. Along with the life-time of the bundle, a bundle node can determine when a certain bundle is obsolete and may be discarded. Such a mechanism is necessary since otherwise bundles could linger forever in the storage or repeatedly traverse the same nodes in loop. In addition, the sender can use the life-time to determine whether it is necessary to re-send a message.

However, an absolute and globally valid time is not always available or can not be achieved with reasonable effort. Thus, there exist considerations to replace the absolute time by a relative time measurement or other mechanisms to resolve the aforementioned problems. So the question is: *How can the dependence on a global time in DTNs get minimized?*

As already discussed in Section 2.5.2, the problem of the loops could be solved analogous to protocols such as IP. They define a counter (hop-count) which is reduced upon the arrival at a station. If it reaches zero, the bundle must be discarded. This effectively prevents an infinite dissemination of bundles, but not the possibility that a bundle lingers forever in the storage of a node and never gets delivered. The BP can be extended for that purpose with an extension block or by changing the semantics of the creation time-stamp and life-time values.

In mixed environments, where nodes wake-up without being synchronized but others with synchronized clock exist, the *Deferred Window Scheme* can be used to delegate the assignment of a creation time-stamp to another node. In order to achieve that, a bundle gets a special marker to indicate that the time-stamp is invalid. As soon as the bundle crosses a node with reliable time information, this node can assign a valid creation time-stamp. Thus, the unique identifier of the bundle gets changed and the bundle will not age until the creation time-stamp is assigned.

In order to limit the life-time and therefore the maximum residence time in a node without the assignment of a valid creation time-stamp, a time measurement needs to be performed. A suggestion is to measure the age of a bundle relatively as done by the AEB extensions presented in Section 4.12.1. For this purpose, the creation time-stamp of a bundle is ignored and its age is incremented every time the bundle leaves a node. Is the original life-time known, the following equation can be used to determine the creation time without having any precise knowledge of a global time.

$$\text{Creation time} = \text{Now} - \text{Age of the bundle}$$

To measure the relative age, it is necessary that each node is aware of how much time a bundle spent in the storage. This is especially challenging if a node is turned off to conserve energy, the transmission delay between two nodes is unknown, or a bundle is being transported on a USB drive. Furthermore, irregularly running clocks lead to an inaccurate determination of the relative age.

5.2 Time-reference Distribution

All alternatives mentioned before mitigate the dependency on synchronized clocks. However, they suffer from inaccuracies and cases in which bundles may linger forever on a node. Thus, the only solution to cover all cases is to apply a clock synchronization approach to the whole network.

Generally, there exist three categories of clock synchronization approaches. The first category includes all algorithms which harmonize the clocks in regard to offset and skew, so that all nodes agree to a common time-stamp and pace after some time. These algorithms pass on external references and have to form a consensus in a distributed way. An example for such an algorithm is the *DCS algorithm* [CS10] previously referenced in Section 2.5.2. The second category of algorithms do not synchronize clocks. Instead, the time-data of messages is transformed every time it enters or leaves the scope of the individual node. These approaches need to determine an offset between every pair of nodes which is done locally by the pairs. The *Römer's synchronization protocol* [Ro1] presented in Section 2.5.3 uses such a principle. The last category of algorithms presented here adjust clocks within the network to a given reference. These algorithms always declare at least one reference which produces some sort of time-data and distribute it to other nodes. The number of references depends on the capability of the algorithm and the network topology.

The approach proposed in this work, which was previously presented in [MW12], fits into the last category and synchronizes clocks in a network where nodes have an unpredictable encounter pattern. A portion of the nodes (at least one of them)

have a high-precision clock and are considered as reference. Nodes synchronized by a GPS signal, using NTP over an Internet connection, or by a DCF77 receiver would perfectly fit this requirement.

5.2.1 Algorithm

As previously explained in Section 5.1 many applications and protocol mechanisms need to synchronize to an external reference or even to the UTC. Although broadcasting time-stamps for more or less adequate synchronization is quite simple in a traditional network, it would not work in DTNs due to the potentially high message delay. One of the main issues here is the tracking and correction of the time-data along the path. We need to know the relative age of the received reference time-stamp to adjust the local clock the right way. Since such a tracking of the age is done using the clocks we want to adjust, we need a way to assess the received data. Therefore, the main issue of distributing time-references is the decision which clock is more accurate than the other and which provides the better time-stamp.

The idea behind our algorithm is a rating of any clock value at any time. To achieve this, we define a rating value ξ first.

$$\{\xi \in \mathbb{R} \mid 0 \leq \xi \leq 1\} \quad (5.1)$$

With this value we can compare two clocks directly. $\xi = 1$ indicates a perfect clock with no error at all and nodes with a rating of $\xi = 0$ have no clue about the current time.

Reference clocks are synchronized in a classical fashion, e.g. with the NTP, GPS or DCF77 radio signal. We trust on them as they provide the most accurate time in the whole network. Thus, reference clocks distribute their time-stamp with a rating of $\xi = 1$. Nodes synchronize their clock to other clocks if the foreign rating is higher than their own. Between each synchronization the cumulative clock error of each node raises and thus the local rating should decrease over time depending on the own estimated clock error.

The determination of the local clock error is done on each synchronization. If a node decides that the clock of another node is better than its own, it derives the clock offset which has arisen since the last synchronization. Together with the preceding time period since the last synchronization, this offset leads to an estimation about how good or bad the local clock behaves.

In detail, we declare: *A* synchronizes to the clock of *B*:

$$A \rightarrow B \quad (5.2)$$

This only happens if the rating of B (ξ_B) is larger than the rating of A (ξ_A):

$$\xi_A < \xi_B \quad (5.3)$$

If this constraint is met, the node A assumes that the clock of B is better than its own and synchronizes itself to the clock of B using the current time-stamp of B (t_B):

$$t_{new} = t_B \quad (5.4)$$

Instead of setting t_A (the time-stamp of A 's clock) directly, it is recommended to adjust it smoothly to the new value. This can be done by speed-up or slow-down the clock until the new value t_{new} is reached. How this is done in detail is an implementation matter and depends on the specific target platform. Section 5.2.4 proposes some approaches to achieve a smooth adjustment.

Further, the node has to set a new rating $\xi(t_0)$ for its own clock. Since we assume that the clock of node B is better than the clock of A , the difference between t_A and t_B indicates how good or bad the clock of A is in relation to B 's clock. Thus, we use the offset to lower the rating ξ_B and use it as new initial rating $\xi(t_0)$. t_0 is defined as the time-stamp of the last synchronization.

$$\xi(t_0) = \xi_B \cdot \frac{\min(t_A, t_B)}{\max(t_A, t_B)} \quad (5.5)$$

Each time the rating is needed, the node can calculate the current rating of its own clock. The rating of the last synchronization is decremented exponentially using the past time interval and a factor $\{\sigma \in \mathbb{R} \mid 1 < \sigma\}$, which specifies how fast the rating decreases.

$$\xi(t) = \xi(t_0) \cdot \frac{1}{\sigma^{(t-t_0)}} \quad (5.6)$$

Figure 5.1 shows the plot of $\xi(t)$ dependent on different values for σ . Dropping values represent an uncertainty about the local time information.

To choose the right value for σ , a self-evaluation of the local clock is needed. On each synchronization σ is re-defined according to the arisen offset between the local and the foreign clock since the last synchronization. Additionally, we use the rating of the foreign clock (ξ_B) to mitigate the effect of this adjustment on σ .

$$\sigma_{new} = 1 + \left| \frac{t_B - t_A}{t_A - t_0} \right| \cdot \xi_B \quad (5.7)$$

Since this may result in $\sigma = 1$ and could avoid reduction of the clock rating, we force the reduction by a given amount ψ during the next period derived from the

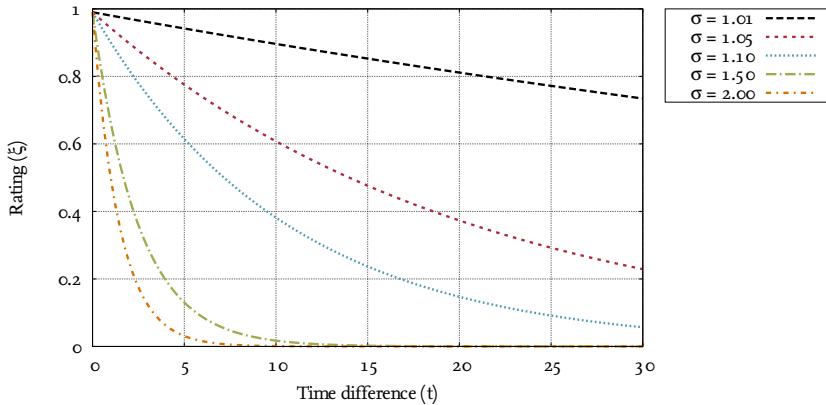


Figure 5.1: Plot of $\xi(t)$ dependent on different values for σ

time since the last synchronization. To do this, we just invert the equation (5.6) and add the result to the previous equation.

$$\sigma_{new} = \psi^{-\frac{1}{t_A - t_0}} + \left| \frac{t_B - t_A}{t_A - t_0} \right| \cdot \xi_B \quad (5.8)$$

With equation (5.8) we gain an automatic adaptation according to the adjustment frequency and the local clock accuracy. The only parameter to set is ψ , which specifies the minimum reduction between each synchronization. In our experiments, we identified $\psi = 0.9$ as a good choice.

5.2.2 Delay-tolerant Network Time Protocol

In this section, we specify a protocol to synchronize clocks in a DTN based on the previously presented approach. In addition to the ability to evaluate the clocks of bundle nodes, an approach is required to determine the offset between two clocks. Since these can not be compared simultaneously, there will be always a delay between reading the first and the second clock. The protocol presented here uses a request-response principle to determine the offset and the delay between the two comparison steps in order to perform the clock synchronization. Since it is based on the BP, it uses bundles to exchange the necessary data between the bundle nodes. This way, the approach does not need to utilize a side-channel and can be performed with any convergence-layer.

To perform a synchronization to the clock of another bundle node, the local time-stamp is placed together with the corresponding clock rating in a bundle and sent to a remote bundle node. This attaches its own time-stamp and clock rating to the bundle and send it back. Finally, the local clock is adjusted using the algorithm of Section 5.2.1 and the offset between the time-stamps in the received bundle.

To determine an accurate offset between the time-stamps, it is not sufficient to simply compare them. There is an undetermined delay between the measurement of the first and the second time-stamp. In order to achieve a synchronization with a high accuracy, that delay must be determined and considered during the calculation of the offset. The delay consists of two parts. The time a bundle is stored on the nodes (*resident time*) and the time it takes to transfer the bundle to another bundle node using a convergence-layer (*transmission delay*). By measuring the RTT between request and response, we get the cumulative time of the way there and back. A naïve approach would be to divide the RTT by two to gain the necessary offset, but since we can not assume that the residential time is symmetric this would lead to an inaccurate result. To determine the transmission delay, we need to measure the resident time separately. Luckily, the AEB presented in section 4.12.1 already supports that as byproduct.

In order to measure the resident time, an AEB is attached to the request bundle. As long as the responding application just reflects the received bundle together with the attached AEB, it will record the resident time of the way there and back. To separate measurements of the way there from the way back, the responding application places a second AEB in front of the existing AEB to measure the resident time of the way back. The transmission delay is excluded in this measurement. Figure 5.2 depicts the different measured time intervals and the message format used to collect necessary data to synchronize the clock of bundle node A to the clock of bundle node B.

1. The request bundle is just created and contains an AEB with the relative age set to zero, the current time-stamp of A's clock and the corresponding clock rating. As next, the request bundle is placed in the storage and the routing components are triggered to route that bundle to its destination.
2. The request bundle leaves the bundle node through a convergence-layer. At this stage, the relative age of the bundle has been increased by the resident time (a). In this scenario the transmission time (b) is undetermined. Thus, the request bundle arrives unmodified at the convergence-layer of the bundle node B, which performs several processing steps and finally delivers the request to the application.

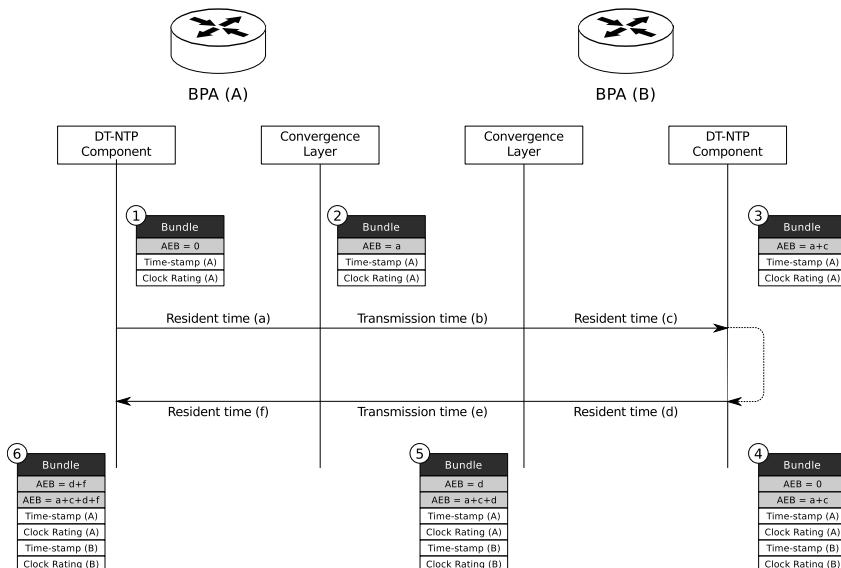


Figure 5.2: Determining the time difference between two nodes

3. The relative age has been increased by the resident time (c) and is processed by the application responsible for the synchronization process.
4. The application attaches the current local time-stamp and the corresponding clock rating at the end of the payload. Additionally, a second AEB is placed in front of the previous one which is now representing the relative age of the response. Then, the response bundle is placed in the storage and the routing components are triggered to route that bundle to its destination.
5. The response bundle leaves the bundle node through a convergence-layer. At this stage, the relative age of the bundle as well as the value of the previous AEB has been increased by the resident time (d). Again, the transmission time (e) is undetermined and the response bundle arrives unmodified at the convergence-layer of the bundle node A, which performs several processing steps and finally delivers the response to the application.
6. The relative age as well as the value of the previous AEB has been increased by the resident time (f) and is processed by the application responsible for the synchronization process.

In order to compare the time-stamps in the response bundle correctly, the residence time and the transmission delay must be determined. First, we need to determine the RTT of request-response procedure using the time-stamp of the response (A) and the current time-stamp (A').

$$\text{RTT} = \text{Time-stamp}(A)' - \text{Time-stamp}(A)$$

Assuming that the transmission delay is the same for both directions, we can estimate the transmission delay (T_p) by deducting the value of the reflected AEB, which contains the sum of the resident time a, c, d , and f , from the previously determined RTT.

$$T_p = \frac{\text{RTT} - \text{AEB}(a + c + d + f)}{2}$$

Finally, we can adjust the received time-stamp using T_p and the relative age of the response to determine the actual offset between the local and the remote clock.

$$\text{Time-stamp}(B)' = \text{AEB}(d + f) + T_p + \text{Time-stamp}(B)$$

By augmenting bundles of the time-synchronization protocol with AEBs, it is possible to determine the transmission delay as well as the resident time. This is important to compare measured time-stamps correctly. Additionally, this extension serves another important purpose. Usually it is almost impossible to exchange bundles between nodes with drifting clocks. Due to the life-time limitation, short lived bundles are rejected or dropped at the receiver if clocks are too far apart. However, this issue can be bypassed by using AEBs as explained in Section 4.13.1. A bundle node which receives such a bundle would consider the relative age for expiration instead of the time-stamp in the primary bundle block. This way, it is possible to synchronize bundle nodes using bundles even if their clocks are drifted too far apart.

5.2.3 Security

To apply this approach to an untrusted environment, mechanisms are needed to prevent attackers from injecting wrong time-references into the network. Using the TLS extension mentioned in Section 3.9.5 or the authentication feature of the *Bundle Security Protocol*, it is possible to close a DTN and exclude untrusted nodes from injecting any data into the network. A less restrictive solution would be to send and accept only signed bundles during the synchronization process. But this would imply to trust all nodes with a valid certificate to behave correctly, because certificates only proof the identity of a single node and not the correctness of the time-data it carries.

The solution would be to sign each reference time-stamp with a trusted asymmetric key. On each synchronization, the node with the better clock rating signs its own time-data, consisting of the time-stamp, the clock rating, and the signed time-data of the last synchronization. The resulting chain of signed time-data entries is verifiable on each hop. Thus, it is impossible to distribute lower time-stamps as those distributed by the trusted reference nodes, even if the attacker owns a valid certificate. In the worst case, an attacker can speed-up or slow-down clocks of other nodes. In regard to the requirements, mentioned at the beginning of this chapter, such attacks would not even break the security-related validity of certificates. Since it is impossible to set the time to an earlier time-stamp, deprecated certificates stay invalid. However, misbehavior-detection systems as presented in [ZDG⁺14] can be used to detect malicious nodes and stop them from injecting manipulated time-data.

As for other cryptography-based approaches, the key distribution needs to be clarified to implement the proposed mechanisms. Furthermore, the reference nodes need special certificates to proof their right to act as reference for the network.

5.2.4 Local Clock Adjustment

During a synchronization between two nodes, the clock may not be adjusted by simply setting the current time-stamp. Such a procedure might have undesirable consequences. For example, a backward adjusted clock may lead to a twice assigned bundle identifier and thus to false-positives during duplicate checking. Instead of setting the clock backwards, the time must be slowed down to get the system timestamp close to the aspired one.

The Linux™ operating system provides a command for this purpose which instructs the kernel to adjust itself according to given parameters. The command `adjtime` respectively `adjtimex` indicates the difference to the aspired time-stamp and instruct the kernel to slow-down or accelerate the system clock to compensate the difference. This method is also used in software for time-synchronization over the Internet using the NTP. Alternatively, the program `chrony` [Cur14] can monitor the local clock. It allows you to manually set a time-stamp and then adjusts the local clock smoothly to it.

5.2.5 Scope and Network-wide Time

In order to synchronize local time information with another node, a bundle node can adjust its system clock according to the synchronization result or just remember an offset to realize an internal view to a logical clock. The latter approach allows

a bundle node to use synchronized time information for all the DTN-related functionality mentioned in section 5.1 without modifying the underlying system. This is especially useful if necessary rights to do so are not granted.

By limiting the synchronization process to the DTN related software, we can avoid side-effects. Additionally, jumping forward in time or delaying the clock for a while has less negative impacts on other system components. But a single danger with such an approach still exists. If there are additional clock-adjusting processes on the system, the estimated offset of an already synchronized bundle node would be invalidated by external clock adjustments, while the bundle node assumes that its time-stamp is correct. To avoid that issue, the bundle node should calculate its local time based on the monotonic time information of the underlying system.

5.2.6 Implementation

To integrate the approach for time-synchronization into the architecture and implementation of IBR-DTN, we added an embedded application named **DTNTPWorker** as already explained in Section 4.11.1. It listens to incoming bundles addressed to the endpoint suffix »dtntp« and sends out synchronization requests if a peer with a higher clock rating has been discovered.

To signal time-synchronization capabilities and the local rating to adjacent peers, the application implements the `DiscoveryBeaconHandler` interface and registers itself at the `DiscoveryAgent` to get asked every time a new discovery beacon is created. Within the callback-method `onUpdateBeacon` the application adds a new service entry with the tag 'dtntp' to each beacon. This entry contains the version, the clock rating, and the current time-stamp. If a peer receives such a service entry, it will be stored in Node entities in the neighbor list available through the `ConnectionManager`.

The **DTNTPWorker** application iterates every second through all neighbors and evaluates the clock ratings of all peers. If there are nodes with a better rating, the best one is selected and a request, with a life-time of 60 s and a high priority, is generated. According to the protocol defined in Section 5.2.2, the requests includes an AEB, the current time-stamp, and the local clock rating. Since the evaluation of peers is performed every second, an internal blacklist disables further attempts to the same peer as long as the last request is valid. To avoid imprecisions by intermediate hops, a SCHL extension block (Section 4.12.3) limits the path of the bundle to a single hop.

As soon as the synchronization response arrives at the application, the offset between the local and the remote clock is calculated. The result and the new local clock rating is assigned to the internal abstraction in the `Clock` component which

may set the clock of the system. Additionally, the offset and the new clock rating is announced using a `TimeAdjustmentEvent` to other components. Every further second, the local clock rating is aged and reassigned to the `Clock` component.

In addition to the mechanism for time-synchronization, the `DTNTPWorker` application also controls the clock rating in case of an obviously wrong clock, e.g. the current time-stamp is before 01/01/2000. In such a case, the application sets the local clock rating automatically to zero and enforces this way an AEB in each newly created bundle.

6 Evaluation

In this chapter, we are going to evaluate the concepts presented in previous chapters by testing the implementation of IBR-DTN. The first section contains a summary about attempts to port the software to other platforms. Each attempt reports the necessary steps and the peculiarities each platform brings with. In Section 6.2, we set-up a measurement testbed for performance testing and compare the result of IBR-DTN with other implementations. During the tests, we measure the basic performance of the API and the throughput in an encounter scenario where one node pushes bundles as fast as possible to another node. Moreover, we consider options to tweak the performance and evaluate their impact. The concept of Continuous Integration is presented in Section 6.3. It explains the steps we have done to improve the overall software quality of IBR-DTN and how multiple package repositories for different platforms are maintained. In Section 6.4, we evaluate the Time-synchronization algorithm presented in Chapter 5. For that purpose, we define multiple DTN scenarios and implemented these in the ONE simulator and the HYDRA emulator environment. The result of both approaches are finally compared and used to proof that the implementation within IBR-DTN behaves similar to the simulated approach.

6.1 Portability

As of the start of the development, the major target was the OpenWrt distribution which is based on the uclibc library. Since, it is hard to develop and build software directly on this platform, development and testing were performed on Standard Linux™ distributions based on Debian. Thus, it is expected that the software works best on these systems. In this section, we will evaluate the portability by discussing the issues discovered while porting the software to several platforms.

6.1.1 OpenWrt

OpenWrt [Ope14] is a Linux™ distribution for embedded devices. Using the corresponding toolchain it is possible to build firmware images for various embedded devices. Most of them were originally distributed as router devices. The system

provides package management which allows a user to customize the installation using a variety of packages.

The distribution is completely based on Linux™ and associated tools. The main difference to Desktop and Server distributions as Debian and Ubuntu™ is the C library on which all packages are based. On OpenWrt the *μclibc* [And12] is used instead of the more common GNU C Library *glibc*. *μclibc* is a C library optimized for embedded systems and has a small memory-footprint.

Although the *μclibc* includes an implementation of the Native POSIX Thread Library (NPTL), the distribution disabled that feature in favor of an approach using processes and shared memory a.k.a. *linuxthreads*. Different to NPTL, this approach is even supported by older kernel versions including 2.4 or 2.2. Due to closed driver binaries, those kernels are still relevant to support older hardware. As of revision *attitude adjustment* of OpenWrt, NPTL are enabled by default, but the development of IBR-DTN has started long before the release of OpenWrt with NPTL support. Thus, we designed the software to be compatible with support for *linuxthreads*.

In fact, there is no need to include platform-specific code to make software work on systems using *glibc* as well as on those based on *μclibc*. But developers must be aware of existing inconsistencies in handling signals and threads. We observed that signals caused by system procedures within a specific thread might get directed to all threads or simply the wrong one. Instead of investigating the exact behavior under certain conditions, we simply decided to avoid procedures which invoke signals.

An inevitable signal is a SIGPIPE in case of a *write()* call on a broken file descriptor. Such a signal is even caused if a socket is closed remotely. Without catching this signal the program is usually terminated. Since we are not able to catch that signal consistently in the thread which caused the signal, we need to block it and handle error cases differently. The code snippet in Listing 6.1 is executed at the top of the *main()* routine to mask out the SIGPIPE signal.

Listing 6.1: Code snippet to block SIGPIPE signal

```

1 sigset_t blockset;
2 sigemptyset(&blockset);
3 sigaddset(&blockset, SIGPIPE);
4 ::sigprocmask(SIG_BLOCK, &blockset, NULL);

```

Another inconsistency between *μclibc* and *glibc* was the usage of cancellation of POSIX threads. The method *pthread_cancel()* unreliable interrupts threads and leads to unexpected behavior especially if locking procedures are present. Thus, we chose to realize threading with the requirement, that each thread must be

interruptible. For that reason, each class implementing `ibrcommon::Thread` must provide a `__cancellation()` method which is called if the thread should terminate its processing. As a result, all other blocking methods within the software, including those in blocking queues and sockets, must be designed as interruptible too.

6.1.2 OS/X®

The OS/X® operating system developed by Apple and distributed along with their computers is a POSIX-compliant BSD-derivate. Although it is recommended to write software for this platform in Objective-C, software written in C++ compiles and runs pretty fine as long as it depends only on methods defined by the POSIX standards.

An elegant way to port applications written for the Linux™ environment to OS-/X® is provided by the MacPorts project [The14]. This is an open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading open-source software on the OS/X® operation system. Once the ports package manager is installed on a system, a large catalog of open-source software is offered to the user. If a package is selected for installation, the tool downloads the sources of the package, builds them, and installs the binaries. For each package exists a recipe which contains references to dependencies and all instruction to build the sources. Dependencies are automatically solved by the package manager.

Although the package manager of MacPorts handles the installation process very nicely and all required libraries for IBR-DTN are present in the package catalog, there are still adaptations required to get the software working on OS/X®. The scaling feature of IBR-DTN queries the number of processors to spawn as many threads as reasonable to utilize the available capacities efficiently. On Linux™ this is done using the `sysconf()` call. OS/X® needs a special code block for the same procedure based on the `sysctl()` method. Since OS/X® supports POSIX threads, there are no special treatments necessary for them except for the location of the `semaphore.h` header. This is differently placed into the `sys/` folder. A configuration script checks at the beginning of the build process where exactly the header is located. The method `basename()` to gain the file-name from a file-path has a different signature. While the Linux™ variant accepts a constant string as input, the OS/X® variant does not. This prevents the corresponding method in the `ibrcommon::File` class from being used as `const`. The work-around is to copy the constant input string in order to remove the `const` property. Further, the socket options `IP_MULTICAST_LOOP` and `IP_MULTICAST_TTL` are not available. These are used to disable loop-back of multicast datagrams and to increase their TTL value.

Since, these options just optimizes the networking routines and does not provide a critical functionality, they are just left out.

In order to react to interface changes, IBR-DTN gets notified via the netlink library about added or removed addresses of bound interfaces. Since this library is specific to Linux™ and not available on other platforms, an alternative is necessary to support OS/X®. If the netlink library is not available during the build, the PosixLinkManager monitors all bound interfaces by simply iterating over all addresses and compares the set with the previous state. This approach is not as fast as getting notified by the kernel about every change, but it is a reliable alternative.

OS/X® does not include the POSIX method `clock_gettime()` to get a monotonic clock value. This is required for time measurements and several other time related mechanisms. The clock type `REALTIME_CLOCK`, offered by the system clock service on OS/X®, provides time values since the system last boot which is similar to a monotonic clock. The `MonotonicClock` class of the `ibrcommon` library returns a monotonic time based on the platform-specific implementation.

A significant difference to Linux™ was found within the `::select()` routine if that is called with a time-out. By specifying a time-out value it is possible to return from a blocking call even if there was no activity on any of the observed file-descriptors. According the POSIX standard, the value of the time-out variable is undefined as soon as the call returns. In contrast, the value is well-defined on Linux™ and is decremented by the time the call has been blocked. For example, if `select()` is called with a time-out value of 4 s and returns due to activity on a file-descriptor after 1 s, the time-out value has been decremented to 3 s. This behavior allows a developer to call `select()` in a loop while it is guaranteed that the method returns after the originally defined time-out, even if the call has returned in-between.

To recreate the behavior of the Linux™ variant on other platforms, we implemented a replacement for the `select` method. Listing 6.2 shows the implementation of the `__compat_select()` method which has the same signature as the POSIX variant. If that method is called without a time-out (line 4), it just redirects the call. If there is a time-out given, the method needs to track the time the `select()` call has spent in blocking state. For that approach an instance of the `ibrcommon::TimeMeasurement` class is used (line 9) which is started (line 13) before the actual call of `select()` and stopped (line 24) afterwards. The remaining code between lines 26 and 41 decrements the original time-out value by the measured time.

All necessary changes are made within the `ibrcommon` library. Other packages was not modified in order to get ported to OS/X®. Adaptations to the threading

Listing 6.2: Emulate the Linux™ behavior of the select() method

```
1 int __compat_select(int nfds, fd_set *readfds, fd_set *writefds,
2                     fd_set *exceptfds, struct timeval *timeout)
3 {
4     if (timeout == NULL)
5     {
6         return ::select(nfds, readfds, writefds, exceptfds, NULL);
7     }
8
9     TimeMeasurement tm;
10    struct timeval to_copy;
11    ::memcpy(&to_copy, timeout, sizeof to_copy);
12
13    tm.start();
14    int ret = ::select(nfds, readfds, writefds, exceptfds, &to_copy);
15
16    // on timeout set the timeout value to zero
17    if (ret == 0)
18    {
19        timeout->tv_sec = 0;
20        timeout->tv_usec = 0;
21    }
22    else
23    {
24        tm.stop();
25
26        struct timespec time_spend;
27        tm.getTime(time_spend);
28
29        timeout->tv_sec -= time_spend.tv_sec;
30        timeout->tv_usec -= time_spend.tv_nsec / 1000;
31        if (timeout->tv_usec < 0) {
32            --timeout->tv_sec;
33            timeout->tv_usec += 1000000L;
34        }
35
36        // adjust timeout value if that falls below zero
37        if (timeout->tv_sec < 0)
38        {
39            timeout->tv_sec = 0;
40            timeout->tv_usec = 0;
41        }
42    }
43    return ret;
44 }
```

procedures of signal handling were not necessary. But we did not verify if those would be necessary if the measures to support OpenWrt were not already applied.

6.1.3 Android™

The Java-dominated Android™-platform is the major platform for mobile devices such as phones or tablets. Recently, Android™ has been settled even to wearables such as glasses and watches. A port of IBR-DTN to mobile devices already present in everybody's pocket will enable DTN-related research in environments where encounters between devices already exist without the need to artificially equip people with experimental hardware.

Since Android™ is just a Linux™ system with a different C library, the port of the software is similar as for OpenWrt. A simple approach is to run a static binary directly on a device. Those binaries can be build using the buildroot project [Kor14]. While this is sufficient for a first proof-of-concept, the set-up is not particularly user-friendly nor redistributable. Applications on Android™ should provide a GUI to start and stop the service. Moreover, third-party applications need access to an API in order to send and receive bundles. Connecting to local sockets as necessary on other platforms is not the preferred way on Android™.

As a step towards a mature port, we investigated the possibilities for C++ code to run within a native Android™ application. Using the Native Development Kit (NDK) provided by Google it is possible to compile C++ code which can be loaded and called from Java applications via the Java Native Interface (JNI). Since those interfaces can get quite complex, we utilized the swig [Davi14] project to generate the required interfaces. To avoid a lot of maintenance for Android™, we also utilized the androgenizer tool to parse Makefiles and generate descriptions for the compilation process required by the NDK. All those measures allows us to build IBR-DTN and embed the functionalities into a native Android™ application. By porting the netlink library and openssl, it was even possible to run the full-featured bundle node on unmodified Android™ devices. The only feature we decided to drop was the support for a SQLite-based storage, because of the poor performance and the additional dependency.

Beside the basic port, we added Android™ specific optimizations and features to the package. First the logging was replaced by calls to the native logging framework of Android™. Instead of instantiating the socket-based API, a native API layer translate calls from the Java world into C++ and back. The API is accessible to third-party applications using a library which communicates with the service process of IBR-DTN. Finally, a smart discover policy to reduce the energy-consumption and a

module to support Wi-Fi Direct for ad-hoc connections between devices has been realized. Figure 6.1 shows screenshots of the Preferences and the Logging activities.

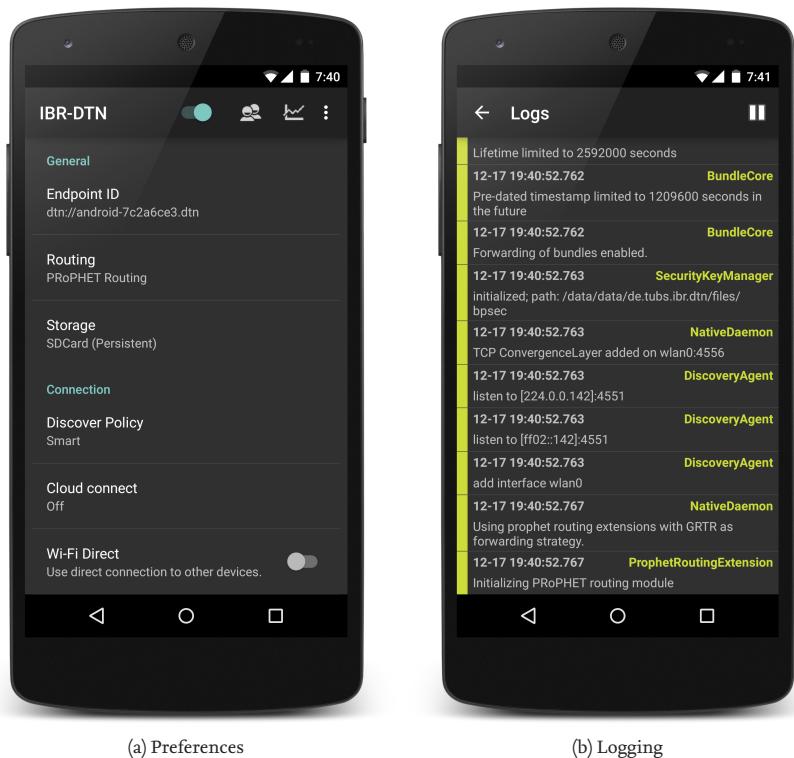


Figure 6.1: Activities of the Android™ port

The IBR-DTN for Android™ is available for free in Google’s Play Store and the sources to build the package are also part of the public repository. Since the service-application alone is almost useless for end-users, we added three applications to the portfolio. *Whisper* is a chat application which shows presence of users and allows them to send text messages, *Talkie* can send and receive voice message, and *ShareBox* can transfer files between devices. These applications are open-source as well and can be considered as reference for application development. Using IBR-DTN for Android™ and the available API library, third-party developers are

now capable in writing DTN-aware applications and can benefit from the capabilities a DTN brings with.

6.1.4 Microsoft® Windows®

By supporting Microsoft® Windows®, we round up the support for the three major operating systems on the market. In contrast to the previously presented platforms, Windows® does not originate from UNIX and offers a complete different set of system calls. Luckily, there exist projects like MinGW [Pet12] which makes the cross-platform development a lot easier. This minimalist development environment allows a Linux™ developer to stick with the well-known build tools, but develop native Windows® applications without additional dependencies to external libraries. MinGW does not change the fact, that the Windows® API is not POSIX compliant and there is a need to adapt procedures to make them Windows® compatible.

As on OS/X®, the netlink library is not available on Windows®. But the alternative, using POSIX calls, is also not supported. Thus, we need to implement a dedicated LinkManager which is capable in listing addresses bound to a given interfaces. Similar to the POSIX variant, this implementation monitors all bound interfaces by simply iterating over all addresses and compares the gathered set with the previous state to finally inform listeners about changes. Beside that, Windows® also requires a special procedure to look-up the number of available processors, identifiers of threads are handled a different way, and the access to a monotonic clock is a bit more complicated than it is on POSIX compliant systems. Moreover, files are handled differently. The obvious detail is that the path separator is a backslash instead of a slash. But there are even more differences, e.g. the attributes of a file are queried by `GetFileAttributesA()` instead of using `stat()` and symbolic links are not supported.

The most challenging part was the adaption to the networking stack. In contrast to other platforms, it must explicitly initialized. Several sockets options are set differently or simply does not exist. Error handling must evaluate the result of `WSAGetLastError()` instead of `errno` and process pipes does not even exist. In IBR-DTN, these are used to release blocking `select()` calls at any time. Our solution to port that feature was to replicate the functionality of pipes using sockets.

Most of the Windows®-specific code is embedded within the `ibrcommon` library and activated during compilation. But some differences made it necessary to implement Windows®-specific code within the daemon package. Due to a different scheme for binding sockets to multicast addresses, the IPND component implements different binding approaches for Windows® and other systems. While it is

necessary on LinuxTM and OS/X[®] to bind to a multicast address in order to receive multicast datagrams, it is sufficient in Windows[®] to just join the desired group. A separate binding is not required and would even lead to an error. Further, some details are necessary to read the host-name on start-up and to identify networking interfaces. Finally, we created a native Windows[®] service to run IBR-DTN in the background. This way, the bundle node is fully integrated within the Windows[®] instance and started even before the user performs the login.

While the port compiles on Windows[®] and seems to work, it is not well-tested and considered as experimental. Further tests and thorough investigations are necessary to get the port stable.

6.2 Performance

In networks with intermittent connections the throughput is especially important, as the goal is to take as much benefit from transient connections as possible. Thus, we were interested in a performance comparison of IBR-DTN and other existing DTN implementations. We selected the two major competitors, DTN2 and ION, and evaluated a single-hop transmission scenario. The results have been published in [PMSW11b] and in more detail as technical report [PMSW11a].

In this part of the evaluation we have concentrated on the throughput that can be achieved in different scenarios. We have evaluated the performance of the API, several types of storages, the raw throughput using a GBit LAN connection, and finally the effect of the TCP-CL segment length onto the throughput.

6.2.1 Experimental Setup

All experiments were conducted in a controlled environment. As target for all implementations we used computers equipped with an Athlon II X4 IV 2.8 GHz including 4 GiB RAM running UbuntuTM. The computers are equipped with a Samsung F3 500GB (HD502HJ) hard drive connected to the on-board SATA controller. The computers are connected using 1 GBit Ethernet. For the basic throughput tests we did not simulate bandwidth-constrained or disrupted links, as our goal was to test the raw performance of DTN implementations rather than their functionality in disrupted networks.

On the PCs we used different physical Ethernet ports for traffic related to the BP and those used for controlling and monitoring the experiments. The GBit NICs for BP traffic used a Realtek (RTL8111 / 8168B) controller. The raw TCP throughput achieved by this set-up is \sim 940 MBit/s which has been measured using iperf [DKo8]. In this work we used DTN2 version 2.7, IBR-DTN version 0.6.3

and ION version 2.4.0. All implementations have been measured using their respective default configuration unless otherwise noted.

For ION, we have removed the artificial rate limit for the TCP-CL that otherwise prevents ION from exceeding 90 MBit/s. To prevent missing bundles, we had to enable reversible transactions in the storage system.

6.2.2 API and Bundle Storage Performance

This experiment focuses on API and bundle storage performance and did not involve any network transfers. While in classical networking protocols there exists a code path from the application to the hardware driver with probably only simple and small ring buffers between the layers, a DTN implementation has to provide some form of permanent or semi-permanent storage, which keeps and manages bundle data and accompanying meta information. Therefore, submitting packets from the application layer to the network is a more heavyweight operation in DTN protocol implementations compared to other networking protocols. It is clear that the speed at which a DTN implementation can put bundles into the storage or retrieve them will also fundamentally limit the maximum bandwidth the bundle node can sustain under ideal conditions.

On each node we measured the time to store and receive 1000 bundles of varying payload size. For each payload size we performed 10 runs. The plots show the arithmetic mean bandwidth and bundle frequency of all ten runs including the standard deviation. The bundle nodes have been restarted after performing the 10 runs for each payload size, to prevent that old runs can influence the following measurements. For DTN2 we used dtncat for sending and dtncat for retrieving bundles. For IBR-DTN we use dtncat and dtncat. For ION we use bpsendfile and bprecvfile. The tools were used (IBR-DTN) or modified (ION, DTN2) in such a way, that a single call was sufficient to create or retrieve the 1000 bundles en-bloc so we do not measure the overhead of starting the tools over and over again.

For each implementation we measured the different available storage backends. This experiment gives a good upper bound on the performance an implementation can reach: A DTN implementation can not sustain linespeeds larger than its maximum performance receiving new data locally through its API. While for “common” networking protocols such as TCP the API (bsdsockets) normally is only a small layer of code which ties directly to the kernel via syscalls, and thus is only speed limited by the hardware, all tested implementations in this work are pure user-space implementations. IBR-DTN and DTN2 use a socket-based API to connect applications to the bundle nodes. IBR-DTN can use Unix Domain Sockets instead of a TCP socket when operating locally, but both approaches are more costly than

using a syscall interface. ION uses a shared-memory approach to facilitate inter-process communication (IPC) between applications and the core components. Independent from the API all implementations have to store received bundles in some kind of storage, which means that API performance can be IO-bound. This is different from protocols such as TCP, where only a relatively small amount of in-flight data is cached and applications are blocked when the internal buffers are filled.

The results of these tests can be seen in Figures 6.2, and 6.3 (DTN2), in Figures 6.4 and 6.5 (IBR-DTN), and in Figures 6.6 and 6.7 (ION). The logarithmic x-axis shows the size of the bundle payload. Solid lines indicate the number of bundles processed per second, dashed lines indicate the throughput to or from the bundle node in MBit/s. Each figure is presented twice, either with linear or logarithmic y-axis.

For the send case it can be seen, that for small bundle sizes all implementations are limited by the bundle frequency, i.e. the overhead for processing the individual bundles. Figure 6.2 shows that DTN2 reaches a bundle frequency of around 407 bundles/s for bundles $\leq 10\,000$ bytes and memory storage. For larger bundles, the frequency decreases as expected since now the tests are limited by the bandwidth provided by the storage backend. DTN2 reaches a maximum send throughput of 1140.1 MBit/s for memory-based storage with disk storage being gradually slower. In Figure 6.4, IBR-DTN shows a similar behavior whereas the bundle frequency starts to decrease for bundles ≥ 1000 bytes. However, the overall frequency is significantly higher for smaller bundles while the frequency for larger bundles is almost the same because the bandwidth of the hard drive or memory is the limiting factor. IBR-DTN shows an interesting behavior since the throughput reaches its maximum for bundles of 2 000 000 bytes. This is caused by disk caching mechanisms and is a side effect of the 4 GiB of RAM in our test machines. ION's bundle send frequency in Figure 6.8 is comparable to DTN2 for smaller bundle sizes. However, ION can sustain the frequency even for larger bundles and consequently reaches a throughput of more than 7000 MBit/s. All in all, IBR-DTN has the highest bundle frequency for smaller bundle sizes while ION reaches the highest throughput for larger bundles.

The receive test shows comparable results: DTN2 (Figure 6.3) and IBR-DTN (Figure 6.5) are bundle frequency limited for small bundles. However in contrast to the sending case here DTN2 achieves significantly higher bundle frequencies compared to IBR-DTN. In both cases throughput rises, reaches a maximum and falls back to lower speeds. The initial increase followed by a decrease in throughput can again be explained by caching: For smaller bundle sizes, all received bundles fit into the file system-cache and can thus be retrieved very fast. For larger amount of data performance is again limited by available disk bandwidth. This can be seen by the

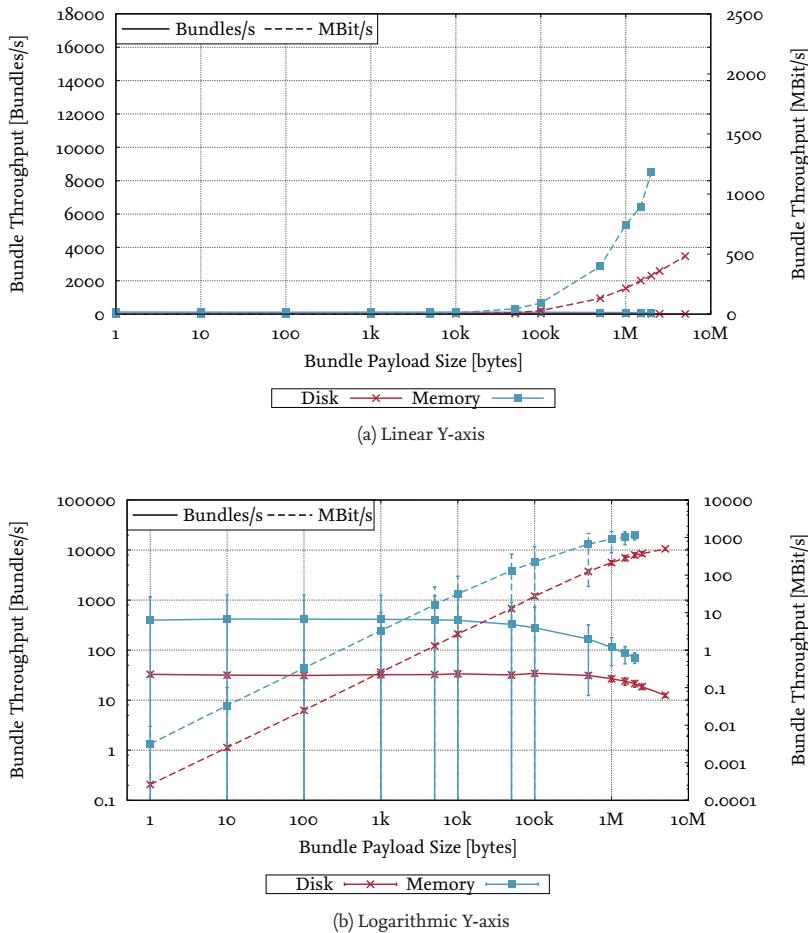


Figure 6.2: DTN2 API send performance

IBR-DTN memory storage performance that does not write anything to disk and thus does not fall back to disk bandwidth. ION (Figure 6.7) again has a significantly lower bundle frequency with an interesting behavior. The frequency is the highest for bundles between 1000 bytes and 100 000 bytes whereas it is lower for smaller and larger bundles. For the receive case, ION is not able to sustain the bundle

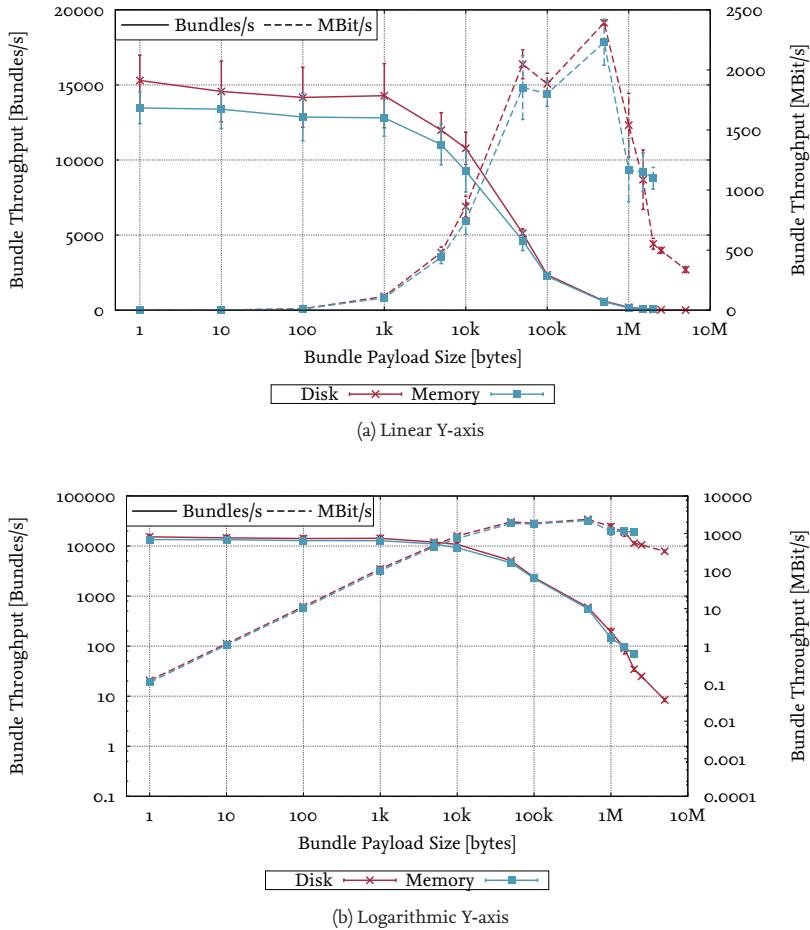


Figure 6.3: DTN2 API receive performance

frequency for larger bundles and reaches a maximum throughput of slightly above 1000 MBit/s.

A direct comparison of the send and retrieve performance of the disk-based storages is given in Figure 6.8 and Figure 6.9. Furthermore, Figures 6.10 and 6.11 show a comparison of the API send and receive performance for memory-based storages.

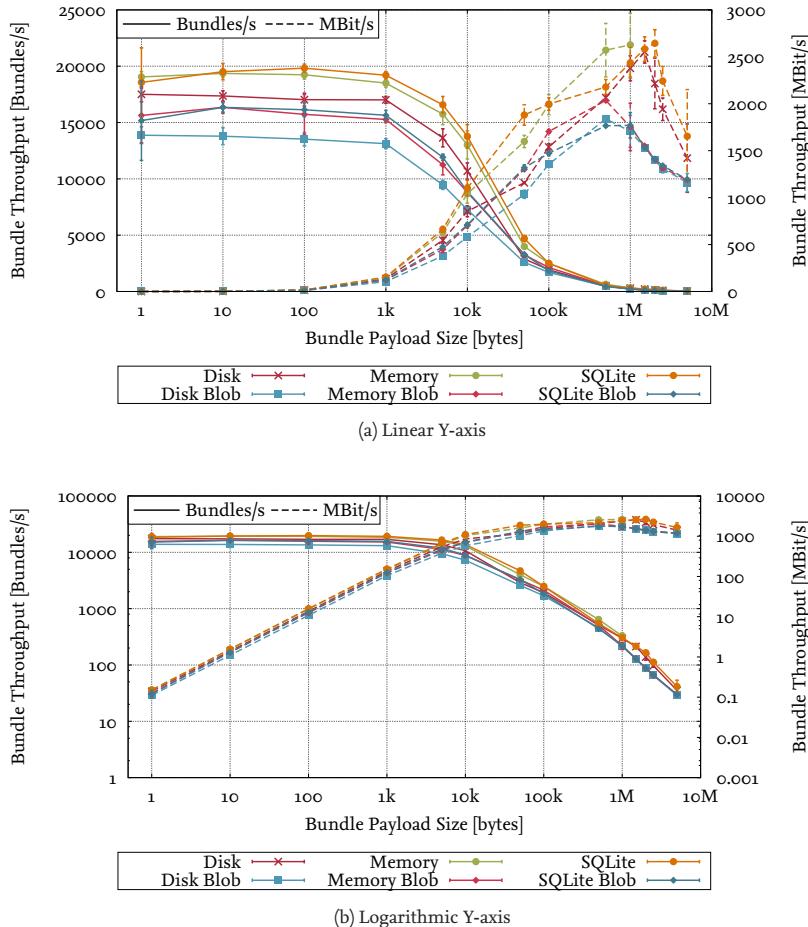


Figure 6.4: IBR-DTN API send performance

In summary it can be seen that when dealing with large amounts of data the underlying storage limits the bandwidth a bundle node can sustain over a period. A low bundle processing overhead is important to deal efficiently with small bundle sizes and a low amount of total data. For the storing case, IBR-DTN outperforms DTN2 and ION in this test, for the retrieve case it is vice versa. However, since the bundle frequency DTN2 achieves when storing bundles is lower than IBR-DTN's

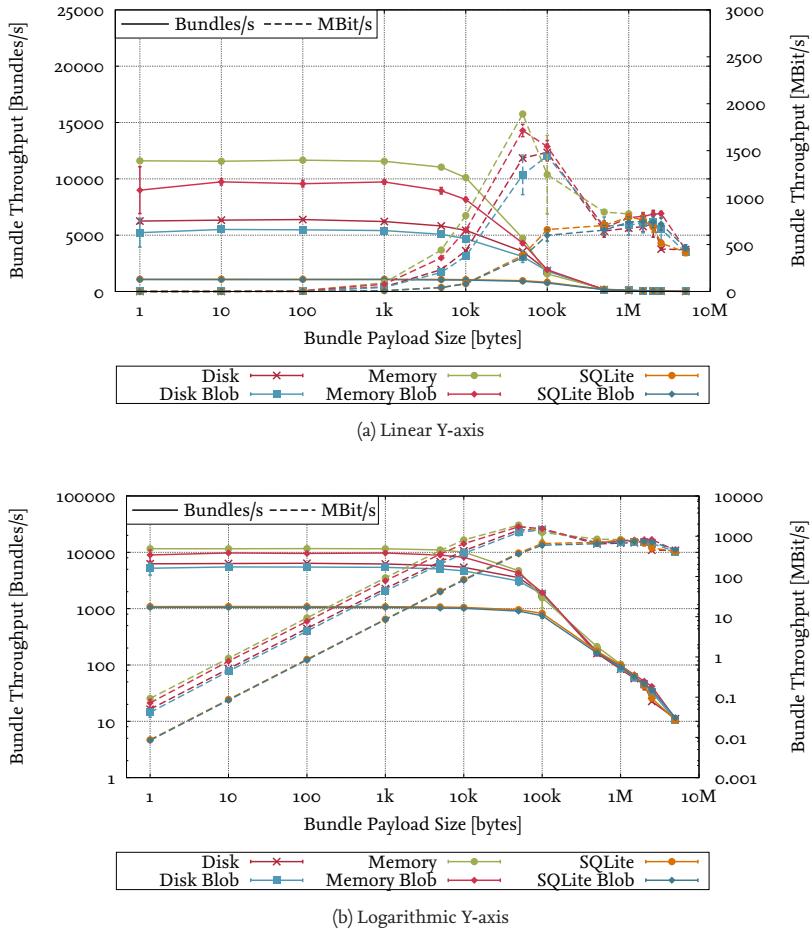


Figure 6.5: IBR-DTN API receive performance

bundle frequency upon retrieving, it is to be expected that in a situation where bundles are continuously being generated, transmitted and consumed a IBR-DTN set-up should outperform a DTN2 set-up, while a combination of an IBR-DTN sender and a DTN2 receiver might reach even higher performance. This is based on the assumption that both implementations perform equivalently well with regard to

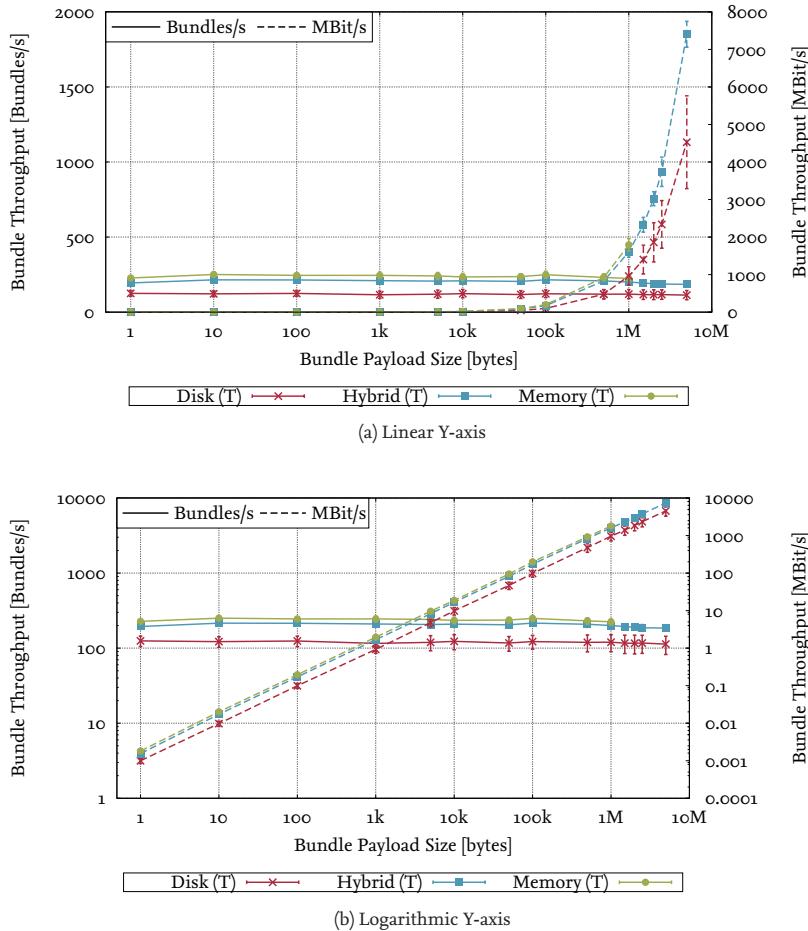


Figure 6.6: ION API send performance

performance and efficiency when transmitting bundles. We will look into the network throughput in Section 6.2.3.

The extremely high bandwidth achieved by ION in the sending case can be explained by the ION architecture that uses a shared-memory approach for communication between different parts of the implementation. We assume that ION has not touched the disk for the Memory and Hybrid storages, as the achieved throughput

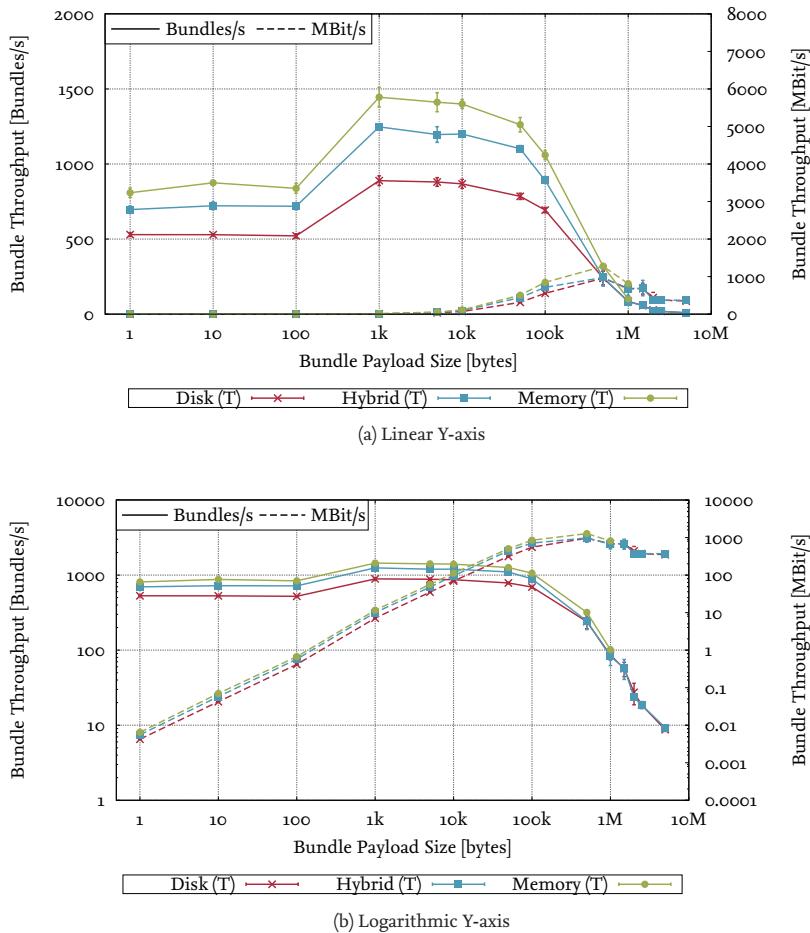


Figure 6.7: ION API receive performance

by the hybrid storage for large bundles is well above even the streaming capability of the used hard drive.

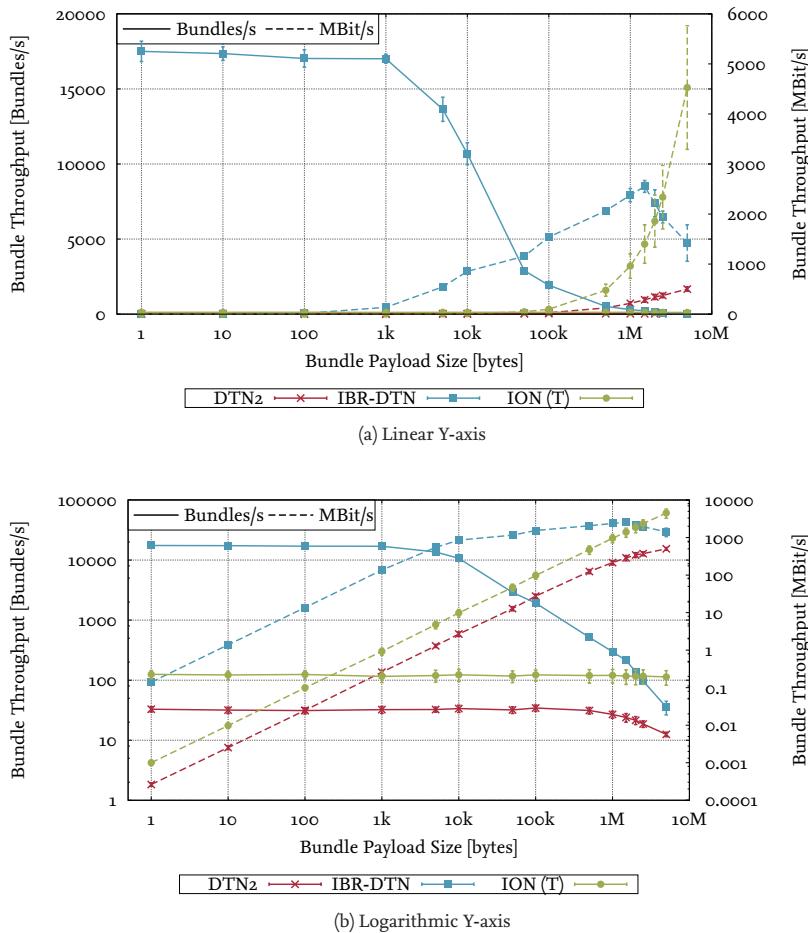


Figure 6.8: API send performance comparison with disk storage

6.2.3 Network Throughput

In this experiment we measured the network performance of the different implementations. We measured throughput between two nodes that are connected via GBit Ethernet. While a GBit link might be uncommon for typical DTN applications, it shows the ability of a bundle node to saturate a given link. Failing to reach

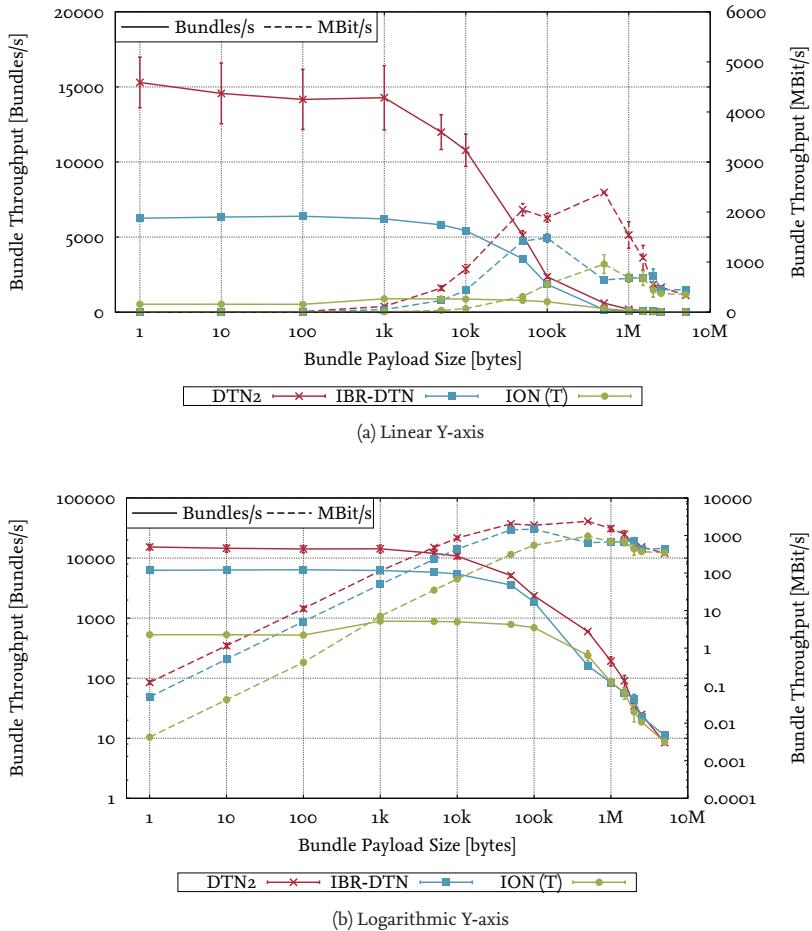


Figure 6.9: API receive performance comparison with disk storage

high bandwidths in this experiment indicates that bundle processing overhead in a given implementation might be too high. This will not only pose a problem with a high bandwidth link, but it could also preclude a resource-constrained node to saturate a lower bandwidth link.

On the sender node we injected 1000 bundles for the receiver. After the bundles had been received by the sending bundle node, we opened the connection and

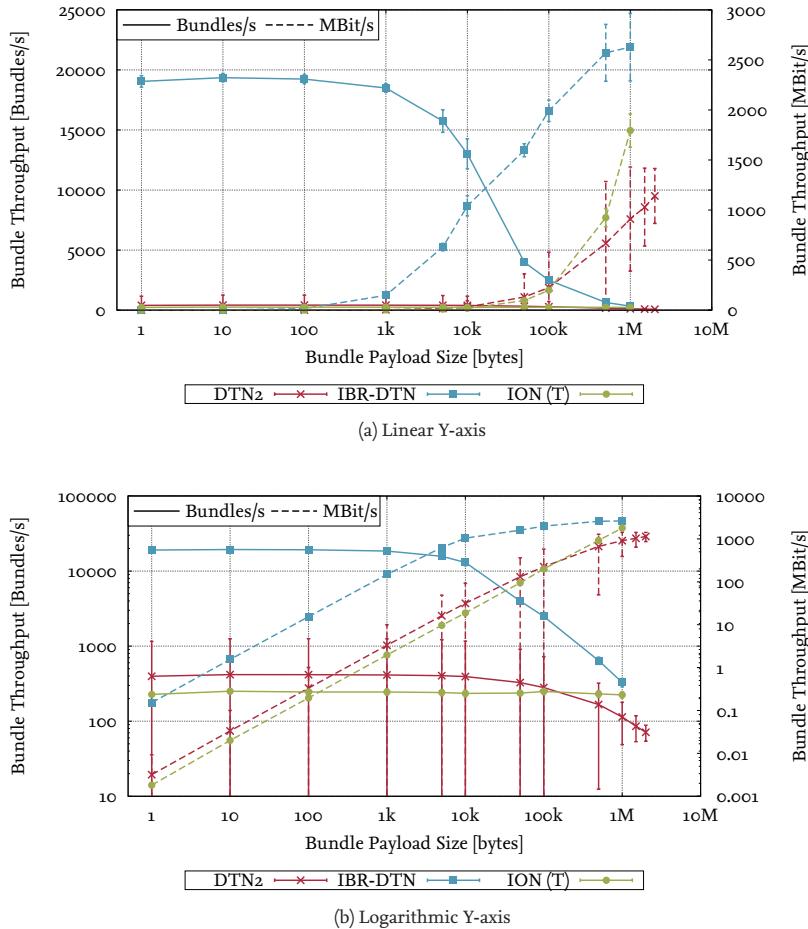


Figure 6.10: API send performance comparison with memory storage

measured the time until all bundles have arrived at the receiver. For each payload size we performed 10 runs. The plots in Figures 6.12, 6.13 and 6.14 show the average of all ten runs as well as the standard deviation. The bandwidth plotted is application layer bandwidth, i.e. it only considers the size of the payload, not protocol overhead. The software has been restarted after performing the 10 runs for each

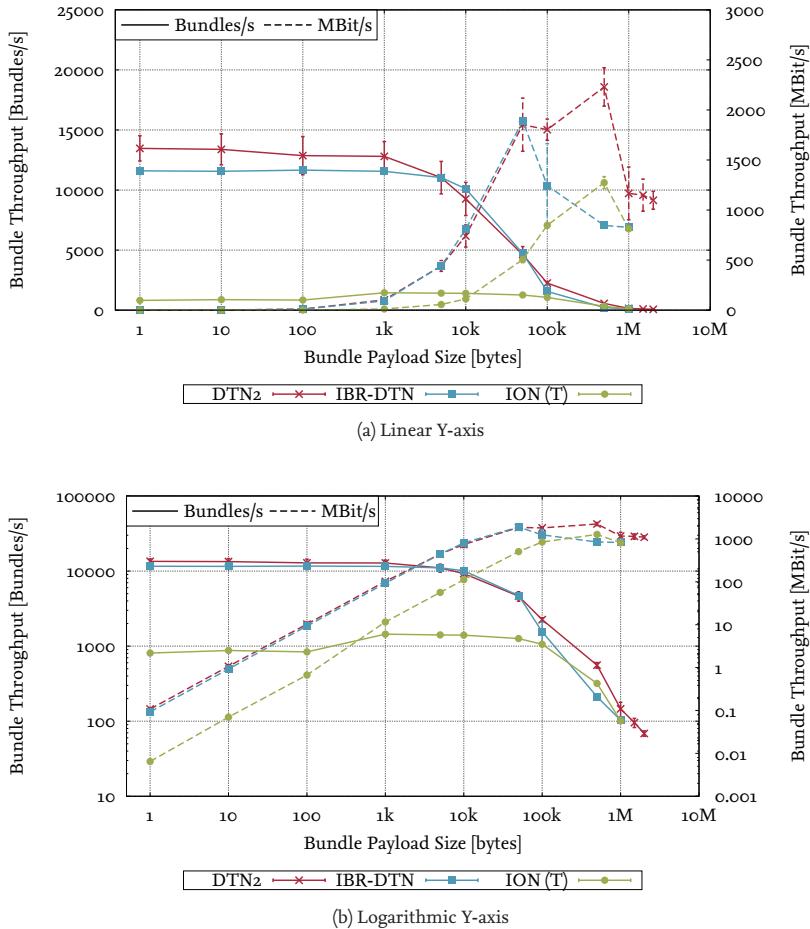


Figure 6.11: API receive performance comparison with memory storage

payload size, to prevent any influence that old runs might have on the following measurements. This experiment uses the TCP Convergence-Layer.

It can be seen, that IBR-DTN comes close to the theoretical limit of the link (940 MBit/s) for large bundle sizes with a throughput of up to 843.341 MBit/s for disk storage. DTN2 reaches a maximum of 719.977 MBit/s with memory storage and ION falls short with 448.833 MBit/s. In fact all storages perform very similar

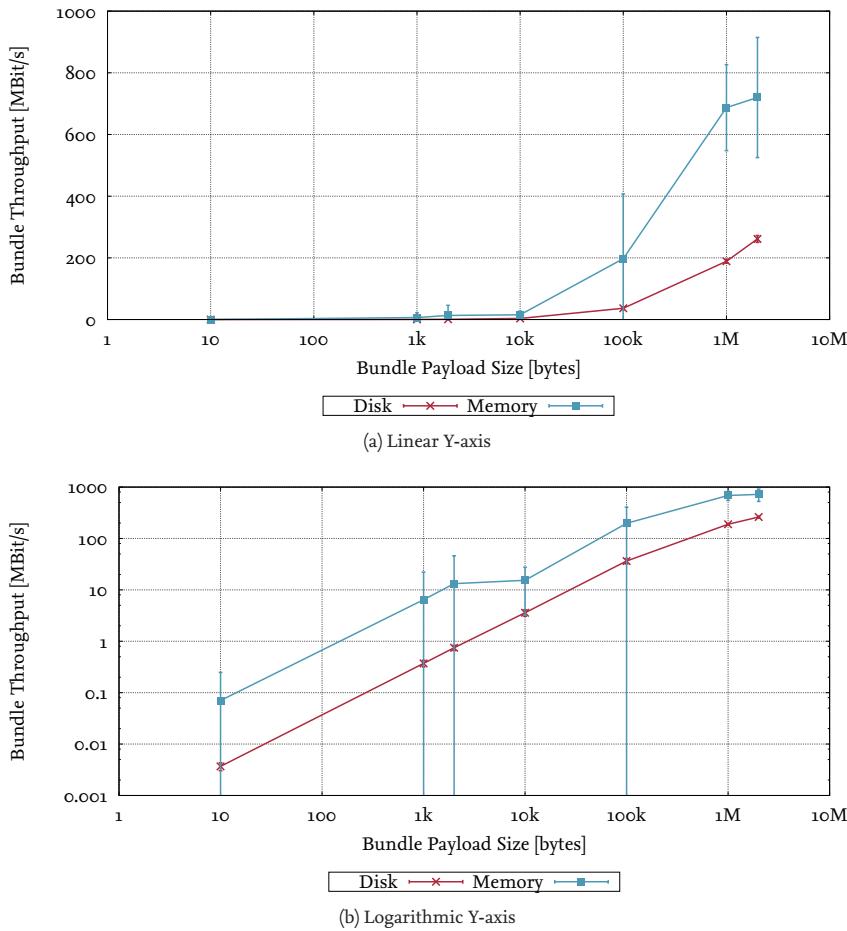


Figure 6.12: DTN2 network throughput

for ION, which indicates that the responsible bottleneck is not located in a specific storage module.

For small bundle sizes DTN2's (Figure 6.12) disk and memory storage achieve almost the same throughput, which indicates that the throughput is bounded by processing overhead. For bundles $\geq 10\text{ kByte}$ DTN2's disk storage achieves lower performance than memory storage, which indicates that for these sizes the

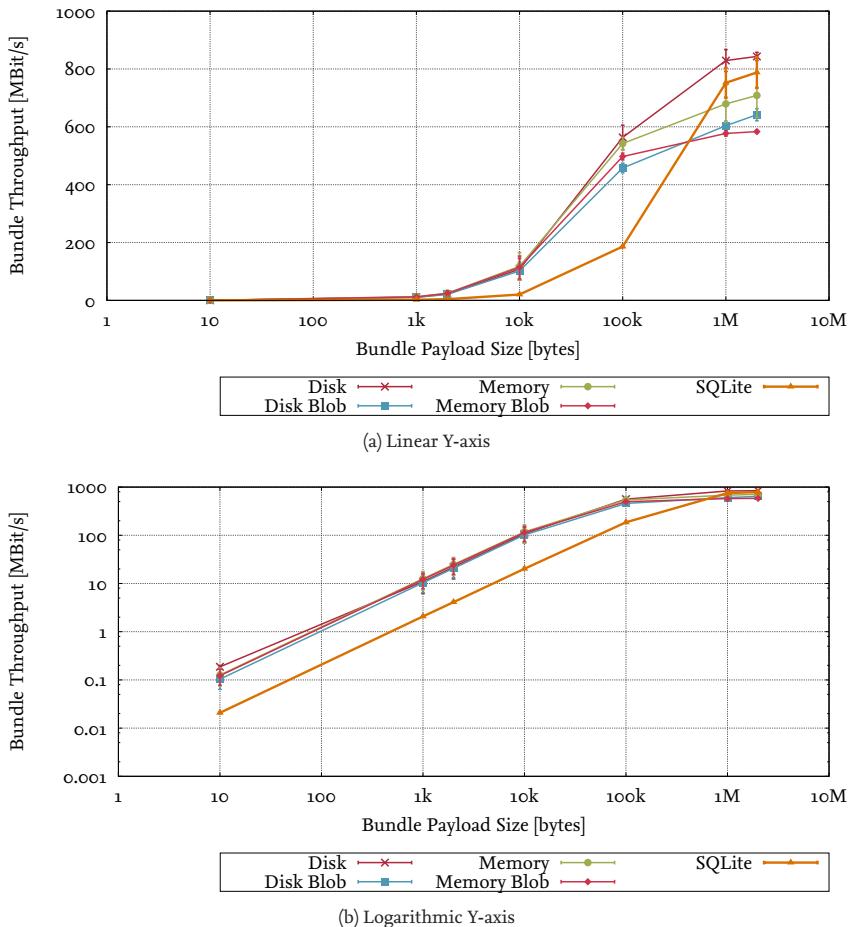


Figure 6.13: IBR-DTN network throughput

throughput of the storage engine limits performance. The variances for DTN2's memory storage are extremely high. Upon further investigation, we discovered that after each run using memory storage DTN2 gets gradually slower. More details on this issue can be found in Section 6.2.5.

IBR-DTN (Figure 6.13) shows a similar behavior where the throughput increases with larger bundle sizes, however in absolute terms throughput is always signif-

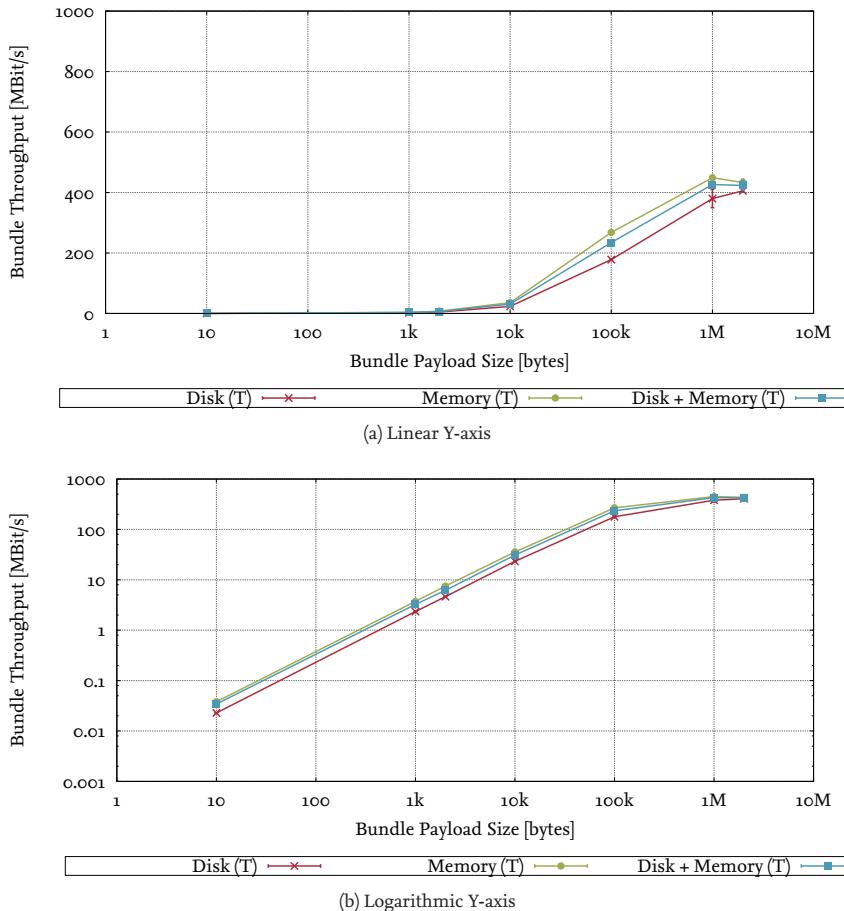


Figure 6.14: ION network throughput (with transactions)

icantly higher than DTN2's. In IBR-DTN the performance is more consistent between different storage backends, while DTN2 performance is much higher when using the memory-based storage compared to its disk-based storage. Interestingly, IBR-DTN shows higher throughputs for disk-based storage compared to memory storage. After further investigation this turned out to be caused by the respective implementation of the storage modules. Disk-based storage uses multiple threads and makes good use of the multiple CPU cores in our test machines while memory-

based storage only uses one thread and subsequently only one core. The SQLite storage of IBR-DTN is the slowest for smaller bundles but then picks up and is only gradually slower than the disk-based storage for bundles $\geq 1\,000\,000$ bytes.

ION (Figure 6.14) shows a performance between the two storage backends of DTN2. It is noteworthy, that the two ION storage backends have comparable performance for all bundle sizes. This fact implies that the bottleneck in ION is not within the storage module but in the processing of the bundle node. While DTN2's memory-based storage is significantly faster than ION's, DTN2's disk-based storage is also significantly slower than ION's.

A comparison of the throughput of all disk-based storages can be seen in Figure 6.15. Clearly in the network throughput test IBR-DTN is the winner, while the picture between DTN2 and ION is not so clear.

6.2.4 TCP-CL Segment Length

As seen in Section 6.2.3 neither implementation was able to saturate the maximum bandwidth of the link. Apart from performance limits in the bundle nodes, unsuitable parameters used in the BP can have an effect on throughput. As DTNs are mostly deployed in wireless networks it can be assumed that the implementations have been tested in such set-ups, where the available bandwidth is much lower than in our test cases.

The parameter which is most likely to influence throughput is the TCP-CL segment length. Essentially, a larger segment length means less processing overhead: Executing fewer TCP send sys-calls saves processing overhead in the bundle node and avoids unnecessary switches between user-space and the kernel. Also, since every segment is acknowledged by the receiver, for larger segment lengths there is less overhead processing the TCP-CL acknowledgements.

The TCP-CL segment length defines how many bytes the TCP-CL will transmit as a single block. This defines the amount of data that is acknowledged by a single ACK and also influences the granularity of reactive fragmentation. It is to be expected that by using a larger TCP-CL segment size a bundle node can saturate a higher bandwidth. Both DTN2 and IBR-DTN use 4096 kBytes as default segment length, while ION does not segment bundles at all. To show the influence of the TCP-CL segment length, we have measured the achievable throughput with the default value of 4096 bytes and an extreme value of 2.5 MB.

Figure 6.16 shows the influence of the segment length for IBR-DTN. It can be seen, that for bundles larger than the default segment length of 4096 bytes the performance is increased significantly. The fastest measured throughput was for memory-storage with the increased segment length of 2.5 MB and a bundle size of

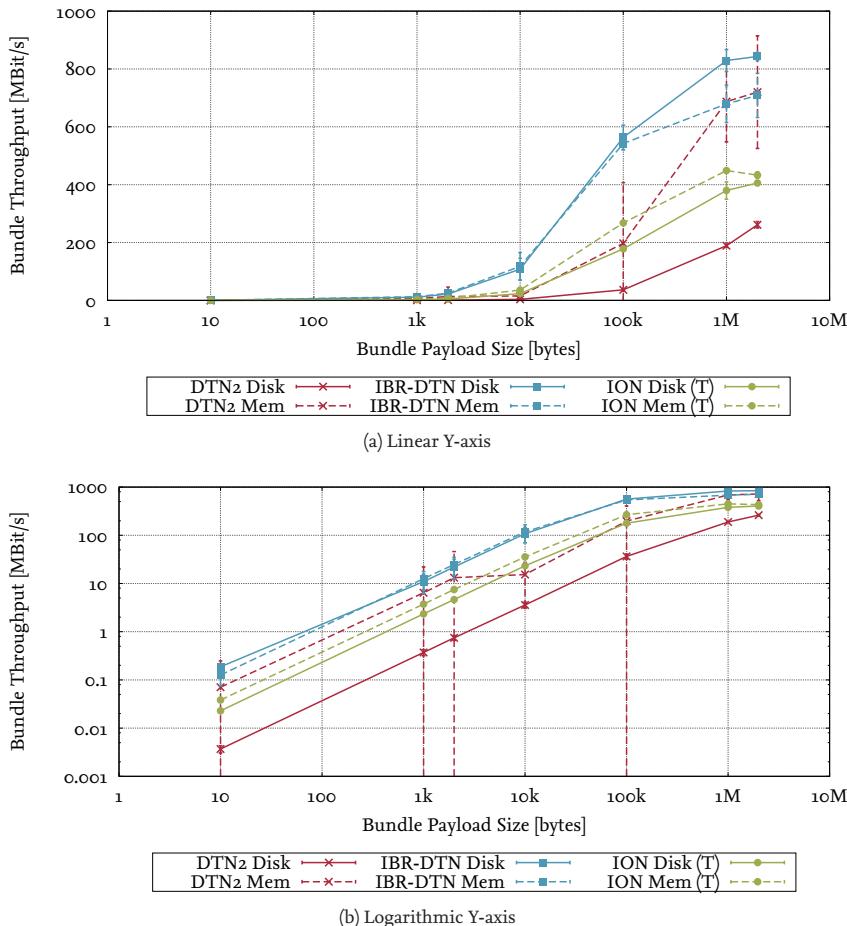


Figure 6.15: Network throughput comparison

1 MB. In this setting, IBR-DTN reaches a speed of 935.200 MBit/s which is 99.489 % of the theoretical maximum of the link.

This measurement has clearly shown that tweaking the TCP-CL segment length can significantly increase the throughput of a BP implementation.

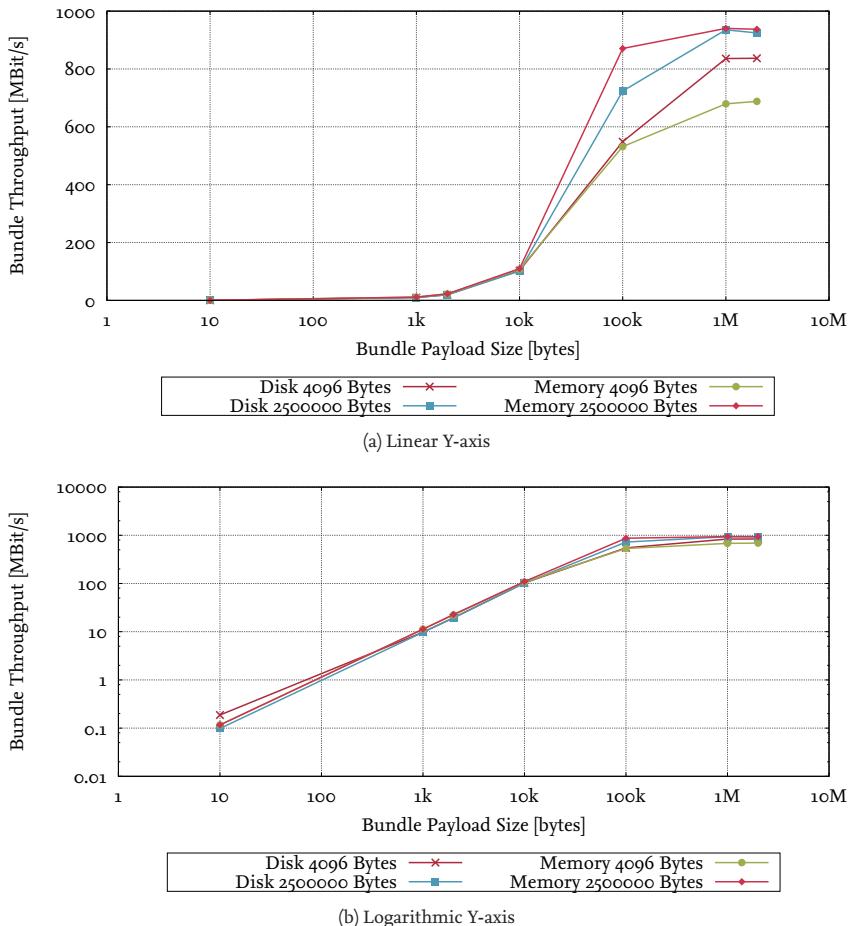


Figure 6.16: Impact of TCP-CL segment length onto Throughput for IBR-DTN

6.2.5 DTN2 Throughput Variances

As shown in Section 6.2.3, DTN2 exhibits high variances between different runs of the same bundle size and storage, especially for medium-sized bundles when using memory storage. Figure 6.17 shows the throughput over different measurement runs. Especially for the smaller bundle size a clear trend of decreasing speed in

each measurement run can be seen. For the big bundle sizes the behavior gets more erratic.

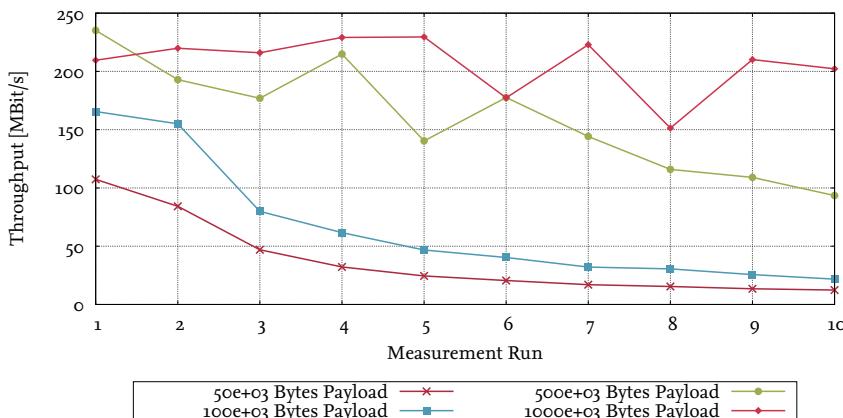


Figure 6.17: DTN2 throughput with different payload sizes over multiple measurement runs.

This behavior might indicate some resource management problems within DTN2, i.e. some resources might not be freed when not needed anymore. As we restarted all bundle nodes for each new set of parameters, the results presented before are not influenced by the order of the experiments.

6.2.6 ION Storage Variants

ION allows a user to configure the underlying storage using a number of flags that enable or disable storage types. That way it is possible to enable multiple storage options at the same time. Figure 6.18 shows the performance of the ION API with different storage configurations and bundle sizes. It can be seen, that memory storage has the highest throughput in bundles and bytes with significant benefits for bundles sizes below 1000 bytes. Also, the combination of disk and memory storage (denoted as hybrid) is significantly faster than normal disk storage. When we have enabled reversible transactions, the performance drops at least an order of magnitude, but still memory storage is faster than hybrid that is still faster than disk storage.

Although this high performance is promising, we were unable to run our throughput tests without reversible transactions enabled, as we experienced severe bundle loss without any error messages under these conditions. Therefore, for our experiments we had to resort to the slower alternative.

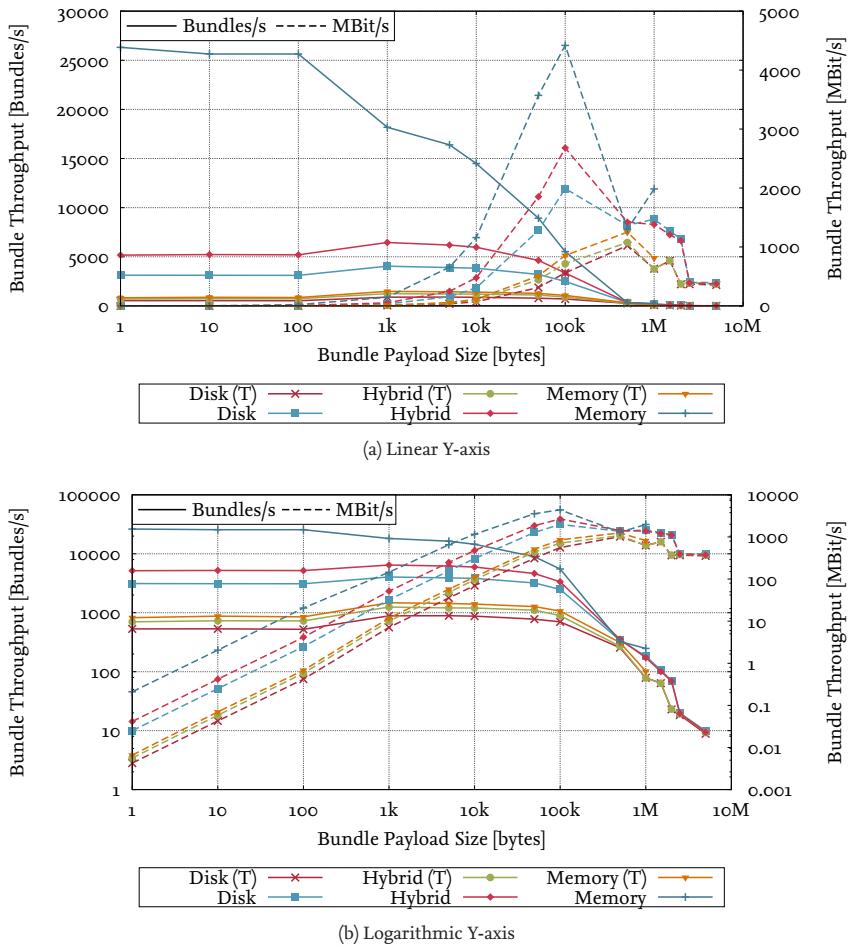


Figure 6.18: ION comparison of the API performance for different storage configurations.

6.2.7 Conclusion

To the best of our knowledge, we have performed the first extensive study of performance and interoperability of the three relevant BP implementations. We have looked at the throughput that can be achieved using TCP-CL as well as the effects of

the storage systems available for the implementations. Finally, we have evaluated the effects of the TCP segment length onto throughput.

One of the major influencing factors is the underlying storage system that effectively limits the achievable throughput for links with high bandwidth. Disk storage can be considered the default option for DTNs and is not only influenced by the throughput of the disk but also by the concrete implementation and caching mechanisms of the underlying operating system. Especially when the total amount of bundles exceeds the available volatile memory, this can be a bottleneck.

6.3 Continuous Integration

The principle of Continuous Integration (CI) can increase the overall software quality by automatically and continuously building the code of a software repository as if it is done for release builds of the product. Such a system can keep release cycles short by disclosing integration issues immediately after they have been caused. Detected issues are automatically reported to the responsible developers, so that they can immediately react. A very popular system to manage automatisms to build and test software is the extendable open-sourced continuous integration server Jenkins. Within that system, a repeatable process description for automated execution is called a »Job«.

During the development of IBR-DTN, we moved the complete code into a repository of the distributed revision control and source-code management system »GIT«. Further, a Jenkins server has been set-up to continuously build and test the code. Multiple jobs were created to realize and publish nightly-builds as well as the manually triggered releases.

6.3.1 Building

The build process is the first step of CI. A job monitors the master branch of the GIT repository and is activated as soon as a new bunch of changes appears. Then a process starts to build the code with all features enabled. This is necessary to cover as much as code as possible. Otherwise, these parts stay untested. As next the unit-tests as described in Section 6.3.2 below are executed. If these pass, then the CI continues with testing the build for different platforms.

As presented in Section 6.1, IBR-DTN was ported to several platforms including OpenWrt, OS/X®, Android™, and Windows®. To keep those ports viable there exist Jobs to build the software on each platform and finally check if the unit-tests pass on them. Since the Jenkins server itself is running on Linux™, it would be very complicated to set-up cross compilation for OS/X® and Windows® on Linux™. A better approach is to build directly on the platform to test. Instead of

setting up multiple Jenkins server, worker slaves can be executed on different platforms. Workers are small Java-based programs which connects to the Jenkins master server in order to get controlled remotely to execute Jobs. While building for Windows® requires a Windows®-based slave and building for OS/X® requires even an Apple Computer, building for OpenWrt, Android™, and Raspbian¹ is done on Linux™-based slaves.

The result of the build jobs are typically packages which are ready to get delivered using the platform-dependent distribution paths. In case of Linux™ builds, the result consists of Debian packages for different versions and distributions. These are directly published in a dedicated repository. Users who like living on the bleeding-edge of software development can simply add the repository URL to their system. As result, nightly-builds are directly delivered automatically without further user interaction. Since the OpenWrt platform is a major target for IBR-DTN, we realized something similar for packages dedicated to those systems. A Jenkins job builds packages for different processors variants and publishes them using a webserver. For other Linux™-based systems without a public toolchain, we utilized Buildroot to generate static binaries without any further dependency. Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux™ systems using cross-compilation. Binaries generated by this toolchain are capable in running on every Linux™ system as long as the processor type matches to the binary. Builds for Windows® result in an executable installer based on the Nullsoft Scriptable Install System. The binary of the build is made publicly available via an URL. The result of the Android™ build is an Android™ Package (APK). Beside publishing it as URL, it is also pushed directly into the alpha program section of the Google Play Store. This allows a user or tester to join the alpha program in order to receive the latest nightly-builds without any further user-interaction. The build for OS/X® does not generate an installer or something similar, because there does not exist a standard way to keep third-party packages of system components up-to-date. Instead, recipes to build the software using the MacPorts Project are published to a dedicated repository. User with an installation of MacPorts can simply add the repository to their system and then install/upgrade the software using a single command.

Distributing nightly-builds are not only a nice give-away for users who like to live on the bleeding edge. In fact they make the casual testing much easier and significantly increases the amount of potential software testers. In case of research, the availability of packages with the latest feature-set eases the set-up of experiments.

¹Debian distribution for the ARM-based Raspberry Pi, a credit-card sized computer.

Since the build-chain for nightly-builds does not differ from those for release builds, it is possible to automate large parts of the release cycle. For IBR-DTN, we utilized the tag-feature of GIT to mark software branches with release numbers. Those tags are automatically picked up by a Jenkins job and the release build begins. Different to the nightly-builds the release builds are not published directly. Further manual testing is expected before the release is considered as stable and safe to publish.

6.3.2 Unit-testing

Unit testing is a software testing approach to test the behavior and functionality of individual units of source code. A unit is tested by several testing procedures. Each of them specifies input data for the unit and compares the output with the expected behavior. Generally, units can be a module, a component, or the complete software package at once. But most of the time it is hard or even impossible to write tests which cover all possible cases of a very complex module. Thus, it is better to test individual functions or procedures separately and thorough. Since unit-testing can be processed automatically and often results in a useful error description, it is an important step during the CI procedures.

Within the packages of IBR-DTN, unit-tests are based on the library `cppunit` and integrated into the `autoconf` routines of the build process. By simply calling »`make check`« within the directory of the package to test (`ibrcommon`, `ibrdtn`, `ibrdtnd`, or `ibrdtn-tools`), the unit-tests are compiled and executed. In case of a failure, the result is printed to the screen and a detailed testing report is stored in a log file. An example for a positive result is shown in Listing 6.3.

Listing 6.3: Result of unit-tests of `ibrcommon`

```
1 =====
2 Testsuite summary for ibrcommon 1.0
3 =====
4 # TOTAL: 1
5 # PASS: 1
6 # SKIP: 0
7 # XFAIL: 0
8 # FAIL: 0
9 # XPASS: 0
10 # ERROR: 0
11 =====
```

While unit-tests can evaluate expected behavior of components, a positive result does not proof the correctness of the whole package. Unit-testing is only effective if all or at least a significant amount of code is covered by the tests defined. To

measure how much code is actually covered by the defined unit-tests, automated tools like gcov can measure the code coverage and can very precisely hint to uncovered parts of the software. With this knowledge it is possible to write dedicated tests to cover the remaining code branches in order to converge successively to a completely tested source-code.

6.3.3 Performance Testing

Beyond unit-tests to check the consistency of small code units, the behavior of the overall system must be evaluated. Even if all units act correctly as individuals, it is possible that they do not cooperate as intended. Testing this is often a manual job done by testers which tends to report non-reproducible behavior due to missing details.

To integrate a more thorough testing into the CI process, we automated the network throughput measurements presented in Section 6.2.3. Every build which passed the unit-tests is being tested by a Jenkins job responsible for performance measuring. The job utilizes three dedicated hosts. Two of them are connected by a dedicated GBit Ethernet connection and used as sender and receiver. The third host is used as controller. It also compiles the software revision to test and deploys it together with configuration files to the sender and receiver. Once the preparation is completed, the performance measurement is performed as described in Section 6.2.3. The results are published to a database which is used to generate graphs for comparison with different runs.

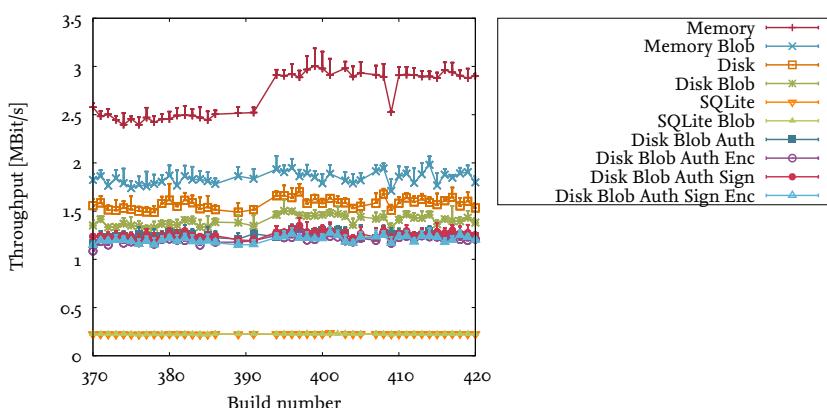


Figure 6.19: Result of the CI performance measurements with a payload size of 100 bytes

Figure 6.19 shows the result of the measurements with a payload size of 100 bytes for revisions from build 370 to 420. The missing data for some build numbers is the result of local networking issues which influenced the overall CI system. In addition to the cases specified for previous measurements, we added more configurations to cover different security related mechanisms. All of them are measured with disk-based storage with enabled BLOB mechanism, because we consider this configuration as relevant for realistic deployment scenarios. The new configurations cover cases with enabled authentication using BABs (Auth), encrypted payload using PCBs (Enc), and signed bundles using PIBs (Sign).

The performance measurement does not only proofs the basic functionality of the software, it is also an indicator to localize the changes to blame in case of a performance impact. On the other side, it is possible to monitor the effect of improvements. For example, the performance enhancement between number 391 and 394 is the result of a refactored bind-pattern for receivers of events. By avoiding switch cases using dynamic casts and the utilization of templates, it was possible to increase the performance significantly.

6.4 Time-synchronization

In this section, we are going to evaluate the DT-NTP algorithm presented in Section 5.2. The approach synchronizes clocks of bundle nodes by distributing time-references within the network. We start by evaluating the algorithm using the Opportunistic Network Environment (ONE) simulator. Since the simulator does not support individual clocks for each node, we need to create a clock model and add it as extension to the simulator environment. Further, we will define three scenarios with varying amount of nodes and density. Finally, we are going to discuss the results and investigate the impact of the algorithm to the average clock offset.

The second part of this section, we reconstruct two of the previously simulated scenarios using an advanced revision of the emulation framework HYDRA [MSW10]. The virtualized nodes instantiated by the framework are based on OpenWrt and run IBR-DTN as they would in real scenarios. HYDRA emulates movement and encounters between the nodes and collects statistical data. As final part of this evaluation, we compare the results with those produced by the ONE simulator.

6.4.1 ONE Simulator

As formerly presented in [MW12], we evaluated the presented algorithm using the ONE simulator [KOK09]. Since that simulator has no support for independent

clocks, we had to extend the software with individual clocks for each simulated node. Further, interfaces to synchronize those clocks are required.

Clock Model

To gain a realistic model for the simulated clocks, we designed them accordingly to the principle of real clocks. Those consist of a crystal with a specific frequency, a tick counter and a register with the current time value. The crystal generates ticks, which increment the tick counter. If a specified amount of ticks is reached, the tick counter is set to zero and the register gets incremented by a predefined amount of time.

To model the simulated clock as realistic as possible, we copied the structure of a hardware clock in software. That means each simulated clock depends on a clock generator. Due to the discrete time steps the simulator is based on, it is not possible to implement a continuously ticking clock generator. Instead, the number of pulses between each query are determined by the hardware crystal.

Each time the `advance()` method is called, the simulated clock queries the generator for the number of pulses. Then it calculates the number of ticks since the last call and the amount of time to add to the local time-stamp. According to the crystal of hardware clocks, we use a predefined frequency to identify the sum of ticks past since the previous call and increase a time variable by a given amount. As on real computer systems, we use a clock frequency of 1 193 182 Hz to generate 1000 ticks per second and increment the time value every tick by 0.001 seconds.

To simulate inaccurate and fluctuating clocks, we have added a frequency error value. On each `advance()` call, this error value influences the number of ticks of the current processing interval and imitates a faster or slower crystal. This error is bound by a maximum of ± 100 ppm and lets clocks drift naturally. Further, the frequency error can change during the simulation using a given error change interval and error change rate.

During the configuration of a simulation scenario, each group of nodes gets an individual clock implementation assigned. Each node in the group creates an instance of that implementation which is considered on every decision which involves the global time. For example, it queries the clock to assign the creation time-stamp to bundles and to decide if stored bundles are expired or not.

Synchronization

On every encounter of one node with another, the synchronization algorithm performs a comparison of both clocks. If the remote clock has higher rating, the offset between both clocks is estimated by simply comparing one time-stamp with the other one. While this is an acceptable approach in a simulator, it would not be

possible to perform that directly in reality. A communication channel would be necessary to exchange messages between the hosts in order to compare the timestamp. As proposed in Section 5.2.2, bundles can be used to send and receive synchronization data. When doing this, we need to consider potential inaccuracies due to unknown transmission delays during that procedure.

However, generated bundles are typically placed into transmission queues on the bundle node before they get transferred to neighbors. Since the ONE simulator does not support priorities, new bundles are only pushed to the end of the queue and never overtake other bundles. Thus, the expected precision would suffer, if the synchronization is performed using bundles as communication channel. Moreover, the approach presented in Section 5.2.2 uses the AEB extension to track the resident time of each bundle. The ONE simulator does not include this extension and thus the approach is not applicable here.

Due to those limitations, we omit the utilization of a simulated communication channel for the exchange of synchronization data and determine the offset between the clocks directly. This offset and the rating of the foreign clock are used to adjust the local clock. Instead of setting the new clock value directly, the simulator adjusts the tick counter to get a smooth convergence to the new determined value as recommended in Section 5.2.1. Additionally, we detect and compensate the wrong clock-skew on each synchronization with a filter similar to the phase-locked loop model referenced by the NTP [Mil85]. Finally, the parameter σ of the local clock is adjusted as defined by the Equation 5.8.

Scenarios

We selected three different simulation scenarios to evaluate the synchronization algorithm. The first one is shipped with the ONE simulator and represents the city of Helsinki with 80 pedestrians, 40 cars and 6 trams. We used two trams as clock references as they might have GPS equipment on the roof. The scenario includes a map-based movement model which selects the shortest path to a random destination on the map. Trams are bound to specific routes, moving with 7 m/s to 10 m/s and have a waiting time between 10 s to 30 s. Cars are bound to streets, moving with 10 km/h to 50 km/h respectively 2.7 m/s to 13.9 m/s and wait between 0 s to 120 s on each destination. Pedestrians can walk anywhere except through buildings. They move with 0.5 m/s to 1.5 m/s and have the same waiting time as cars. As default, each node has a communication interface with a range of 10 m. This is similar to a class 3 bluetooth device.

To provide a scenario which is more comparable to existing papers, we defined a second simulation based on a random way-point (RWP) movement model in a square of 5 km \times 5 km. We added 50 pedestrians moving with 0.5 m/s to 1.5 m/s

and configure the first node of the group as reference. The transmit range of each node is set to 250 m to model connectivity similar to Wi-Fi.

In a third scenario, we adjusted the settings of the second scenario according to the simulation in [CS10]. The only adjustment is the movement area, which raises to $50 \text{ km} \times 50 \text{ km}$. Since there are just 50 pedestrians randomly distributed in the area and the communication range is low compared to the node density, we expect very rare encounters between the nodes. This will have an impact on the accuracy of the clock synchronization. To distinguish between the second and the third scenario, we name the second one »RWP 25« and the third »RWP 2500« according to the size of the movement area.

Results

The most important evaluation metric to be studied here is the offset between the reference clock in the network and a single node. We always consider the absolute values only, because we do not need to distinguish between positive and negative clock offsets. All simulations were done 10 times with different random seeds and the results have been averaged for the analysis.

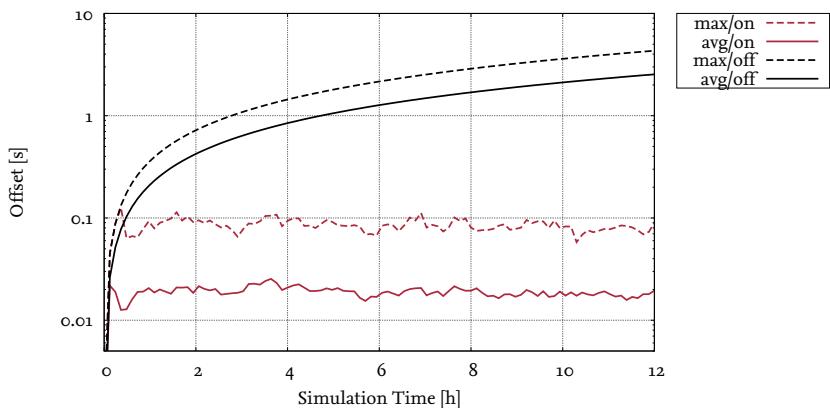


Figure 6.20: Clock offset in the Helsinki scenario with and without time distribution algorithm

In the following, we concentrate on the average and maximum offset of each scenario. The first one is the result of the Helsinki scenario depicted in Figure 6.20. The graphs of the simulations without any time-synchronization is titled with »off«. Since all clocks are set to a random frequency error of 0 ppm to 100 ppm

and there is no algorithm to set any clock to a correct value, the offset raises linearly during the whole simulation. With enabled time distribution algorithm, titled »on«, the raising clock offset stops after a short time and stays with a little variance at about 20 ms in average and 100 ms as maximum.

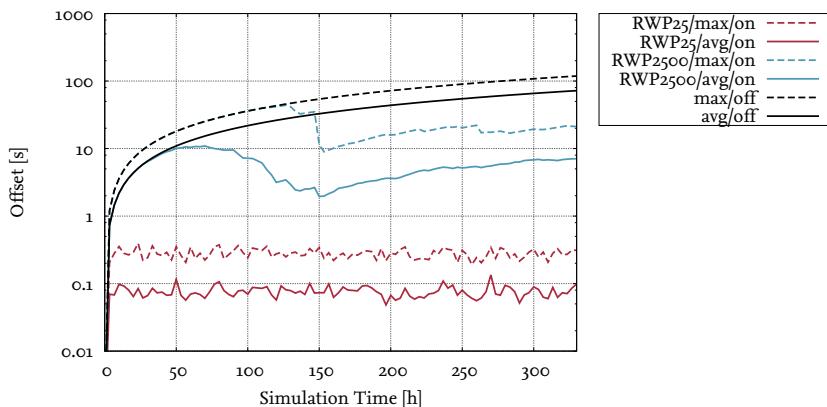


Figure 6.21: Clock offset in the RWP 25/RWP 2500 scenario with and without time distribution algorithm

The results of the RWP 25 and the RWP 2500 scenario are shown in Figure 6.21. Compared to the Helsinki scenario, the clock offset drifts at the same speed because the same frequency error is used. The runs with enabled time distribution shows the same behavior as before but with less accuracy. The average value of the RWP 25 scenario levels out at approximately 80 ms and the maximum at approximately 300 ms. The average and maximum offset is much higher in the RWP 2500 scenario. Further, the obvious dip in the maximum graph of RWP 2500 at 150 h indicates that this is the first time where all nodes were synchronized at least once.

To explain the results of the different scenarios, we have to look at the main difference between them, the encounter frequency. We analyzed the number of synchronization opportunities and determined how often the clocks are adjusted in each scenario. The results are shown in Table 6.1. The global adjustment count is the number of how often a synchronization is performed. Using the number of non-reference-nodes, we can determine how often each node was adjusted in average. The rate in Table 6.1 describes the interval between each adjustment, globally and for each node in average.

Scenario	Helsinki	RWP 25	RWP 2500
Simulation time	12 hours	14 days	14 days
Number of nodes	126	50	50
Reference nodes	2	1	1
Adjustments (global)	31190	49813	396
Adjustments (each node)	251.53	996.26	7.92
Rate (global)	1.38 s	24.28 s	50.9 min
Rate (each node)	171.74 s	20.23 min	42.42 h
Average Clock Offset	20 ms	80 ms	5.9 s
Maximum Clock Offset	100 ms	300 ms	23.3 s

Table 6.1: Adjustment frequency comparison for the scenarios

The results show that the Helsinki scenario has the most and the RWP 2500 the fewest encounters of all. Together with the measured clock offsets, a direct relation between the reachable accuracy and the adjustment frequency is given. Since the clocks have more time to drift away if the encounter frequency is low, the results are obvious.

Skew Variance

As next, we investigate the impact of varying clock-skew. The previously implemented clock model allows us to change the frequency error randomly over a specified period. In the previous experiments the frequency error was set to a randomly chosen value between 0 ppm and 100 ppm at the beginning of the simulation. This time we chose to change the frequency error randomly by an amount of up to 1 ppm every three minutes using the mechanisms described in Section 6.4.1. This behavior represents a temperature change due to rain showers or something similar. Once again, we look at the average offset of all nodes, depicted in Figure 6.22.

The result of this simulation shows only a small impact on the accuracy of the time distribution process with varying random (rnd) compared to a constant skew (const). But we expect that the accuracy and adaption ability depends on the encounter frequency of the scenario. Therefore, we repeated the same test in the RWP 25 and RWP 2500 scenarios. The results in Figure 6.23 show that the varying random skew has indeed an impact on the accuracy of the algorithm. But the impact is only significant if the adjustment rate is very low.

Algorithm Adaption

One ability of the algorithm is to adjust its local rating according to the accuracy of the individual clock. To show the benefit of this mechanism, we simulated the

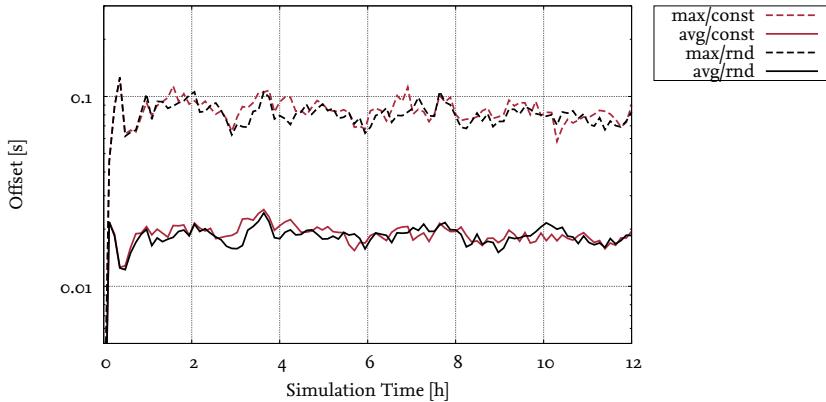


Figure 6.22: Effect of varying skew in the Helsinki scenario

three different scenarios with (on) and without (off) the adaptation of σ as defined in equation (5.8) of Section 5.2.

The Figure 6.24 shows the results of the Helsinki scenario. Here, the mechanism improves the average (avg) and the maximum (max) offset up to 50 %. The RWP 25 scenario (Figure 6.25) exposes a stronger improvement, especially for the average offset. The last scenario (RWP 2500), shown in Figure 6.26, reveals just a tiny impact on the offset. Once again, this might be the result of the low adjustment rate as explained before.

6.4.2 HYDRA

In this section, the previously developed approach is investigated using HYDRA, a virtualized testbed for realistic large-scale network simulations. While simulations as done before only provide approximations of the protocol stack, HYDRA virtualizes nodes running a complete Linux™ system. Mobility models and connection management integrated into HYDRA allow a simulation of various wireless networking scenarios. Our distributed virtualization approach achieves excellent scalability and the automated node set-up makes it easy to deploy large set-ups with hundreds of nodes.

Revision Two

HYDRA, as presented in [MSW10], has been partially rewritten to ease the set-up procedures further. The *Master* is now written in Java and runs in a Tomcat Appli-

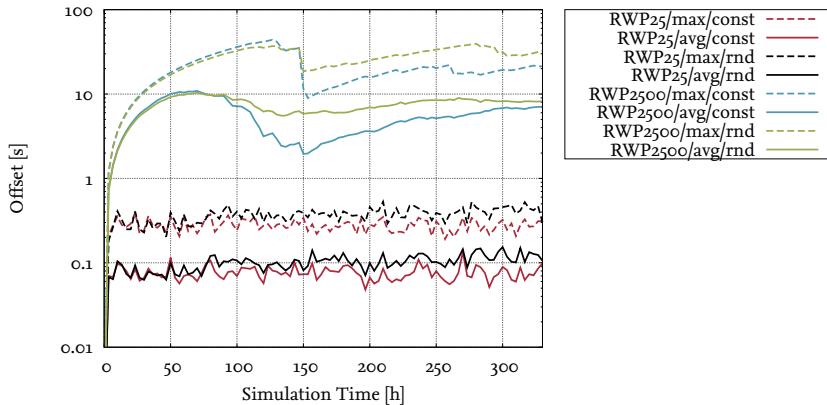


Figure 6.23: Effect of varying skew in the RWP 25 and RWP 2500 scenario

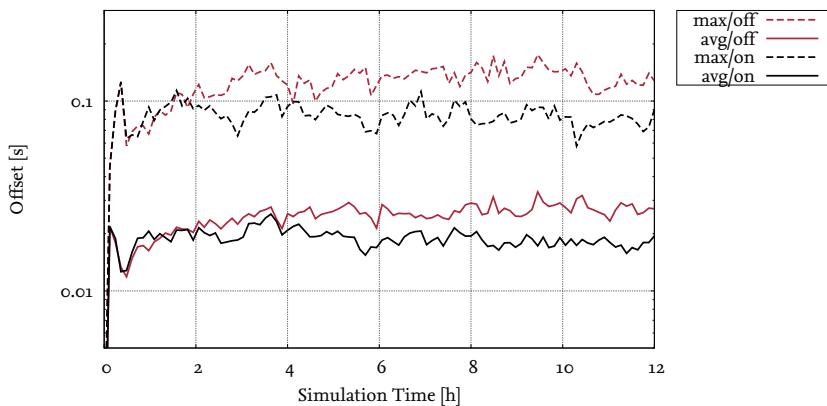


Figure 6.24: Benefit of the sigma adaptation in the Helsinki scenario

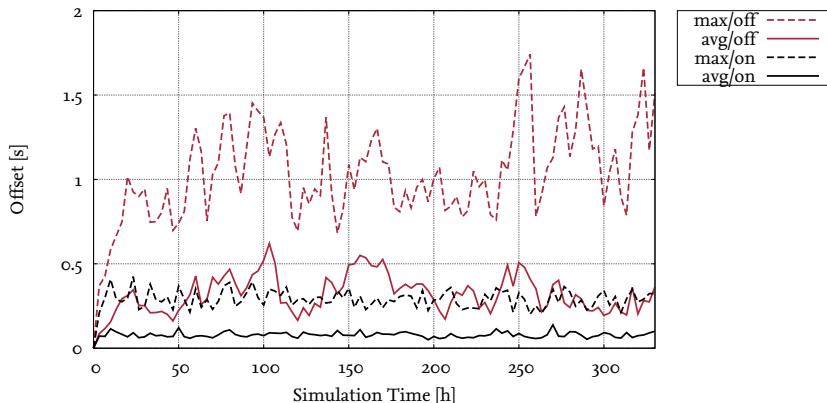


Figure 6.25: Benefit of the sigma adaptation in the RWP 25 scenario

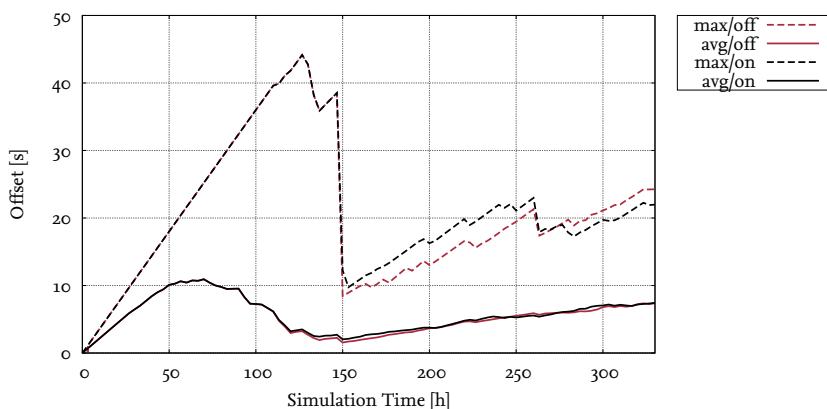


Figure 6.26: Benefit of the sigma adaptation in the RWP 2500 scenario

cation Server environment. The SSH-based control-connection between the distributed components has been replaced by a plain-text protocol over TCP. Virtualized nodes are controlled by *Slaves* using Node Control Protocol, thus, it is necessary to install a software component, called *Node Control Daemon* (ncd), on the nodes in order to allow them to communicate with their associated *Slave*. This software can allow/deny connections to other nodes, collect statistical data, and measure the clock offset to a given reference.

The most significant improvement is the way how sessions are configured, controlled, and observed. Instead of modifying text-files directly, a user can login into a web-based interface to create, edit, and run sessions. Figure 6.27 shows the section »Base« of the edit panel for a session. In the previous version of HYDRA, all these options were distributed through a lot of files. Additionally, it is possible to choose between different base images and a user can upload OpenWrt packages which will be installed during the node set-up.

The particular strength of HYDRA is the capability to collect and visualize statistical data. A user can configure an interval which defines how often these are collected. During the run, these data is visualized using interactive graphs as shown in Figure 6.28. Moreover, the simulated movement is visualized during the run in a Google Maps view.

Once the run is finished, the user can download all statistical data in a single file. This data may even contain custom data which is not visualized in the statistical view of the session. Further, replay-able movement and contact traces are generated which is especially useful to analyze randomly generated movement.

Scenarios

In order to evaluate the implementation of the synchronization algorithm in combination with IBR-DTN and a more realistic environment, we are going to copy the scenarios of Section 6.4.1 to HYDRA. Since the Helsinki scenario is quite dense and the result of the simulation did not reveal any issues, we chose to perform the runs with the more challenging random way-point scenarios. Both scenarios consist of 50 nodes moving with 0.5 m/s to 1.5 m/s according to a random way-point model. Each one has a transmission range of 250 m which is similar to Wi-Fi. The first scenario has a movement area of 5 km × 5 km and the second one an area of 50 km × 50 km. Both scenarios run for the duration of two weeks.

Each node is based on the *Attitude Adjustment* release of OpenWrt and is configured with 128 MB of RAM. Unnecessary services as `httpd` and `dnsmasq` are disabled during the set-up procedure. Further, the standard NTP daemon is disabled on all non-reference nodes.

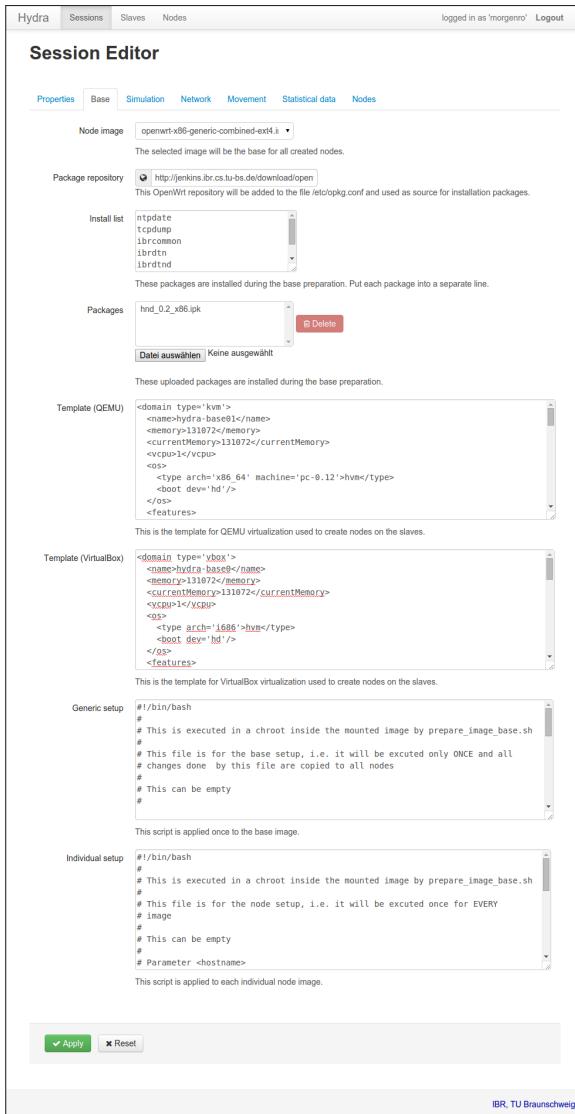


Figure 6.27: Session editor in HYDRA



Figure 6.28: Statistical reports in HYDRA

A limitation of this evaluation is that we cannot parametrize the clocks of the nodes to behave the same way as they did in the simulation. Instead, the clocks of the nodes are inherent inaccurate due to the virtualization. Without any further measurement, clocks of virtualized systems are running out of sync. Typically, those systems are synchronized using NTP or a specific guest software. In this test only the reference node is synchronized using the NTP protocol to a server connected to the same subnet.

Results

During the two weeks each run takes, the HYDRA system collects a record of statistical data for each node in an interval of 60 s. Those data includes in- and outgoing network traffic separated into ICMP, TCP, and UDP traffic. Further, the positions of the nodes, various counters for processed bundles, the number of connected neighbors, and the amount of bytes in the storage are collected. Especially of interest for this evaluation is the state of each individual clock. Thus, the statistical records also include the actual clock offset as well as the clock offset and rating determined by the time-synchronization algorithm component within IBR-DTN. In order to generate an expressive report using the gathered data, the records of all nodes are aggregated. Negative clock offsets are inverted to represent an absolute distance to the reference clock. This way the average of the values expresses how close the clocks are synchronized to an optimum. The following analysis considers the aggregated average as well the maximum offset of all nodes.

Figure 6.29 shows a comparison between the estimated clock offset of the RWP 25 scenario evaluated in HYDRA and the ONE. The courses of both runs are similar but not equal. One reason for this is that the nodes are not moved the same way. Only the parameters of the scenario and the movement model are equal. Further, the clock offsets determined during the run within HYDRA are higher at the beginning and then drop-down below 10 ms. The reason for this is an initial offset of the clocks due to a delay in the start-up routines of the virtualization system. As result each system has an unpredictable clock offset right after instantiation. This detail is more obvious when considering Figure 6.30 which compares the clock offset as they would be in an environment without any clock synchronization for the ONE simulation as well as the HYDRA emulation. Although this property is usually bad for simulations, it imitates reality where clocks of nodes are initially unsynchronized pretty good.

The drop-down within the first hours is the result of the synchronization which achieves an average clock offset of below 10 ms in the network for a short time. Later the clock offset raises in average. Although it converges against the simulated result, it stays below that because the clocks in the emulation are drifting

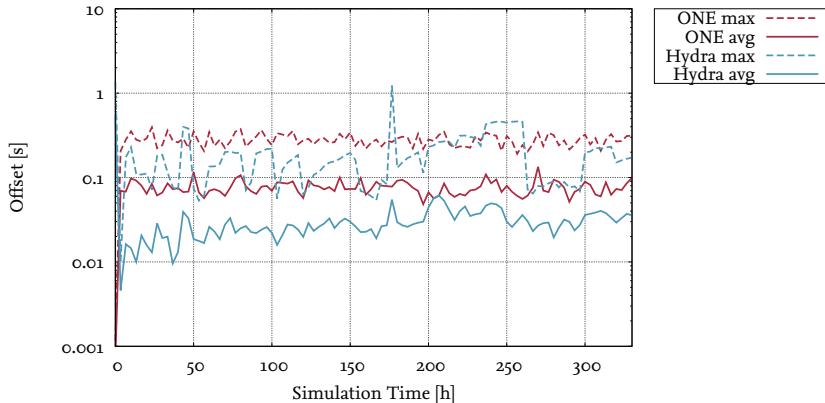


Figure 6.29: Comparison of the regulated clock offset estimated in the RWP 25 using ONE/Hydra

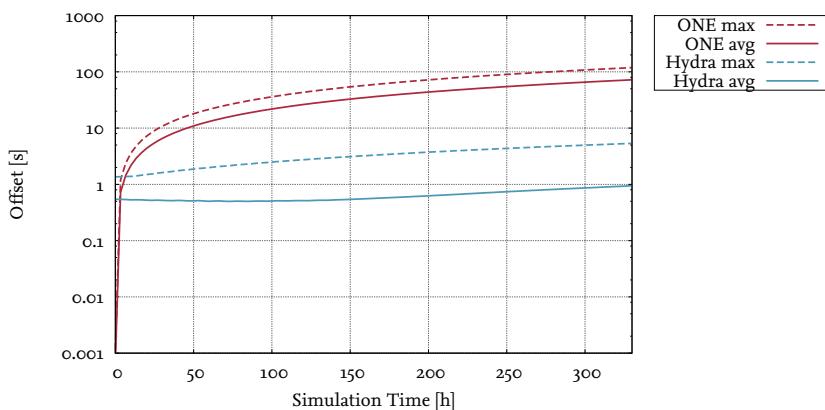


Figure 6.30: Comparison of the unregulated clock offset estimated in the RWP 25 using ONE/Hydra

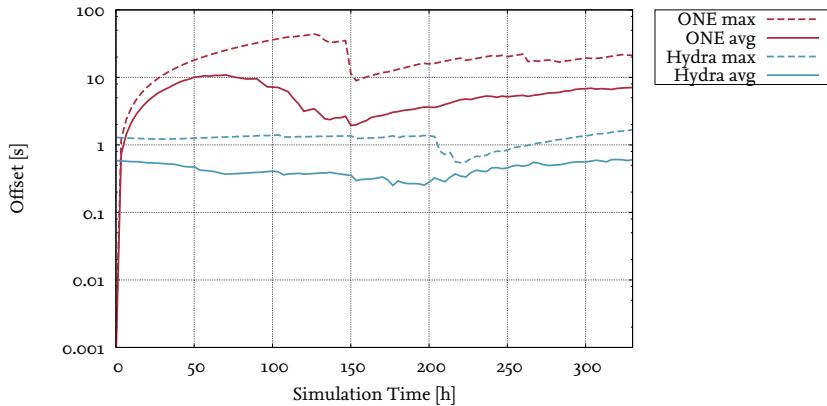


Figure 6.31: Comparison of the regulated clock offset estimated in the RWP 2500 using ONE/Hydra

slower than those in the simulation. That difference is clearly to see in Figure 6.30. At the end of the emulation, the synchronization approach achieved an average clock offset of 28 ms with a maximum of 72 ms. This is a significant improvement in comparison to the average of 959 ms and maximum of 5420 ms without clock synchronization.

Figure 6.31 shows the same comparison for the RWP 2500 scenario. A similarity of the graphs is also visible here. The differences are the result of different movement, initial clock offsets, and clock behavior as explained before. The run finishes with an average clock offset of 546 ms with a maximum of 1718 ms while the average offset would be 942 ms with a maximum of 3294 ms without synchronization. This is not as significant as the result of the RWP 25 scenario, but still an improvement. As explained before in Section 6.4.1, the achievable accuracy depends on the encounter frequency which is much lower in the RWP 2500 scenario compared to the RWP 25 scenario.

7 Conclusion

Delay- and Disruption-Tolerant Networking allows devices to communicate even if there is no continuous path to the destination by replacing the end-to-end semantics with a hop-by-hop store-carry-and-forward approach. Since several years, there exists a specification for a DTN architecture as well as a dedicated protocol designed to deal with the challenging characteristics of DTNs. Although the corresponding documents are in an experimental state, the approach is basically feasible. Thus, we considered existing experimental implementations and finally came to the conclusion that none of them fits all applications and scenarios we identified as use-cases for DTNs. Summarizing, they suffer from instability, missing features, low performance, and excessive resource consumption. Especially, those which strictly implement the BP are not yet ready for a carefree deployment. Their deficiency are not alone the result of insufficient engineering. Rather, the protocol itself has weaknesses which have to be solved by reasonable extensions and policies.

During the development of IBR-DTN and performing several experimentations, we figured out the remaining weaknesses of the *Delay-Tolerant Networking Architecture* [CBH⁺07], the *Bundle Protocol Specification* [SBo7], and extensions made to it. However, we did not declare the whole architecture as failed. Instead, we focused on useful extensions combined with reasonable policies to realize a lightweight and portable networking stack for terrestrial DTN communication which is sufficiently sophisticated to run on hardware for productive operations such as smartphones, embedded devices for vehicular networking, and standard laptops. For scenarios like wild-life monitoring, we also considered systems-on-chip solutions like the Arduino Yún as potential target.

7.1 Connecting the Dots

Based on the performance considerations and findings made in the early chapters, an architecture has been constructed which scales among the resources available on different hardware platforms. It consists of basic component groups named Core, Routing, Storage, Connectivity, and API. The Core provides the event-system which couples the components loosely together. Due to this approach, they can be replaced by specialized editions or even excluded from the build in

case a specific feature is not necessary for a given scenario. Even if all features are included in the distribution packages, it is still possible to enable or disable specific features during run-time by applying a custom configuration. Additionally, the event-system realizes a good utilization of available resources. By simply increasing the number of workers instances for event processing, it is possible to fully utilize multi-processor architectures.

The work-flows, defined in Section 3.6, refine the rough definitions made in the standard documents and explain the corresponding event-types. In addition to standard work-flows for bundle processing, we defined a procedure to exchange routing data between bundle nodes. Instead of utilizing an additional side channel, we encapsulate routing requests and responses into bundles as if the routing component itself would be an application. Further, we augmented related bundles with AEBs to solve issues with diverting clocks (Section 4.12.1), added SCHL blocks to limit the scope (Section 4.12.3), and compressed the payload of responses using the CEB extension (Section 3.9.3). The latter extension is a unique feature of IBR-DTN and has been designed as part of this work. In contrast to application-specific compression schemes, this approach has the advantage that a generic compression can be applied transparently to bundles and is available for all attached applications. Even if a receiving application is not aware of the compression extension, the received bundle gets transparently inflated and delivered to the application as long as the associated bundle node supports the extension.

The routing exchange is not the only work-flow which suffers from inaccurate or unsynchronized clocks. As discussed in Chapter 5, globally synchronized clocks are necessary to expire bundles correctly and supply unique identifiers to bundles. Additionally, scheduled routing approaches and energy-saving strategies benefit from synchronized clocks. Since there did not exist an approach to synchronize clocks of bundle nodes to one or more references, the Section 5.2 presented a time-synchronization approach which solves this most annoying dependency of the BP. Using that approach, it is possible to synchronize clocks, even if there is no continuous end-to-end path between all nodes. A fixed hierarchy, as required for existing protocols like NTP, is not needed. Instead, a nouveau rating metric is determined for each clock in the network and used to compare them. During the synchronization procedure, the better clock is used as reference for the clock with the lower rating.

7.2 Contribution

Large effort was spent to implement the architecture mentioned before. As result, the implementation is lightweight, portable, scalable, extensible, and inter-

operable with existing bundle node implementations. It provides interfaces to support further research and runs on embedded platforms, at least on everything supported by *OpenWrt*. By implementing an abstraction library (*ibrcommon*) to isolate platform-specific system routines from the programming language, it is possible to easily port the software to new system's conditions.

The routing interface supports reactive routing decisions based on the network topology. Beside direct delivery and static routing the bundle node also supports multi-hop routing approaches including flooding, epidemic, and *PRoPHET* routing. The routing extensions direct their decisions to forward a bundle to the convergence-layers. Those are based on a simple interface and adapt any kind of connectivity for bundle delivery to adjacent nodes. In case of DTNs, this might be even a flash drive which is plugged into and carried between several nodes. Beside the File Convergence-Layer which handles this case, *IBR-DTN* supports TCP, UDP, and IEEE 802.15.4 as underlay-network. The latter technology is integrated using the Datagram convergence-layer which is a generic component to reduce the effort to integrate further communication technologies based on datagrams.

To detect changes in the network topology and therefore opportunities to transfer bundles to adjacent bundle nodes, *IBR-DTN* includes a component to discover neighbors. The implemented approach is based on the IPND protocol introduced in Section 3.8. We expanded that approach to fit all kinds of convergence-layers and broadcast the same beacon format over broadcast-capable non-IP underlay-networks.

With *IBR-DTN*, we were the first who implemented the complete feature-set of the *Bundle Security Protocol* [SFWL11]. Until we started with the development on that, existing implementations had only realized the authentication part. Today, at least *DTN2* and *ION* implements the complete specified future-set. Equal to *IBR-DTN*, they provide features like integrity, confidentiality, authenticity and non-repudiation for all transmissions. In addition to those basic capabilities, we considered the protocol in regard to uncovered attacks and identified black-hole attacks as possible vulnerability. The TLS extensions for the TCP-CL introduced in Section 3.9.5 prevents those attacks by authenticating connected peers. As explained, this approach needs special adaptation to the convergence-layer and is not feasible if the transmission delay between two nodes is too high, because the required handshake at the beginning of a connection would time-out. However, this approach supports terrestrial scenarios and connection-oriented convergence-layers quite well.

Since common issues are now solved, we additionally addressed reasonable deployment strategies. Even if DTNs are designed to replace classical networking approaches, existing structures can be used to interconnect non-neighbored bun-

dle nodes. The naïve approach would be to maintain a list of nodes reachable over the Internet. The list can be used to establish static connections between all bundle nodes to gain a fully meshed network. Since that approach would not scale well, we presented the DTN-DHT concept in Section 3.9.1. Using that scalable approach, a bundle node can resolve EIDs to IP addresses using a peer-to-peer DHT. Indeed, a boot-strapping approach which utilizes the *Domain Name System* of the Internet is required. But in case of isolation, a bundle node can discover peers to connect to using the *DiscoveryAgent*.

Another issue we addressed was the recently upcoming demand for DTN software on unmodified mobile devices a.k.a. smart-phones. With *Bytewalla* and *SCAMPI Router*, two implementations for the Android™ platform are described in Section 2.4. While *Bytewalla* is a re-write of the C++ code of DTN2 to Java, the *SCAMPI Router* is a complete re-engineered implementation. Since both projects do not introduce any advantage compared to the existing implementations (except of running on Android™), it is not clear why the authors did not port existing software instead. However, we claimed IBR-DTN as a portable and feature-rich software which should generally run on almost any platform. Thus, we explored the possibilities to bring the existing code of IBR-DTN directly to Android™. Section 6.1.3 has explained the necessary steps to realize this proposal successfully. By porting the C++ code base to another platform, we proved that the implementation is flexible enough to be adapted to different systems with relative low effort. As result, we gained an implementation for the Android™ platform which is usually dominated by Java applications.

Thorough testing is essential during development of every software. In Chapter 6, we described our concept for continuous integration combined with unit-tests and automatic performance measurements. Beside those basic testing principles, we needed to test the software in large constructed scenarios with multiple DTN nodes and emulated movement within a given area. While the ONE simulator provides a quite sophisticated approach to simulate DTN related algorithms, it is much more complicated to perform large-scale testing of real software implementations because that usually implies a manual set-up. The HYDRA emulation framework presented in Section 6.4.2 covers those cases and allows a user to set-up large-scale DTN scenarios with minimal effort. The system spawns a virtual host with a dedicated OpenWrt system for each node in the scenario. The software, installed on each instance, is defined by a configuration associated with the given scenario. HYDRA monitors each node and collects statistical data in a defined interval. Collected data is stored within a database and visible in the live-statistics as well as available for off-line analysis. Additionally, the user can attach a console to each instance to inspect a node or perform actions on it, while the emulation is

running. In our cases, the software was used to compare the behavior of the clock synchronization algorithm, but since this system is open-sourced, as IBR-DTN too, it is a useful tool for future research on routing algorithms or other software mentioned to work in a DTN environment.

7.3 Reasonable Limitations

With IBR-DTN, we implemented the complete feature-set of the BP based on the *Delay-Tolerant Networking Architecture*. Both specifications are very liberal in regard to constraints and limitations. We tried to follow that intention and build the software as open and flexible as possible. However, we identified potential issues which could be avoided with carefully selected policies and restrictions on the implemented protocols. Section 4.13 discussed existing issues and reasonable measures to mitigate them.

One of the first issues we have noticed is related to the fragmentation process. The specifications define reactive and pro-active fragmentation. Both approaches would work without any issue in conjunction with single-copy-routing approaches. But as soon as a routing approach utilizes multiple paths, the exact handling of fragments created on intermediate nodes gets a little fuzzy. For that reason, we limited the cases when fragmentation is permitted. Pro-active fragmentation is only applied at the source and the reassembly of fragments is only performed at the destination of the bundle. Thus, the complete bundle is never injected into the DTN. In case of reactive fragmentation, only the receiving peer is allowed to generate a fragment from the received data. The sending peer remembers how many data has been forwarded to the peer with the interrupted connection, but will forward only complete bundles to other peers. As soon there is a successive encounter with the same peer, the transmission can be resumed by generating a virtual fragment. Any other case, where the bundle is fragmented and reassembled on the path, is not part of IBR-DTN due to missing specifications.

The time-synchronization approach presented in Section 5.2 covers scenarios where nodes in a DTN require globally synchronized clocks. Although our approach is a large improvement, it has its limitations. At least one reference must be defined to synchronize clocks in a DTN. If there are multiple references, the approach requires that all of them are synchronized with another sophisticated approach. Further, the accuracy of the approach depends on the inter-contact times. If a part of the network is isolated from all reference nodes for a long time, the clock rating of all nodes will tend towards zero and synchronization will barely happen.

7.4 Future Work

As nearly always, there is potential for some future work. More sophisticated routing algorithms are necessary to forward bundles more efficiently. At the moment, PRoPHET is the most sophisticated routing protocol in IBR-DTN and covers many scenarios. But it does not scale very well and tends to flood large parts of the network with copies. Especially in case of scheduled contacts, a specialized routing approach can reduce the delivery delay and increase network capacity. Another routing related gap to fill is a good routing concept for bundles addressed to non-singleton endpoints. These bundles are mentioned to be delivered to multiple destinations. But it is not defined which endpoints are part of the group. In the routing implementation of IBR-DTN, we bypass the issue by flooding those bundles to all nodes in the network. Based on the application registrations, each individual bundle node can decide locally if it should deliver non-singleton bundles or not.

Further potential for future research exists for the time-synchronization. As mentioned in the previous section, the approach presented here has its limitations if the network is partitioned for a long time. In that case, the approach should be extended to allow the nodes to synchronize their clocks within the isolated networking group similar to the DCS algorithm [CS10].

The port of IBR-DTN to the Android™ platform has also revealed necessary future work and topics. In the current version of the implementation, most of the state is not stored persistently. Neighbor information and the data related to application sessions do not survive a restart of the bundle node. Since Android™-based devices are rebooted sometime and the system itself might stop services and later restart them if the device runs out of memory, it is essential to store important data persistently. The missing persistent state is not only an issue on Android™. Nodes of vehicular networking are usually off while the vehicle is not in use. Therefore, a bundle node running on such a node would lose its volatile state every time the vehicle is parked.

Another issue on Android™ is the power-consumption introduced by the neighbor discovery process which sends out a beacon every second. This procedure keeps the device almost constantly active and requires a lot of energy. On embedded devices for vehicular networking, this energy consumption is negligible. But on mobile nodes, the continuous scanning for neighbors is a major disadvantage of such a system. Intelligent scanning patterns and advanced power saving strategies are required to make DTN software more energy efficient.

Bibliography

- [And12] Erik Andersen. A C library for embedded Linux, 2012. Available online <http://www.uclibc.org/about.html> accessed on 12/16/2014.
- [Arc14] Jake Archibald. The ServiceWorker: The network layer is yours to own, June 2014. Available online <http://www.w3.org/TR/service-workers/> accessed on 12/13/2014.
- [Ard14] Arduino. Arduino Yún – Product Page, 2014. Available online <http://arduino.cc/en/Main/ArduinoBoardYun> accessed on 12/16/2014.
- [BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum Disclosure Proofs of Knowledge. In *Journal of Computer and System Sciences*, volume 37, pages 156–189. Academic Press, Inc., Orlando, FL, USA, October 1988.
- [BFB10] Daniel W. Brown, Stephen Farrell, and Scott Burleigh. DTN Bundle Age Block for Expiration without UTC. Internet-Draft (work in progress), DTN Research Group, October 2010.
- [Bla12] Marc Blanchet. Postellation: An Enhanced Delay-Tolerant Network (DTN) Implementation with Video Streaming and Automated Network Attachment. In *SpaceOps 2012*, Stockholm, Sweden, 2012. American Institute of Aeronautics and Astronautics, Inc.
- [BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), January 2005. Updated by RFCs 6874, 7320.
- [BM07] X. Boyen and L. Martin. Identity-Based Cryptography Standard (IBCS) #1: Supersingular Curve Implementations of the BF and BB1 Cryptosystems. RFC 5091 (Informational), December 2007.
- [Bur] Scott Burleigh. Time synchronization within DTNs. E-mail to the IRTF dtn-interest mailing list, 13. July 2009. Available online <http://www.ietf.org/mail-archive/web/dtn-interest/current/msg01006.html> accessed on 10/27/2014.

- [Bur09] Scott Burleigh. Interplanetary Overlay Network (ION) - Design and Operation. Technical report, Jet Propulsion Laboratory, California Institute of Technology, 2009.
- [Bur11] S. Burleigh. Compressed Bundle Header Encoding (CBHE). RFC 6260 (Experimental), May 2011.
- [Bur14a] Scott Burleigh. Bundle Protocol Extended Class Of Service (ECOS). Internet-Draft (Experimental), Jet Propulsion Laboratory, California Institute of Technology, January 2014. Available online <http://tools.ietf.org/html/draft-irtf-dtnrg-ecos-05> accessed on 10/27/2014.
- [Bur14b] Scott Burleigh. NASA's Disruption Tolerant Networking Challenge Series - Astronaut Email, 2014. Available online <http://www.topcoder.com/dtn/astronaut-email/> accessed on 10/27/2014.
- [CBH⁺07] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay-Tolerant Networking Architecture. RFC 4838 (Informational), April 2007.
- [CFSD90] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.
- [Con14] SQLite Consortium. SQLite, October 2014. Available online <https://www.sqlite.org/> accessed on 10/27/2014.
- [CS10] Bong Jun Choi and Xuemin Shen. Distributed Clock Synchronization in Delay Tolerant Networks. In *IEEE International Conference on Communications*, IEEE ICC '10, pages 1–6, New York, NY, USA, May 2010. IEEE.
- [Cur14] Richard Curnow. Chrony Home, September 2014. Available online <http://chrony.tuxfamily.org/> accessed on 12/16/2014.
- [DAR14] Beman Dawes, David Abrahams, and Rene Rivera. boost C++ libraries, November 2014. Available online <http://www.boost.org/> accessed on 12/16/2014.
- [Dav09] Elwyn Davies. D2.1: Preliminary Investigations, Framework and Architecture. Deliverable, Networking for Communications Challenged Communities, Luleå, Sweden, June 2009.

- [Dav10] Elwyn Davies. D2.2: Functional Specification for DTN Infrastructure Software. Deliverable, Networking for Communications Challenged Communities, Luleå, Sweden, 2010.
- [Dav14] Dave Beazley. What is SWIG?, May 2014. Available online <http://www.swig.org/exec.html> accessed on 12/16/2014.
- [DBF⁺04] Michael Demmer, Eric Brewer, Kevin Fall, Sushant Jain, Melissa Ho, and Robin Patra. Implementing Delay Tolerant Networking. Technical Report IRB-TR-04-020, Intel Research, December 2004.
- [Deu96a] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.
- [Deu96b] P. Deutsch. GZIP file format specification version 4.3. RFC 1952 (Informational), May 1996.
- [DG96] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996.
- [DKo8] Jon Dugan and Mitch Kutzko. Iperf, March 2008. Available online <http://iperf.sourceforge.net/> accessed on 12/16/2014.
- [DOP14] M. Demmer, J. Ott, and S. Perreault. Delay-Tolerant Networking TCP Convergence-Layer Protocol. RFC 7242 (Experimental), June 2014.
- [DPW10] Michael Doering, Tobias Pögel, and Lars Wolf. DTN routing in urban public transport systems. In *Proceedings of the 5th ACM workshop on Challenged networks*, CHANTS '10, pages 55–62, New York, NY, USA, 2010. ACM.
- [DRW11] Michael Doering, Stephan Rottmann, and Lars Wolf. Design and Implementation of a Low-Power Energy Management Module with Emergency Reserve for Solar Powered DTN-Nodes. In *Proceedings of the 3rd Extreme Conference of Communication (ExtremeCom 2011), Manaus, Brazil*, 9 2011.
- [DW12] Michael Doering and Lars Wolf. Work-in-Progress: Evaluation of generic Bundle Transmission Scheduling Strategies in Vehicular Disruption Tolerant Networks. In *Proceedings of the 4th Extreme Conference of Communication (ExtremeCom 2012), Zurich, Switzerland*, 3 2012.

- [EAGB12] Daniel Ellard, Richard Altmann, Alex Gladd, and Daniel Brown. DTN IP Neighbor Discovery (IPND). IETF Draft, Internet Research Task Force, 2012. Available online <http://tools.ietf.org/id/draft-irtf-dtnrg-ipnd-02> accessed on 10/27/2014.
- [ED11] W. Eddy and E. Davies. Using Self-Delimiting Numeric Values in Protocols. RFC 6256 (Informational), May 2011.
- [EKKOo8] Frans Ekman, Ari Keränen, Jouni Karvo, and Jörg Ott. Working day movement model. In *MobilityModels '08: Proceeding of the 1st ACM SIGMOBILE workshop on Mobility models*, pages 33–40, New York, NY, USA, 2008. ACM.
- [Elso03] Jeremy Eric Elson. *Time Synchronization in Wireless Sensor Networks*. PhD thesis, University of California, Los Angeles, CA, USA, 2003.
- [Falo03] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 27–34, New York, NY, USA, 2003.
- [Falo10] Kevin Fall. DTN Scope Control using Hop Limits (SCHL). IETF Draft, Intel Labs, Berkeley, September 2010. Available online <http://tools.ietf.org/html/draft-fall-dtnrg-schl-00> accessed on 10/27/2014.
- [FCB14] Stephen Farrell, V. Cerf, and M. Božnar. Networking for Communications Challenged Communities, October 2014. Available online <http://www.n4c.eu/> accessed on 10/27/2014.
- [Fey13] Alex Feyerke. Designing Offline-First Web Apps, December 2013. Available online <http://alistapart.com/article/offline-first> accessed on 12/13/2014.
- [FKK10] Nikolaos M. Freris, Hemant Kowshik, and P. R. Kumar. Fundamentals of Large Sensor Networks: Connectivity, Capacity, Clocks and Computation. In *Proceedings of the IEEE*, volume 98, pages 1828–1846, New York, NY, USA, November 2010.
- [FLKL10] Stephen Farrell, Aidan Lynch, Dirk Kutscher, and Anders Lindgren. Bundle Protocol Query Extension Block. Internet-Draft (Experimental), Internet Research Task Force, November 2010. Available online

- <http://tools.ietf.org/html/draft-farrell-dtnrg-bpq-00> accessed on 10/27/2014.
- [FMO09] Stephen Farrell, Alex Mc Mahon, and Joerg Ott. Handling Issues with Real Time in the Bundle Protocol. Internet-Draft (work in progress), DTN Research Group, November 2009. Available online <http://tools.ietf.org/html/draft-farrell-dtnrg-alt-time-00> accessed on 10/27/2014.
- [GCB⁺02] Jim Gray, Wyman Chong, Tom Barclay, Alexander S. Szalay, and Jan Vandenberg. TeraScale SneakerNet: Using Inexpensive Disks for Backup, Archiving, and Data Exchange. Technical report, Microsoft Research, May 2002.
- [GGV12] B. Grašič, S. Grasic, and S. Vrbinc. Prophet DTN implementation, June 2012. Available online <http://sourceforge.net/projects/pluti/> accessed on 10/27/2014.
- [Hos14] Qt Project Hosting. Qt Project, 2014. Available online <http://qt-project.org/> accessed on 12/16/2014.
- [HPJ02] Yih Chun Hu, Adrian Perrig, and David B. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 12–23, New York, NY, USA, 2002. ACM.
- [HT11] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), September 2011.
- [ian14] Bundle Protocol. Protocol Registry Document, Internet Assigned Number Authority, June 2014. Available online <https://www.iana.org/assignments/bundle/bundle.xhtml> accessed on 12/11/2014.
- [Int06] Intel Corporation. Intel Mote 2 – Engineering Platform Data Sheet, 2006. Available online http://wsn.cse.wustl.edu/images/c/cb/Imote2-ds-rev2_2.pdf accessed on 12/16/2014.
- [Ish] Joseph Ishac. DTN Time Sync Issues. E-mail to the IRTF dtn-interest mailing list, 31. March 2008, and subsequent discussion. Available online <http://www.ietf.org/mail-archive/web/dtn-interest/current/msg01913.html> accessed on 10/27/2014.

- [JOW⁺02] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shizuan Peh, and Daniel Rubenstein. Energy-efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 96–107, New York, NY, USA, 2002. ACM.
- [KJO14] H. Kruse, S. Jero, and S. Ostermann. Datagram Convergence Layers for the Delay- and Disruption-Tolerant Networking (DTN) Bundle Protocol and Licklider Transmission Protocol (LTP). RFC 7122 (Experimental), March 2014.
- [Kly14] Graham Klyne. Uniform Resource Identifier (URI) Schemes. Protocol Registry Document, Internet Assigned Number Authority, October 2014. Available online <https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml> accessed on 12/11/2014.
- [KN93] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). RFC 1510 (Historic), September 1993. Obsoleted by RFCs 4120, 6649.
- [KOK09] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, SIMUTools '09, New York, NY, USA, 2009. ICST.
- [Kor14] Peter Korsgaard. Buildroot – Making Embedded Linux Easy, December 2014. Available online <https://www.openwrt.org/> accessed on 12/17/2014.
- [KZ11] A. Kathirvel and H. Zhuang. Integration of bytewalla 5, October 2011. Available online <https://code.google.com/p/bytewallal5/> accessed on 10/27/2014.
- [LDDG12] A. Lindgren, A. Doria, E. Davies, and S. Grasic. Probabilistic Routing Protocol for Intermittently Connected Networks. RFC 6693 (Experimental), August 2012.
- [LDPL06] Sven Lahde, Michael Doering, Wolf-Bastian Pöttner, and Gerrit Lammer. Mobile and Distributed Measurement of Air Pollution in Metropolitan Areas Using Car2X Techniques. In *Proceedings of 3rd*

- Symposium on Informationssysteme für mobile Anwendungen*, IMA, Braunschweig, Germany, October 2006.
- [LDS03] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic routing in intermittently connected networks. In *SIGMOBILE Mobile Computing and Communications Review*, volume 7, pages 19–20. ACM, New York, NY, USA, July 2003.
- [LWSW11] Jó Ágila Bitsch Link, Christoph Wollgarten, Stefan Schupp, and Klaus Wehrle. Perfect difference sets for neighbor discovery: Energy efficient and fair. In *Proceedings of the 3rd Extreme Conference on Communication: The Amazon Expedition*, ExtremeCom ’11, pages 5:1–5:6, New York, NY, USA, 2011. ACM.
- [Mil85] D.L. Mills. Network Time Protocol (NTP). RFC 958, September 1985. Obsoleted by RFCs 1059, 1119, 1305.
- [MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), September 2007. Updated by RFCs 6282, 6775.
- [Moo14] B. Moon. JDTN, January 2014. Available online <http://sourceforge.net/projects/jdtn/> accessed on 10/27/2014.
- [MPW11] Johannes Morgenroth, Tobias Pögel, and Lars Wolf. Live-streaming in delay tolerant networks. In *Proceedings of the 6th ACM workshop on Challenged networks*, CHANTS ’11, pages 67–68, New York, NY, USA, 2011. ACM.
- [MSW10] Johannes Morgenroth, Sebastian Schildt, and Lars Wolf. HYDRA: virtualized distributed testbed for DTN simulations. In *WiNTECH ’10: Proceedings of the fifth ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 71–78, New York, NY, USA, 2010. ACM.
- [MW12] Johannes Morgenroth and Lars Wolf. Time-reference distribution in delay tolerant networks. In *Proceedings of the seventh ACM international workshop on Challenged networks*, CHANTS ’12, pages 1–8, New York, NY, USA, 2012. ACM.
- [Nag84] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

- [Ope14] OpenWrt Project. OpenWrt – Wireless Freedom, October 2014. Available online <https://www.openwrt.org/> accessed on 12/17/2014.
- [Pet12] Colin Peters. Minimalist GNU for Windows, May 2012. Available online <http://www.mingw.org/> accessed on 12/16/2014.
- [PFHo4] Alex (Sandy) Pentland, Richard Fletcher, and Amir Hasson. DakNet: Rethinking Connectivity in Developing Nations. In *Computer*, volume 37, pages 78–83. IEEE Computer Society Press, Los Alamitos, CA, USA, January 2004.
- [PH83] J. Postel and K. Harrenstien. Time Protocol. RFC 868 (INTERNET STANDARD), May 1983.
- [PKO⁺12] Mikko Pitkänen, Teemu Kärkkäinen, Jörg Ott, Marco Conti, Andrea Passarella, Silvia Giordano, Daniele Puccinelli, Franck Legende, Sacha Trifunovic, Karin Hummel, Martin May, Nidhi Hegde, and Thrasyvoulos Spyropoulos. SCAMPI: Service Platform for Social Aware Mobile and Pervasive Computing. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC ’12, pages 7–12, New York, NY, USA, 2012. ACM.
- [PMSW11a] Wolf-Bastian Pöttner, Johannes Morgenroth, Sebastian Schildt, and Lars Wolf. An Empirical Performance Comparison of DTN Bundle Protocol Implementations. Informatikbericht 2011-08, Technische Universität Braunschweig, 9 2011.
- [PMSW11b] Wolf-Bastian Pöttner, Johannes Morgenroth, Sebastian Schildt, and Lars Wolf. Performance Comparison of DTN Bundle Protocol Implementations. In *Proceedings of the 6th ACM Workshop on Challenged Networks*, CHANTS ’11, pages 61–64, New York, NY, USA, 2011. ACM.
- [Pos81a] J. Postel. Internet Control Message Protocol. RFC 792 (INTERNET STANDARD), September 1981. Updated by RFCs 950, 4884, 6633, 6918.
- [Pos81b] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.
- [Rö01] Kay Römer. Time synchronization in ad hoc networks. In *MobiHoc ’01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182, New York, NY, USA, 2001. ACM.

- [RFdAGo08] Juan-Carlos Ruiz, Jesús Friginal, David de Andrés, and Pedro Gil. Black Hole Attack Injection in Ad hoc Networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN2008, pages G34–G35, Anchorage, Alaska, June 2008.
- [RS14] Alex Russell and Jungkee Song. Service Workers, November 2014. Available online <http://www.w3.org/TR/service-workers/> accessed on 12/13/2014.
- [SBo07] K. Scott and S. Burleigh. Bundle Protocol Specification. RFC 5050 (Experimental), November 2007.
- [SBK05] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. Clock Synchronization for Wireless Sensor Networks: A Survey. In *Ad Hoc Networks*, volume 3, pages 281–323. Elsevier, 2005.
- [SDS09] Susan Flynn Symington, Robert C. Durst, and Keith L. Scott. Delay-Tolerant Networking Bundle-in-Bundle Encapsulation. IETF Draft, Internet Research Task Force, August 2009. Available online <http://tools.ietf.org/html/draft-irtf-dtnrg-bundle-encapsulation-06> accessed on 10/27/2014.
- [Sew14] Julian Seward. bzip2 and libbzip2, September 2014. Available online <http://www.bzip.org/> accessed on 12/16/2014.
- [SFWL11] S. Symington, S. Farrell, H. Weiss, and P. Lovell. Bundle Security Protocol Specification. RFC 6257 (Experimental), May 2011.
- [SLM⁺12] Sebastian Schildt, Till Lorentzen, Johannes Morgenroth, Wolf-Bastian Pöttner, and Lars Wolf. Free-riding the bittorrent dht to improve dtn connectivity. In *Proceedings of the seventh ACM international workshop on Challenged networks*, CHANTS '12, pages 9–16, New York, NY, USA, 2012. ACM.
- [Sym11a] S. Symington. Delay-Tolerant Networking Metadata Extension Block. RFC 6258 (Experimental), May 2011.
- [Sym11b] S. Symington. Delay-Tolerant Networking Previous-Hop Insertion Block. RFC 6259 (Experimental), May 2011.
- [Tano01] Andrew S. Tanenbaum. *Modern Operating Systems*, volume 2. Prentice Hall, Upper Saddle River, NJ, USA, 2001.

- [The] The Open Group. POSIX.1-2008. Available online <http://pubs.opengroup.org/onlinepubs/9699919799/> accessed on 10/27/2014.
- [The14] The MacPorts Project. The MacPorts Project Official Homepage, 2014. Available online <https://www.macports.org/> accessed on 12/16/2014.
- [Tra14] Transcend Information, Inc. Wi-Fi SD Karte – Product Page, 2014. Available online <http://www.bzip.org/> accessed on 12/16/2014.
- [TW10] A. Tanenbaum and D. Wetherall. *Computer Networks*. Prentice Hall Press, Upper Saddle River, NJ, USA, 5th edition, 2010.
- [VBoo] Amin Vahdat and David Becker. Epidemic Routing for Partially Connected Ad Hoc Networks. Technical report, Department of Computer Science, Duke University, Durham, NC, USA, April 2000.
- [vZBPW12] Georg von Zengen, Felix Büsching, Wolf-Bastian Pöttner, and Lars Wolf. An Overview of μ DTN: Unifying DTNs and WSNs. In *Proceedings of the 11th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN)*, Darmstadt, Germany, September 2012.
- [Wei66] Joseph Weizenbaum. ELIZA – A Computer Program for the Study of Natural Language Communication Between Man and Machine. In *Communications of the ACM*, volume 9, pages 36–45. ACM, New York, NY, USA, January 1966.
- [Wi-14] Wi-Fi Alliance. Wi-Fi Direct, 2014. Available online <http://www.wi-fi.org/discover-wi-fi/wi-fi-direct> accessed on 10/27/2014.
- [WTSBo9] Shin-Ywan (Cindy) Wang, J. Leigh Torgerson, Joshua Schoolcraft, and Yan Brenman. The Deep Impact Network Experiment Operations Center Monitor and Control System. In *IEEE International Conference on Space Mission Challenges for Information Technology*, pages 34–40, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [YAMA11] Rerngvit Yanggratoke, Abdullah Azfar, María José Peroza Marval, and Sharjeel Ahmed. Delay Tolerant Network on Android Phones: Implementation Issues and Performance Measurements. In *Journal of Communications*, volume 6, pages 477–484. Academy Publisher, 2011.

- [YCo08] Qing Ye and Liang Cheng. DTP: Double-Pairwise Time Protocol for Disruption Tolerant Networks. In *The 28th International Conference on Distributed Computing Systems, ICDCS '08*, pages 345–352, New York, NY, USA, June 2008. IEEE.
- [ZDG⁺14] Haojin Zhu, Suguo Du, Zhaoyu Gao, Mianxiong Dong, and Zhenfu Cao. A Probabilistic Misbehavior Detection Scheme toward Efficient Trust Establishment in Delay-Tolerant Networks. In *IEEE Transactions on Parallel and Distributed Systems*, volume 25, pages 22–32. IEEE, January 2014.
- [Zig14] ZigBee Alliance. What is ZigBee?, 2014. Available online <http://zigbee.org/what-is-zigbee/> accessed on 12/16/2014.

