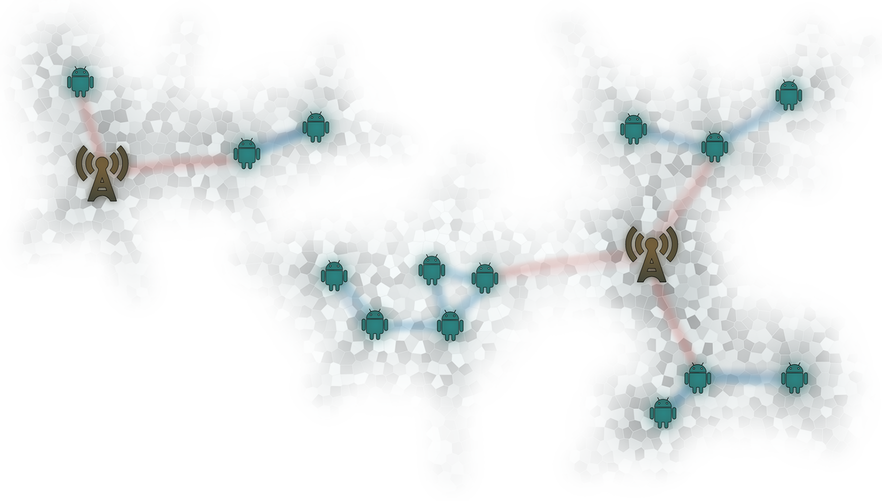
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Jan Haller

Bringing Network Access to Disconnected Regions using DTN Throwboxes



Master Thesis MA-2014-04
January to July 2014

Tutor: Abdullah Alhussainy
Co-Tutor: Dr. Karin Anna Hummel
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

In this Master Thesis, a software framework for delay-tolerant networking (DTN) on Android and Desktop platforms is developed. Based on WiFi and Bluetooth technology, devices running the framework connect autonomously and exchange data over multiple hops. Employing the *store-carry-forward* paradigm allows contacts to be intermittent, in the way that a device continues to forward packets as soon as it encounters new nodes. Our implementation widely complies to the Bundle Protocol as specified in RFC 5050 and is thus able to interoperate with other BP-enabled devices.

A focus of the work is laid on *throwboxes* – stationary access points that support delay-tolerant networks by providing additional contact opportunities – as well as probabilistic routing algorithms. The proposed routing protocol extends PRoPHET, an opportunistic routing protocol, by an extended calculation of a node's popularity. A new parameter is introduced to prioritize throwboxes over ordinary nodes with respect to forwarding.

An experiment was conducted to test the DTN framework under real-world conditions. A C++ application to simulate a DTN network was written during the thesis, and used to evaluate a community scenario based on random node movement as well as a mobility model based on GPS traces.

We could show that the DTN framework is operational in a fully autonomous setting. A core insight of the experiment and the simulations was that nodes prefer direct communication via WiFi, while Bluetooth is of limited use for data exchange in delay-tolerant networking. The addition of WiFi throwboxes in strategic locations resulted in considerable improvement of delivery ratio and end-to-end latency. The routing protocol developed during this work performed better than epidemic routing and a protocol that uses a constrained number of duplicates, but does not consider node popularities.

Acknowledgments

Looking back on the past six months, I would like to thank Abdullah Alhussainy and Karin Anna Hummel for the input they have given me during my Master Thesis. With their experience on the subject of DTNs and research in general, Abdullah and Karin have contributed in a very helpful and nice way to my work. Whenever I needed new directions or requested feedback, they were on the spot and allowed me to consider and question different approaches.

I would like to thank Professor Bernhard Plattner for giving me the opportunity to write a Master Thesis on a very interesting topic in the Communication Systems Group at ETH. I would also like to thank the CSG members who have participated in my experiment and made it possible to test my DTN framework under real-world conditions.

I truly appreciate the work of Jim Miner, the developer of the JDTN library. By providing a very well-structured and well-documented software, he has allowed me to delve into the subject very quickly, and start directly with the fascinating parts.

Contents

Chapter 1

Introduction

In the year 2014, connectivity and instant communication with other people has become a ubiquitous property of many societies. Not so long ago, the upcoming of WiFi hotspots in public places has allowed commuters to check their e-mails while waiting for the train, which was a considerable step forward – not only phone calls, but also Internet access had become mobile. The efforts of cell phone providers to establish 2G, 3G and 4G networks have led to a constant availability of Internet connectivity, even while moving at high velocities such as in trains. This has fundamentally changed the rate at which we are able to exchange information with others [1].

At the same time, many regions on Earth are still disconnected from the Internet. Sometimes, stationary Internet access is rare and expensive, and even more so is mobile access. The progress in smart phone hardware development has caused the development of low-cost smart phones with wireless capabilities that would theoretically allow for all sorts of communication. However, infrastructure in developing countries does often not keep pace with these advances, making connectivity a rare good.

Delay- and disruption-tolerant networks (DTNs) offer interesting new possibilities in such scenarios. The underlying assumptions in DTNs are completely different from classical network architectures such as the Internet. As implied by the name, DTNs are designed to account for propagation delays and disruptions in the path a message takes. Most importantly, there is no guarantee of a connected end-to-end path. Instead, messages have to travel over multiple hops, possibly at different times with intermediate interruptions. Prevalent in delay-tolerant networking is the *store-carry-forward* paradigm, meaning that nodes store messages and carry them along until they encounter other nodes, to which the messages can eventually be forwarded. Delay-tolerant networks are not limited to wireless communication; they can be interconnected with local area networks or the Internet where such an option exists. Even technologies such as USB keys for data carrying are possible.

1.1 Focus of this Work

This Master Thesis entails the development and implementation of a delay tolerant network for mobile devices in the form of smart phones and stationary access points in the form of standard laptops. Building on top of Android, Windows and Linux operating systems, a Java framework for delay-tolerant networking is developed. This framework allows smart phones to discover neighbor devices, to establish a connection to them and to exchange data. Disruption and delays in the connection require the store-carry-forward approach, making sure that messages that cannot be immediately forwarded are stored until a new contact opportunity emerges. The goal of the Java framework is to provide a DTN service to the user which runs automatically in the background and meets autonomous decisions. It is then possible to write specific applications such as text or photo sharing on top of the framework, and thus to abstract from the networking part and concentrate on application logic.

A focus of the work is the incorporation of *throwboxes*, which represent stationary DTN nodes that act as access points in the network. The idea underlying throwboxes is to improve the number of contact opportunities, as they allow the exchange of messages between the nodes that pass a location at different times. Because of higher throughput, range, storage and more stable power supply, throwboxes can typically process more load in the network, which makes them desirable nodes to forward packets to.

Delay-tolerant networking introduces many challenges into the design of networks, one of which is routing. As every DTN node has only a very limited view of the whole network and mostly does not have a connected path to the destination of a message, routing protocols are often implemented in a probabilistic way. The difficulty lies in predicting future contact opportunities, in order to achieve high probability of message delivery. There are several trade-offs to consider; while spreading messages in larger parts of the network generally increases the probability of arrival, it comes at a cost of network resources such as available bandwidth or storage capacity.

1.2 Thesis Organization

Chapter 2 gives an overview of related work in the area of delay-tolerant networks and discusses research in the last decades. Insights regarding a DTN architecture in general and the standardization of the bundle protocol in particular, but also recent developments regarding throwboxes, routing protocols and Android implementations are used as a starting point for this work. Based on past achievements, a scope for this Master Thesis is laid out.

In Chapter 3, the approach of conceiving our delay-tolerant network is described. Beginning with a discussion about underlying networking technologies, the chapter continues to point out shortcomings in the design of the existing routing protocols PRoPHET and PRoPHET+ and extends them based on that observation. Throughout the rest of the chapter, a new routing protocol is presented. Its interaction with the overall network protocol is described.

Chapter 4 discusses design choices and technical details concerning the implementation of the DTN framework. The underlying JDTN library is briefly described, and a focus is laid on different components that form together the whole framework. Furthermore, this chapter sheds light upon the WiFi and Bluetooth technologies from a developer's point of view.

An evaluation of the achievements is treated in Chapter 5. An experiment has been conducted to test the framework's operation under real-world conditions, as well as to get practical insights about WiFi and Bluetooth communication and its utility with respect to delay-tolerant networking. A simulation software written as a part of the thesis is used to evaluate the routing protocol in two scenarios: a community-based random mobility model and a dataset containing GPS traces.

The results are concluded in Chapter 6.

Chapter 2

Related Work

Delay tolerant networking has its origins in the middle of the 20th century, where space travel imposed new challenges to long-distance communication, such as significant message delays and intermittent links. This eventually led to the development of the network protocol InterPlaNet (IPN) and later the Interplanetary Internet [2, 3]. With the upcoming of portable devices and wireless communication technologies for personal use in the 1990s, the research in delay-tolerant networking was extended to terrestrial applications. In the first years of the 2000s, efforts were made to explore DTN capabilities using the more and more widely deployed WiFi 802.11 standard. Starting with laptops and PDAs around 2000, a larger number of portable devices have been equipped with wireless communication technologies over the years. At the same time, research in ad-hoc and sensor networks was shifted towards mobile ad-hoc networks (MANETs) [4, 5].

Delay tolerant networks have been deployed in larger scale, in order to investigate their behavior under real-world influences. One example is the N4C project [6], where testbeds in rural areas of Sweden and Slovenia have been set up, to gather reindeer- and meteorology-related data and to provide basic network access like e-mail to disconnected villages. Vehicular Delay Tolerant Networks (VDTNs) are an extension of DTNs to transit networks, with applications ranging from traffic information exchange to Internet access in rural areas [7].

2.1 Throwboxes

The concept of *throwboxes* was first mentioned in 2006, in the context of DieselNet at the University of Massachusetts [8]. The idea of throwboxes lies in the creation of new contact opportunities by placing stationary nodes in strategic locations, in addition to the mobile *peers*. These stationary access points often provide more storage, higher processing power and a more stable power supply than mobile nodes [9]. The DieselNet project is a vehicular network consisting of buses carrying WiFi routers [10], with the main purpose of collecting real-world traces for DTN research. When the buses approach each other, they establish a WiFi connection; stationary open WiFi access points are used to contact a central server and upload collected data. DieselNet was extended by placing throwboxes in selected locations in the town.

Another project, KioskNet, was developed by the University of Waterloo, Canada [11]. Internet kiosks in remote villages provide means of cheap Internet connectivity, however they are often faced by various technical issues, which KioskNet attempts to address. One of its improvements lies in buses that connect kiosks to central servers with reliable Internet connectivity. Based on insights in KioskNet, the University of Waterloo later developed the VLink delay tolerant network [12]. In VLink, data is transmitted through USB keys, as they provide a cheap and effective way to transport large amount of data. For urgent messages, a SMS-based text service can be used. DTN has also been an interest in military applications, for example in the SPINDLE project from the United States DARPA [13].

A common scenario involving throwboxes is data exchange by means of so-called *data mules* or *ferries*. Data mules act as carriers for messages, often in the form of vehicles to connect locations separated by large distances [7, 14, 15]. However, when mobile devices do not exchange data among each other, they are required travel the whole distance between two throwboxes. Other DTN scenarios consider a network of only mobile devices [16]. In those cases, throwboxes can enhance the network performance, since data can be transmitted between two nodes that pass the same location at different times. In this work, we aim to combine both approaches: direct data exchange between mobile peers, in addition to strategically placed throwboxes.

2.2 The Bundle Protocol

A notable milestone in DTN research was the foundation of the Delay Tolerant Networking Research Group (DTNRG) [17] as a part of the Internet Research Task Force (IRTF). The DTNRG has focused research in the area of DTNs and published the RFC 4838 [18] as a first attempt to standardize the network architecture. Their later published RFC 5050 [19] introduced the Bundle Protocol (BP), a more concrete specification of the network architecture focused on data exchange via *bundles*. There have since been several extensions, for example the Bundle Security Protocol (BSP) [20].

Along with the written specification, DTN2 was developed, a reference implementation of the Bundle Protocol written in C++. DTN2 serves as a base for future development and real-world usage and has been ported to several platforms and programming languages.

The most important aspects of the Bundle Protocol can be summarized as follows:

- A new OSI layer, the *bundle layer*, is introduced between transport layer and application layer. This makes it possible to abstract from underlying network technologies and run BP on top of existing transport protocols.

- Almost no assumptions are made about the network topology. In delay- and disruption-tolerant networks, constantly changing connections and breaking links are not exceptional, but rather the main case. This leads to the use of the *store-carry-forward* approach, where devices often do not immediately forward a packet, but instead store it until a further opportunity to transmit it emerges.

- The *bundle* denotes the basic unit of transmission in BP. Like other packet formats, bundles store a header and payload, which can wrap packets of lower-layer network protocols.

- The Bundle Protocol is conceived to fit into existing technology. Single links can be operated over common transport protocols such as TCP, to benefit of the provided reliability. Furthermore, BP uses an URI addressing scheme to be compliant with Internet technologies.

2.3 DTN on Android

The emergence of smart phones towards the end of the decade opened a whole new domain for delay-tolerant networking. The Google Android operating system has gained massive popularity in the recent years and holds currently the biggest market share among mobile operating systems [21]. In contrast to other systems such as iOS, Android comes with a very open philosophy. Being open-source on one hand and making it very easy to develop and deploy applications on the other, Android has been a popular choice for scientific and commercial applications on smart phones. As such, it has also been a target platform for several applications built on top of a delay-tolerant network architecture. At the time of this writing, the following Android ports are available:

- **Bytewalla** is a research project of the KTH Royal Institute of Technology in Sweden [22]. The Bytewalla Android app was developed over multiple years by different students, adding more features in each revision. It is inspired by the DTN2 reference implementation and provides additional features such as PRoPHET routing or security protocols.
- **HURRywalla** is an adaptation of Bytewalla with HURRy, a custom routing protocol [23].
- **NUS DTN** has been developed at the National University of Singapore and provides a layer between hardware/networking and application [24]. It comes with several apps, but uses its custom DTN protocol in place of BP.
- **BP-RI** is a project of the Ohio University [25]. It builds on top of the Licklider Transmission Protocol (LTP), which originates from the same university and is specified in RFC 5326 [26].
- **IBR-DTN** is another implementation of the Bundle Protocol [27, 28]. It is written in C++ and has a Java port for Android, but it is primarily designed for embedded systems.
- **SocialDTN** is an Android implementation using the Bluetooth convergence layer and dLife social routing [29]. The source code is not publicly available.
- **JDTN** is a Java port of the DTN2 reference implementation for Desktop and Android systems, which was developed by Cisco Systems. It implements the Bundle Protocol and provides TCP, UDP and LTP convergence layers.

In order to develop a DTN implementation on Android, one of these implementations can be reused, or a new protocol could be developed from scratch. Given the number of existing implementations and the fact that, with the Bundle Protocol, there exists a de facto standard that is likely to be used in future research and development, we decided against handcrafting our own solution. Instead, we built on top of a framework implementing BP as specified by the DTN research group.

Bytewalla is a promising candidate, however its code quality and documentation remarkably suffer from the number of different people that have worked for a short time on the project. The general impression was that the focus lies on providing as many features as possible rather than a robust and stable framework. The different versions are not gathered in a central place, and source code is only available for older ones, which makes it difficult to get an overview in order to know where to start developing. IBR-DTN is well-structured and documented, however its main development focus lies on Embedded Systems and C++, while Android seems to be the second priority. The last framework in question, Cisco JDTN, focuses on Java-based BP implementations for both Desktop and Android operating systems. As a professionally developed framework, its base is well-structured, covered with unit tests and extraordinarily well documented through the strict use of Javadoc and separate manuals of great detail. The fact that JDTN already provides support for TCP and UDP, comes with a few example applications and has a very liberal open-source license (BSD-3) makes it even more attractive as a base for our framework. We therefore decided to build the DTN framework on top of JDTN.

2.4 DTN routing protocols

Routing protocols are responsible of finding a *best* path in a network, along which packets are sent. What "best" concretely means depends on the scenario in which the routing protocol is deployed; often, it is associated with minimal latency and minimal packet loss.

The problem of routing in delay-tolerant networks can be described as follows: given a bundle addressed to a destination node, the node currently holding the bundle must determine to which neighbor node(s) it forwards the bundle, so that the latter has a high probability of arriving at the destination. DTN routing is a considerable challenge, since the node's view of the network is very limited and there is rarely a connected end-to-end path available. The fact that contact opportunities are in many cases short, rare and unpredictable adds to the difficulty of finding appropriate algorithms for routing in DTNs.

Since DTNs are intermittent, it is likely that a node does not know in advance the path the bundle is going to take. The idea is to transfer it to another node which is closer to the destination, i.e. which has a higher probability of reaching the destination. The challenge in DTN routing is to quantify this probability by means of different parameters. Knowledge of past contacts can prove helpful to infer future probabilities; for example, a node that was encountered in regular intervals is likely to be met again.

2.4.1 Overview of existing protocols

With respect to routing protocols in delay-tolerant networks, *epidemic routing* was an early and simple approach to cope with limitations of stationary ad-hoc networks, namely the lack of end-to-end connections [30]. The basic idea is to duplicate packets whenever they can be forwarded to a neighbor that has not seen them yet. Epidemic routing is described in greater detail in Subsection 2.4.2.

After the year 2000, a wide range of more sophisticated routing protocols were proposed. Most of them were probabilistic, and many followed the flooding approach of epidemic routing, but tried to address scalability issues. *MaxProp*, which was used in DieselNet, slightly enhances epidemic routing by ordering packets to forward and drop according to probabilities of arrival [31]. The *RAPID* protocol solves an optimization problem of specific network metrics such as average delay or delivery ratio; it does so by employing a utility function that rewards operations improving the metric [32]. *Spray and Wait* is a routing protocol using a restricted flooding-based approach; exactly N copies of a bundle exist in a network [33]. This is achieved by distributing initially N packets to different nodes, which then wait until they meet the destination node directly.

The *PRoPHET* routing protocol has been selected for this work, as it has had an enormous influence on research about DTN routing and was regularly used for comparison with new protocols [34]. PRoPHET assigns a delivery predictability value to DTN nodes based on the rate of direct and indirect contacts, so that nodes with higher predictability are prioritized in routes. A few shortcomings of PRoPHET were addressed in version 2, called PRoPHETv2 [35]. An extension of PRoPHET named PRoPHET+ was presented as an adaptive protocol that takes into account further factors such as available storage, power or location. Both protocols are described in Sections 2.4.3 and 2.4.4.

2.4.2 Epidemic routing

Epidemic routing is a very basic approach to route packets. Bundles are duplicated at every contact opportunity, eventually leading to a flooding of the network. While this increases the *delivery ratio* – the ratio of bundles that arrive at their destination – it imposes considerable resource utilization, most notably storage, transmission bandwidth and with it also power.

It is noteworthy that epidemic routing does **not** maximize the delivery ratio, because attempting to transmit bundles to every neighbor does not imply that these neighbors actually get the bundles. As the list of bundles to transmit grows over time, it is rather unlikely that a possibly short time window suffices to transmit all the bundles. Therefore, some bundles are implicitly excluded from forwarding in favor of others. A more sophisticated algorithm however could change the order of bundles to transmit, so that *important* ones arrive first.

2.4.3 PRoPHET

Probabilistic Routing Protocol using History of Encounters and Transitivity (PRoPHET) has been among the first and the most popular routing protocols in DTN research [34]. It has been improved and is now in its second version, PRoPHETv2 [35]. In the following, the term ”PRoPHET” is used to refer to version 2. As implied by the name, the protocol uses a probabilistic approach to find out which nodes are good candidates to forward bundles to, based on three metrics that affect an overall *delivery predictability* or *delivery value* (sometimes also simply referred to as *predictability*).

A delivery predictability is not an absolute quantity of a node in the network, but rather describes how a node is seen by another node. D_{AB} refers to the delivery value of B in A’s point of view. The higher it is, the more node A expects that bundles can be delivered via node B. It is noteworthy that delivery predictabilities are not symmetric: $D_{AB} \neq D_{BA}$ in general.

In PRoPHET, the delivery value is affected by the following three functions:

Refreshing

Refresh is a term coined by us, as the original proposal does not use strict terminology. When two nodes encounter each other, the predictability is *refreshed* by means of Equation 2.1. This models the idea that nodes, that are encountered more often, represent better candidates for forwarding. Given the current delivery predictability D_{AB}, the new predictability D^*_{AB} is computed as follows:

$$\begin{aligned} D^*_{AB} &= D_{AB} + (1 - D_{AB}) \cdot D_{enc} \\ D_{enc} &= \begin{cases} D_{max} \cdot \frac{I_B}{I_{typ}}, & \text{if } I_B \leq I_{typ} \\ D_{max}, & \text{otherwise} \end{cases} = D_{max} \cdot min\left(\frac{I_B}{I_{typ}}, 1\right). \end{aligned} \tag{2.1}$$

D_{max} is a constant. I_B denotes the time interval since there has been the last connection to node B, I_{typ} is a constant representing a typical connection interval.

Aging

The second feature of PRoPHET is the *aging* function, which decreases predictability over time. The idea is that contacts that have been encountered in the distant past are given less weight than recently seen ones. The new predictability is computed in the following way, where γ is a constant and k the number of passed time steps of predefined length:

$$D^*_{AB} = D_{AB} \cdot \gamma^k. \tag{2.2}$$

Transitivity

PRoPHET incorporates a third function that models *transitivity* in the network. The purpose of this function is to propagate delivery predictabilities further than a single hop. If node A has no direct contact to node C, but A often sees node B, which in turn has many contacts to node C, then C's delivery predictability from the view of A will increase. In other words, A indirectly learns about C, although it has never met that node.

The transitive function influences the predictability as shown in Equation 2.3, where β is a constant:

$$D^*_{AC} = max\left(D_{AC}, D_{AB} \cdot D_{BC} \cdot \beta\right). \tag{2.3}$$

2.4.4 PRoPHET+

As an extension of PRoPHET, the PRoPHET+ protocol was proposed in 2010 [36]. Besides delivery predictability based on the number of encounters, other parameters that affect the probability of arrival are considered. Possible parameters are:

- Available storage
- Power supply or battery status
- Wireless capabilities (bandwidth)
- Ratio of successful and total transmission attempts
- Location and mobility

The parameters are combined to a new metric, the *weighted function* V_{AB}. Each parameter p can be quantized as a value $v^p_{AB} \in [0, 1]$. Although the subscripts AB indicate that the parameters depend on a relation between node A and B, some of them only specify an attribute for the node B that is seen the same way by all possible nodes A. Examples are the available storage or power on that node.

In PRoPHET+, V_{AB} is computed as a weighted sum from these parameters, using s_p as the scaling factor for parameter p:

$$V_{AB} = \sum s_p \cdot v^p_{AB} \tag{2.4}$$

By choosing the scaling factors accordingly, it is possible to assign different priorities to the parameters.

It should be noted that in the original paper describing PRoPHET+ [36], the term ”popularity” is used to refer to a single parameter, namely the transmission success ratio. In this work, ”popularity” refers to a value that is computed by combining multiple parameters. It is introduced in Section 3.2.2.

2.4.5 Routing protocols used in this work

This work uses a combination of PRoPHET+ and PRoPHETv2 as its foundation. It reuses PRoPHETv2’s idea of ranking nodes based on three separate functions for refreshing, aging and transitive updates, as well as the PRoPHET+ approach to consider multiple parameters in the node metric. This work introduces a new parameter depending on the node type, which allows to differentiate between stationary access points (throwboxes) and mobile devices (peers), in order to prioritize one over the other when building routes.

PRoPHETv2 is an encounter-based protocol, which implies some shortcomings that can lead to counter-productive prioritizing of routing metrics. PRoPHET+ comes with a weakness in the way it combines its parameters. This work addresses these problems and proposes a new routing protocol as an extension of PRoPHETv2 and PRoPHET+. The protocol is discussed in further detail in Section 3.2.1.

Chapter 3

Approach

In this work, we aim to increase the connectivity of devices in areas that are disconnected from the world, i.e. that do not provide stationary or mobile Internet access. Delay-tolerant networks are an approach to establish communication over a longer time in intermittent scenarios, as such they represent an interesting choice for our work. We also aim to incorporate *throwboxes*, stationary access points that are supposed to improve network performance compared to direct communication between mobile devices (*peers*). In order to build our own delay-tolerant network, the following steps are necessary:

- Develop a DTN software framework that can be run on Android and Desktop systems, for both throwboxes and mobile peers. This framework aims to abstract from the networking part and to provide a simple interface for higher-level applications running in an intermittent contact scenario.
- Extend PRoPHET, a routing protocol that has convinced by ranking nodes according to the number of encounters, in order to account for new properties of our throwbox setting.

Section 3.1 describes the underlying wireless technologies that have driven the design of our delay-tolerant network. Section 3.2 takes a deeper look at routing in DTNs. Beginning with a discussion about existing routing protocols, some shortcomings of PRoPHET and PRoPHET+ are pointed out. Extensions are proposed to mitigate those problems, and to account for new properties of our throwbox setting.

3.1 Wireless networking

This section describes the interaction with the wireless network adapters on Android. We decided to use two wireless communication technologies:

- **WiFi (IEEE 802.11).** As a ubiquitous standard for wireless local area networks (WLANs) which is available on every smart phone and laptop, 802.11 is a popular approach for communication in DTNs. With a decent range and a high throughput, WiFi can be used to associate with stationary access points acting as DTN throwboxes. Unlike WiFi Direct, access point WiFi is an asymmetric technology requiring a client-server architecture. Throughputs of IEEE 802.11n, which is the latest standard supported by the phones we used, go theoretically up to 600 Mbit/s [37].

- **Bluetooth (originally IEEE 802.15.1, now Special Interest Group).** Conceived for personal area networks (PANs), Bluetooth is a wireless communication technology with shorter range, lower throughput and lower energy consumption than WiFi [38]. Bluetooth is symmetric and requires no centralized infrastructure. We decided to use Bluetooth for communication between mobile devices, to add more contact opportunities in the delay-tolerant network.

Figure 3.1 visualizes the connections in a possible DTN setting with access points and mobile devices. The mobile Android devices share Bluetooth connections among each other, and some of them are connected to stationary WiFi access points.

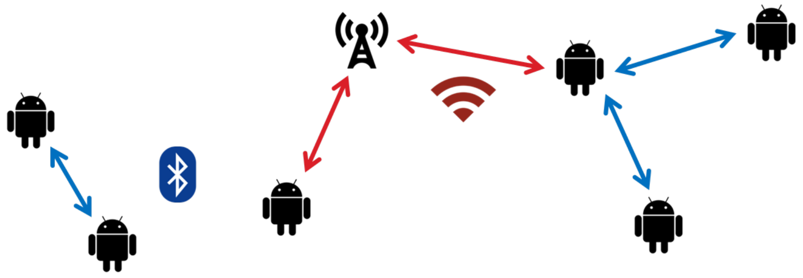


Figure 3.1: DTN setting with WiFi access points and mobile smart phones.

Although there are many physical and implementation-specific details that differentiate WiFi and Bluetooth, the basic concepts used for both are similar. Before communication via the Bundle Protocol is possible, the devices need to discover each other, establish a connection on several networking layers and exchange metadata. These steps are briefly explained in the following subsections; for a more detailed explanation with respect to the implementation on Android, consult Section 4.2.2.

3.1.1 Neighbor discovery

Both WiFi and Bluetooth provide a discovery mechanism that allows a device to learn about neighbors using the same wireless technology, although it works differently for both.

In Bluetooth, the discovering device broadcasts messages in its area. Bluetooth devices that are in *discoverable* mode respond to these announcements, notifying the discovering device about their presence. The response contains information required to establish a connection, such as device name, MAC address or the set of provided service IDs. A Bluetooth discovery lasts for 10-16 seconds and can detect multiple devices at once. It has to be triggered explicitly and is a quite energy-consuming process.

WiFi access points, on the other hand, constantly broadcast their service set identifier (SSID) which can be seen by any WiFi-enabled devices. WiFi devices typically look for these announcements in regular intervals without user interaction. On Android, WiFi networks are scanned every few seconds.

In order to recognize networks that run the DTN framework, predefined service IDs (Bluetooth) and SSIDs (WiFi) are used. Therefore, a device directly knows whether discovered devices are worth connecting to.

3.1.2 Connection establishment

In order to build up a connection, one device must initiate a connection attempt and the other must accept it. The initiating device can be considered a client, while the accepting device acts as a server. For WiFi, the access point is always the server and the mobile device the client. Regarding Bluetooth, every device can act as a server or client during connection establishment – once the connection is set up, the initial roles become irrelevant, and further communication is symmetric.

On top of WiFi, the Transmission Control Protocol (TCP) is used for reliable communication. Bluetooth on Android is provided using the Radio Frequency Communication (RFCOMM) protocol, which is a stream-based reliable transport protocol similar to TCP [39].

3.1.3 Disconnection

The most common reason why two devices disconnect is when they leave communication range (there are also other reasons, e.g. the user disabling the wireless adapter). When disconnection occurs, the DTN framework should be informed as quickly as possible, since its further decisions must be adapted accordingly. The bundle forwarding strategy, the computation of routing metrics as well as the measuring of connection times and other statistics depends on the availability of connection to a certain device, therefore a fast notification is desirable.

3.2 Developing a routing protocol

This section describes the routing protocol developed during this work. Starting with an investigation of PRoPHET and PRoPHET+, a few shortcomings in their equations are pointed out. Based on that, an approach that mitigates those problems is proposed.

3.2.1 Problems of existing routing protocols

Weaknesses in PRoPHET

PRoPHET's refresh equation (2.1) has been improved from version 1 to 2 because of a fundamental flaw. Originally, only the number of encounters has affected the delivery value, leading to the paradox situation where many connection/disconnection cycles within a short amount of time are considered better than a lasting, stable connection. This has been partially mitigated in PRoPHETv2 by incorporating the time since the last encounter [35].

However, one problem remains: the increase of the delivery value during one connection is upper-bounded by the minimum function. This is not a bad decision per se, but it *still* leads to the counter-productive situation where interrupted connections are sometimes better than stable ones. Consider a situation where two nodes are connected for three times the duration of the typical interval, that is, $I_B = 3I_{typ}$. The D_{enc} value which expresses the increase in the delivery value is then:

$$D_{enc} = D_{max} \cdot min\left(\frac{I_B}{I_{typ}}, 1\right) = D_{max} \cdot min\left(3, 1\right) = D_{max}. \tag{3.1}$$

Now, consider the situation where two nodes are connected for the duration of I_{typ}, then disconnected for I_{typ} and then again connected for I_{typ}. This yields an increase as follows – once for every connection, thus twice in total:

$$D_{enc} = D_{max} \cdot min\left(\frac{I_B}{I_{typ}}, 1\right) = D_{max} \cdot min\left(1, 1\right) = D_{max}. \tag{3.2}$$

Therefore, the delivery value is increased twice, although the nodes have been connected for only two thirds of the time and they have been disconnected for the duration of I_{typ}. One might argue that I_{typ} has to be chosen high enough, but that only shifts the problem and negates the idea that this interval is supposed to be a *typical* connection interval.

A further problem in PRoPHET is the fact that it updates the delivery values by means of the refresh function only on encounter events. This implies that the delivery value D_{AB} is not increased during the connection of two nodes A and B. An interesting case occurs when A and B do not know each other and meet for the first time: the mutual delivery value remains at the initial value, even if both are connected for a whole day. The effects are more severe when combined with the transitive PRoPHET property: a third node C that encounters A after one day will believe that B is quite unpopular in A's view (even more so, if PRoPHET aging decreases the popularity during that day). As a result, C might not forward a bundle destined to B, although it would immediately be delivered once A receives it. Combined with the aforementioned issue, the consequence is that delivery values would be more appropriate when A and B disconnected every hour just to reconnect again.

These issues are inherent to metrics that are recomputed in an event-based rather than time-based manner, especially if the frequency of events (in this case node encounters) is not predictable – which it hardly ever is in delay-tolerant networks. Even the most sophisticated formula to update the delivery values cannot help if it is not recomputed often enough to keep the network state up-to-date. This can lead to bad routing decisions because of incomplete information.

A time-based approach that does not suffer from these limitations is proposed in Subsection 3.2.3.

Weaknesses in PRoPHET+

A weighted sum of multiple parameter values is a simple and effective approach to assign priorities to different influencing factors. However, the use of addition comes with the semantics of logical disjunction:

one parameter **or** another can be used to increase V_{AB}. As a consequence, one parameter can compensate for another one – if one value is zero, the same value of V_{AB} can still be reached if another value is sufficiently high.

This approach is not always appropriate. There are parameters that are crucial to the operation of a node, in the sense that if the corresponding value is zero, the node should not be considered for routing at all. This concerns many of the above-mentioned parameters. For example, a node of which the storage is depleted is not a meaningful forwarding candidate, **even if** it is a very "good" node otherwise. Or a device that is about to shutdown because of low battery is very unlikely to deliver anything, in spite of otherwise high parameter values.

This brings the need for logical conjunction: a node must have free storage **and** enough battery to be a good candidate for bundle forwarding. Mathematically, this can be modeled by multiplying the parameter's values instead of adding them. This ensures that if one parameter is zero, the whole product will be zero as well. Nevertheless, relying on products only can have disadvantages, too, as there can indeed be parameters that compensate each other. A combination of both products for parameters in a logical AND relation and sums for parameters in a logical OR relation would be ideal. In this work, products are used, as the considered parameters are not optional.

The routing protocol developed in this Master Thesis is based on both PRoPHETv2 and PRoPHET+. It attempts to combine the strengths of both protocols, as well as to mitigate their weaknesses and to adapt them to our specific setting with throwboxes.

3.2.2 New concepts

The following sections introduce aspects that have not been part of the PRoPHET or PRoPHET+ routing protocols. In our custom routing protocol, new parameters are introduced to address the weaknesses described in Subsection 3.2.1. These are the contact time on one hand, the separate treatment of throwboxes on the other.

Contact time

The problem with event-based popularity updates as described in Section 3.2.1 can be addressed by relying on the contact time rather than the number of encounters. Two nodes are *in contact* or *connected* when they are able to exchange user data in the form of DTN bundles. We define the *contact time* τ as the duration during which two nodes are in contact. This time excludes neighbor discovery and connection establishment on physical, link, network, transport and bundle layers. The contact time measurement begins once two nodes have exchanged metadata so they are ready for bundle transmission, and it ends as soon as connectivity is lost.

The *contact time ratio* $\rho_{AB} \in [0, 1]$ is a quantity that expresses to what fractional part of elapsed time two nodes A and B have been connected. It is computed as follows:

$$
\begin{aligned}
\rho_{AB} &= \frac{1}{T_W} \cdot \int\limits_{t_{now}-T_W}^{t_{now}} c_{AB}(t)\,dt \\
c_{AB}(t) &= \begin{cases} 1, & \text{the nodes A and B are connected at time } t \\ 0, & \text{otherwise} \end{cases},
\end{aligned}
\tag{3.3}
$$

where t_{now} is the current time. T_W is the *contact time window*, the duration during which the contact time ratio is measured. T_W determines a time point in the past, before which being in connection or not has no influence on the contact time ratio. In other words, T_W expresses for how long nodes remember their contact times.

Throwbox-aware routing

One aspect that differentiates our scenario from classical DTNs is the coexistence of two node types: throwboxes and peers. While peers are mobile and act as carriers for bundles, throwboxes are stationary and can be placed in strategic locations to increase the contact opportunities. On one hand, they are able to connect to peers that traverse the location at different times, by forwarding bundles from the earlier peer to the later one. On the other, throwboxes typically have higher wireless range and bandwidth, which allows them to cover a larger area and compensate for short contact times with higher throughput. Furthermore, stationary devices such as computers or WiFi routers often have access to hard disks of large capacity and are supplied by the power grid, which makes them more robust nodes in the network. Some throwboxes may also have access to the Internet and thus act as a gateway between a disconnected DTN and the classical TCP/IP infrastructure.

These semantic differences demand for a special treatment of throwboxes in the delay-tolerant network. Because of their robustness, their enhanced communication capabilities and their strategic placement, throwboxes are a popular target for bundle forwarding, possibly more popular than other peers. It is standing to reason to account for throwboxes in the routing protocol, by making routes over throwboxes the preferred choice. In our setting, we introduce a new parameter called *node type factor* n_X that depends on the node type of X:

$$
n_X = \begin{cases} 1, & \text{node } X \text{ is a throwbox} \\ C_{peer}, & \text{node } X \text{ is a peer} \end{cases},
\tag{3.4}
$$

where C_{peer} is a constant.

Node popularity

The concept of *node popularity* as used by our routing protocol is very similar to the delivery predictability from PRoPHET or the weighted sum from PRoPHET+. Popularity is not a property of a node, but rather a relation between two nodes. The value P_{AB} is said to be the popularity of node B, from the point of view of node A, and it is in general different from the reverse P_{BA}. It expresses how desirable it is for node A to forward bundles to node B.

Since popularity is a quantity that depends on the point of view, a node in the network can be very popular for one node and very unpopular for another node. Furthermore, a node does not know its own popularity as viewed from any other node.

The absolute value of a popularity has no meaning, only the comparison to the popularity of other nodes **viewed from the same node** contains useful information. Locally ordering the nodes according to their popularity makes it possible to meet routing and forwarding decisions.

3.2.3 Regular popularity updates

In contrast to PRoPHET, our routing protocol updates the popularities in regular intervals rather than at fixed events. The popularity update interval T_U is a fixed parameter. Every time it elapses, a node A recomputes the popularity P_{AB} of node B in its view according to the following functions.

Refreshing

The refresh function is responsible for increasing the popularity when two nodes are in contact. It computes a new popularity P^*_{AB} out of the current popularity P_{AB}:

$$P^*_{AB} = P_{AB} + (1 - P_{AB}) \cdot \rho_{AB} \cdot n_B \tag{3.5}$$

ρ_{AB} is the contact time ratio during a constant contact time window T_W. When two nodes are not connected for T_W, it will be zero, leaving the popularity unchanged. The contact time window T_W can be chosen to match the update interval T_U, however it can be meaningful to consider a longer time span in the past.

n_B is the node type factor of node B. It is either 1 (if B is a throwbox), or a predefined constant C_{peer} (if B is a peer). In the latter case, the popularity increases at a smaller rate.

Aging

Aging is also slightly adapted from PRoPHET. Instead of decreasing popularities unconditionally, it takes into account whether nodes are currently connected, and does not decrease the popularity while they are. Again, the contact time ratio ρ_{AB} is used to determine the ratio of time during which there has been a connection. The popularity changes as follows:

$$P^*_{AB} = P_{AB} \cdot \gamma^{1-\rho_{AB}} \tag{3.6}$$

When the nodes have been connected for longer than T_W, ρ_{AB} will be 1, and $\gamma^{1-\rho_{AB}}$ will also be 1. This stops aging as soon as a connection lasts long enough.

Transitivity

It is not possible to compute transitivity in a purely time-based way, because it depends on information of a third-party node to which the current node is possibly not connected. That is why nodes need to exchange metadata, the process of which is described in Section 3.2.4. As long as this metadata exchange happens in regular intervals, we are very close to a time-based (rather than event-based) approach.

Our routing protocol recomputes the transitivity function on a node A whenever it receives contact information about node C from another node B. In this case, the popularity P_{AC} expresses how popular C is in the eyes of A. Apart from the fact that this information is received in regular intervals and not once on encounters, the transitivity function from PRoPHET is reused:

$$P^*_{AC} = max\left(P_{AC}, P_{AB} \cdot P_{BC} \cdot \beta\right) \tag{3.7}$$

3.2.4 Contact history

Every node maintains a list of contacts it has known in the past. The *contact history* contains a record for encountered nodes with the following information:

- DTN node name
- Type of node (*throwbox* for access points, *peer* for mobile devices)
- Popularity
- Contact times

Whenever a connection to a new node is successfully established, an *encountered* event will be triggered. Loss of connection leads to a *leave* event. These events are processed by the contact history, which updates its entries correspondingly.

3.2.5 Routes and routing table

A *route* maps a destination node to a next-hop node. A *routing table* stores all available routes. The routing table only contains routes with a next-hop node that is currently connected to the node.

A *default route* maps any destination to a next-hop node and is consulted if no other route is found. Default routes allow forwarding to destinations even if those are unknown to the node, which occurs frequently in delay-tolerant networks.

Routes in the routing table are updated when a node is encountered or left or when the popularity is updated. Routes are generated according to the following algorithm:

```
1: for all connected neighbors N do
2:     Add a route with node N as destination and next-hop.
3:     if N is among the most popular nodes then
4:         Add a default route with node N as next-hop.
5:     end if
6: end for
```

Every bundle is constrained by a duplication limit Λ that puts an upper bound on the number of times it can be forwarded from one hop. This enables a restricted form of flooding, which limits the utilization of network resources. The bundle duplication limit is independent for bundles on different nodes; every time a node receives a new bundle, its duplication limit is reset.

Given a bundle, the routing table infers from its destination a list of next-hop nodes, to which nodes the bundle can be forwarded. Forwarding is constrained by further conditions:

- The next-hop node must not know the bundle already. The way this condition is verified is described in detail in Subsection 3.3.1.
- The bundle's duplication limit must be greater than zero.

If these conditions hold, the bundle is considered *outbound* and put into a queue where it waits for its transmission. Immediately before the transmission, the algorithm checks again whether connection to the next-hop node is available and whether the two constraints still hold, since any of those may have changed in the meantime. If the bundle is no longer ready for forwarding, the bundle duplication limit is increased and the bundle is stored, until another forwarding opportunity emerges.

3.3 Processing DTN bundles

This section describes other components of our network protocol that are not part of the routing algorithm. From a semantic perspective, the way how DTN bundles are processed and memorized is explained.

3.3.1 Bundle history

Every bundle is identified by its *bundle ID*, which is a pair of source node name and a sequence number. The bundle ID is globally unique, given that node names (DTN endpoint identifiers) are unique.

The *bundle history* is a list of bundle IDs. Every node maintains a bundle history for itself, which contains the IDs of all the bundles it has seen in the past. In addition, the node maintains a bundle history for neighbor nodes and updates them when it encounters them.

The purpose of bundle histories is to memorize which node has already processed which bundles, and to avoid useless transmissions of large bundles. A node is supposed to have always an up-to-date view of its neighbor nodes' bundles, so that it can decide whether forwarding is appropriate or not. To achieve this, the network protocol exchanges bundle IDs with connected neighbors under the following circumstances:

- When two nodes meet, they initially exchange their bundle histories, so that every node learns which bundles the other node already knows. Bundles that are already known by the remote node are not transmitted.
- Every time a node receives a bundle, it informs all its currently connected neighbors (except the last hop of the bundle) about the bundle ID, so that they know that the node has seen the bundle.
- When a node transmits a bundle successfully, it learns that the receiver knows the bundle. Recognizing successful arrival is possible because of reliable transport protocols.

Every bundle is assigned a lifetime (time to live) that determines for how long the bundle is retained in the network. This time depends on the application; ping bundles have a lifetime of a few seconds, whereas e-mail messages may live for several days. When a bundle's lifetime expires, every node holding it will discard it. All bundle ID entries referring to that bundle are deleted.

3.3.2 Bundle dispatching

When a node receives a bundle, it has to decide how to further process it. This decision depends on the bundle destination, the available routes and the configured forwarding policies. The following actions can be taken:

- **Discard.** There are various reasons why a bundle is discarded. The bundle's lifetime may have expired, there may have been an error with decoding or there is not enough storage left to hold the bundle.
- **Hold.** The bundle is stored locally until it is processed again at a later time. This is the most prominent option in delay-tolerant networking, since there is usually no opportunity to forward a contact directly.
- **Forward.** If a matching route is immediately available, the bundle can be forwarded along it.
- **Deliver.** If the bundle is destined to the current node, it will be delivered to the application.

Changes in the current contact situation (e.g. encounter of a new node) trigger an update of stored bundles, so that they are reconsidered for dispatching.

Chapter 4

Implementation

4.1 JDTN and the Bundle Protocol

Based on an evaluation of existing DTN frameworks (Section 2.3), we choose JDTN [40] as a base for our DTN framework. JDTN is a library written in Java which implements the Bundle Protocol (BP) [19]. As such, it can be regarded a Java port of the DTN2 reference implementation provided by the DTNRG [17].

JDTN is written in a modular way. The components that have been used for this Master Thesis are described in the following subsections. Figure 4.1 gives an overview over the most important modules in JDTN. The core of the library is the implementation of the Bundle Protocol, which includes the following parts: the Bundle Protocol agent, data structures for neighbors, links and addresses, and a module taking care of bundle encoding and decoding. The convergence layers bind the BP to lower-layer network technologies. On top of the Bundle Protocol implementation, several sample applications have been written. A graphical user interface on Android and a command-line interface on Desktop systems allows the user to interact with JDTN.

This work uses only a part of the functionality provided by JDTN. A complete and detailed description of the framework's capabilities can be found in the JDTN tutorial, a manual accompanying the SourceForge repository [40].

4.1.1 Neighbors, links and addresses

In JDTN terms, a *neighbor* corresponds to the notion of a node in the network. Each neighbor contains an *endpoint identifier* (short *endpoint ID* or *EID*), which is a URI that identifies the node in the whole network. The URI scheme used in this work is `dtn:`. JDTN would also support the `ipn:` scheme for interoperability with the Interplanetary Internet [2]. An example endpoint ID is `dtn://host/app`, where the authority part `host` represents a globally unique *node name* and the path `app` refers to services or applications on that node. A specific null endpoint ID `dtn:none` is used to denote the absence of an addressing scheme.

A JDTN *link* describes a network interface, which deviates from the term "link" as it is often used in DTN literature. Links contain additional information depending on the underlying network technology. For example, a link running over the Internet Protocol is associated with the IP address of the remote host.

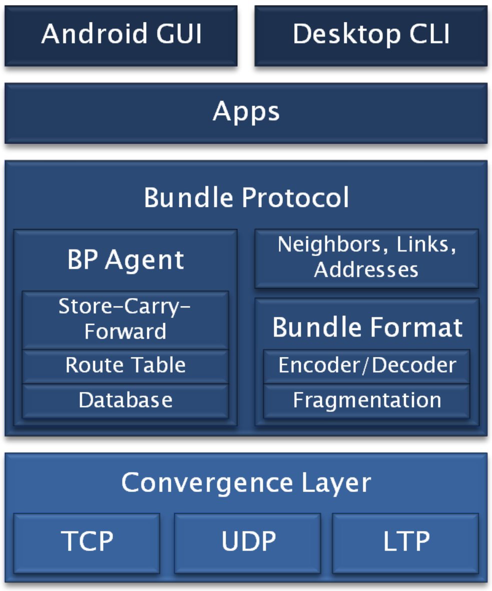


Figure 4.1: JDTN components

Links are closely related to *addresses*, which – in the JDTN sense – refer to generic host identifiers that are assigned different meaning depending on the OSI layer. On the data link layer, MAC addresses are used, while the network layer typically uses IP addresses. Both addresses and links are provided in the most abstract way possible, to allow seamless integration of further network protocols.

4.1.2 Bundle encoder and decoder

This module is responsible of transforming payloads to bundles and vice versa. The bundle format is specified in Section 4 of RFC 5050 [19]. A DTN bundle consists of multiple *blocks*. The *primary block* contains header information such as source and destination endpoint IDs, timestamps, sequence numbers and BP-specific processing flags. It also involves a dictionary of self-delimiting numeric values (SDNVs), which are used for compression of repeating patterns in the bundle. The *payload block* encodes the actual user data carried by the bundle. Further *extension blocks* can be used to implement additional features in BP, such as security.

JDTN implements the bundle format according to the BP specification, except for a few places like bundle fragmentation, which is only partially supported (proactive, not reactive). As a result, applications written using JDTN are widely interoperable with other frameworks implementing BP, in particular the reference implementation DTN2.

4.1.3 Bundle protocol agent

The *Bundle Protocol agent* (BPA) implements the store-carry-forward mechanism of delay-tolerant networks. Its responsibilities include:

Route table

A *route* maps an endpoint ID (more precisely, a regular expression that can potentially match multiple EIDs) to a next-hop neighbor. JDTN provides a routing table as a data structure to store routes. Given a bundle, the routing table allows quick lookup for an appropriate route. Routes are built depending on the routing strategy.

Routing exchange strategies

JDTN provides limited means of building the routes that can be stored in the routing table:

- **Static routing.** Every node is configured manually with respect to its routes to other nodes. Because of the big amount of manual interaction needed, this strategy is mainly useful for testing.
- **Hub-and-Spokes.** In a star-shaped network topology, a single BP router is the center node (the *hub*), and other nodes (the *spokes*) are connected to it. Every spoke node has a single route to the hub, but not to other nodes. Nodes announce themselves to the hub, which then learns about its neighbors and can forward a bundle from one neighbor to another. This approach has several drawbacks: the hub-and-spokes topology is not scalable and a failure of the hub brings down the whole network.

These routing strategies may be useful in very specific scenarios, but they do not address a general DTN situation with intermittent contacts and highly dynamic topology changes. That is, in such cases the generation of appropriate routes is delegated to the user of JDTN.

However, since the routing functionality is abstracted in a Java interface, it is easily possible to provide custom routing protocols of arbitrary complexity.

Store-carry-forward mechanism

The bundle dispatching mechanism as described in Section 3.3.2 is the responsibility of the Bundle Protocol agent. JDTN performs most of the tasks: for bundles coming from the convergence layer adapters, it meets the decision whether to forward, store or discard them.

There is a wide variety of policies, even besides the routing protocol itself, that affects the decisions of the Bundle Protocol agent. For example, it can be configured up to what size bundles are stored locally, or if they are stored at all (rather than forwarded immediately).

Also JDTN bundles themselves provide many additional options that are part of the BP specification, but not exploited in this work. For example, there is a reliability mechanism on the bundle layer called *custody transfer*, which makes certain nodes responsible of carrying the bundle. Another example are status reports, that are sent to specified end points whenever bundle-related events such as forwarding or deletion occur.

Serialization

Persistent data is stored in a SQLite database, configurations are stored in XML files. The bundle database maintains a separate file for every bundle. Bundles that exceed a certain size are never completely loaded into memory, since the available RAM is quite constrained on Android systems. Whenever a bundle is created, deleted or modified, JDTN will update the SQLite database accordingly.

In addition to stored bundles, temporary files are created to hold the payload of incoming bundles, so that the byte buffer need not be kept in memory. Once a bundle has been completely received and written to the database, the file can be deleted.

4.1.4 Convergence layer adapters

A *convergence layer adapter* (or simply *convergence layer*) binds the Bundle Protocol to the underlying network technologies. It acts as an abstraction layer so that BP need not know anything about lower OSI layers. JDTN already provides convergence layers for TCP and UDP (over IP) as well as for LTP, the Licklider Transmission Protocol [26]. LTP runs on top of UDP.

Convergence layer adapters are invoked by the Bundle Protocol agent whenever a forwarding decision is met. They receive the encoded bundle as a stream of bytes, referring to a memory buffer or a file for large bundles. The data can then be moved to Java network sockets. The opposite direction works analogously: when sockets receive a bundle, it is decoded and passed to the Bundle Protocol agent, which then determines how to further proceed.

4.1.5 Application and media repository

An *application* is a component that is built on top of the Bundle Protocol and that uses the delay-tolerant network service provided by JDTN to exchange data with other nodes. JDTN comes with several sample applications: Ping, Text Message, File Transfer, Image, Video and Voice Exchange, as well as Cisco-specific Router apps for their SAF and CAF protocols. The File Transfer application implements the DTN2-conforming `dtncp` interface, which allows JDTN to exchange files with DTN nodes that run another DTN2-compliant software.

Media exchanged by the applications is persistently stored in the *media repository*, a directory containing the exchanged files.

4.1.6 User interface

On Desktop systems, a command-line interface, the JDTN Shell, allows to inspect and modify the current state of a running JDTN application. For example, new neighbors, links and routes can be set for the static routing. The shell also gives access to a list of locally stored bundles, policies set in the Bundle Protocol agent and statistics about the convergence layers.

JDTN also comes with an Android application called JDTNG1. This application provides primarily a graphical user interface for JDTN on the Android system. JDTNG1 further provides alternative back-ends for components that have to be implemented differently on Android, e.g. database access.

4.2 Extending JDTN

The main part of this Master Thesis consisted of extending the JDTN Java library to a point, where it is able to act in DTN scenarios and meet autonomous decisions regarding WiFi/Bluetooth connectivity and bundle routing. For the rest of this work, the extended library is going to be referred to using the term *DTN framework*. The functionality covered by that framework is described in the remaining sections of this chapter.

While JDTN provides a robust foundation for DTN applications on Desktop and Android systems, it is not ready to be used directly in delay-tolerant networking. The following parts are missing:

- **Neighbor discovery.** On Android systems, JDTN does not interact with wireless network adapters (for WiFi, WiFi Direct or Bluetooth) in any way. The task of finding other devices and connecting to them is left to the user, until TCP or UDP sockets can be used. In the case of Bluetooth, no communication is possible at all, since neither TCP nor UDP are used as transport protocols.

- **Contact management.** JDTN stores a list of currently connected neighbors, but it has no notion of the past and future. It does not store how often it has been connected to a specific neighbor, or for how long.

- **Routing protocol.** The basic routing strategies (static and hub-and-spokes routing) are insufficient for delay-tolerant networking. A more dynamic approach needs to be found that considers information gathered by the contact management.

- **Metadata exchange.** When two nodes meet, JDTN exchanges information such as the node name during a negotiation phase. What it does not exchange is further metadata related to contacts and routing, which would be able to reduce unnecessary transmissions. For example, exchanging a list of bundles seen in the past would avoid that a node receives a bundle it already knows.

Figure 4.2 gives an overview over the software components that have been implemented in addition to JDTN. A new convergence layer for Bluetooth as well as an adapted one for TCP are now part of the framework. On Android, a major part of the efforts have been invested into the networking module, which includes neighbor discovery, connection management and a packet-based data exchange protocol. This part will be further described in Section 4.2.2. Furthermore, the concepts mentioned in Section 3.2 from the previous chapter have been implemented in the extended framework. This comprises the routing protocol and its interaction with the contact history and the bundle ID history. Implementation-specific details of this part are described in Section 4.2.3.

The extended framework contains also an API designed to make it easier for Java developers to write DTN applications. JDTN already provides an application interface, but it assumes a certain knowledge of the Bundle Protocol and exposes internally used types. This has the advantage of allowing the developer to fine-tune the interaction with the network, but in some cases it is easier if developers can operate directly on abstract byte streams. In Figure 4.2, this component is called “high-level app API”.

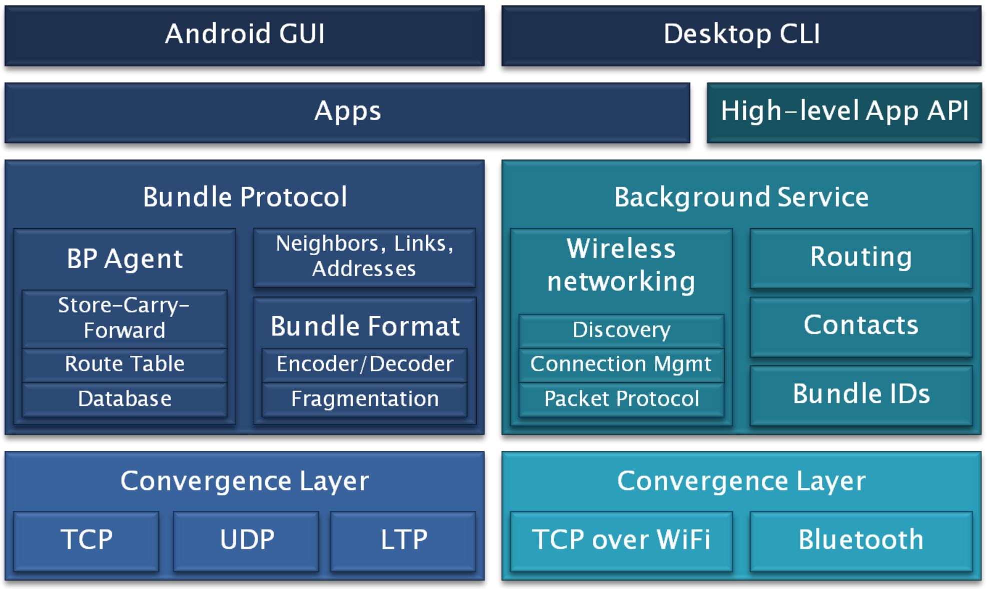


Figure 4.2: Components of the extended DTN framework. The left side shows the modules of the original JDTN library, while the right side shows the extension.

4.2.1 Android service

JDTN itself is running in an Android activity, thus the whole network communication requires the application to remain open. This work extends the Android application by running a background service which performs all the necessary work without interfering with other user actions. The service can be explicitly started and stopped.

The networking and BP parts run in separate threads, so they do not block the Android UI during longer computations. Threads communicate via message queues provided by Android's `Handler` class. This makes it possible to keep the threads modular and independent and elegantly avoids race conditions, but it imposes a minor latency for events passing through those queues.

4.2.2 Android wireless networking

The interaction with the WiFi and Bluetooth wireless interfaces constitutes a large part of the DTN framework on Android. Apart from minor platform-specific differences such as filesystem or database access, the networking part is the only module that differs between Desktop systems (Windows/Linux) and the mobile Android system.

For throwboxes, it is enough to run a DHCP server and a IEEE 802.11 access point service outside JDTN in order to provide connectivity on the data link and network layers. On the transport layer, the throwbox is looking for incoming TCP connections on a specific port, and accepts them to start communication with peers that request a connection. Within the DTN framework, this functionality is implemented using the standard Java socket API (`java.net.Socket`). As network sockets are an application-level abstraction of the underlying networking protocols, it would theoretically be possible to connect to a throwbox using different link-layer protocols than WiFi, for example Ethernet. This has not been tested however.

On the contrary, mobile Android devices act as peers that actively connect to other peers or to throwboxes, therefore APIs on a lower level than sockets must be used. For low-level network operations, the `android.net.wifi` and `android.bluetooth` APIs have been accessed.

The protocol stack used by the framework is depicted in Figure 4.3. The two convergence layers for WiFi and Bluetooth come with different sets of protocols up to the transport layer. Above WiFi IEEE 802.11 physical and link layer protocols, a TCP/IP architecture is used. For Bluetooth, a variety of protocols is responsible of operation on all OSI layers up to the transport layer. Bluetooth sockets build upon the Radio Frequency Communication Protocol (RFCOMM), which is a stream-based reliable transport protocol similar to TCP [39]. A Service Discovery Protocol (SDP) announces capabilities and services provided by the Bluetooth device.

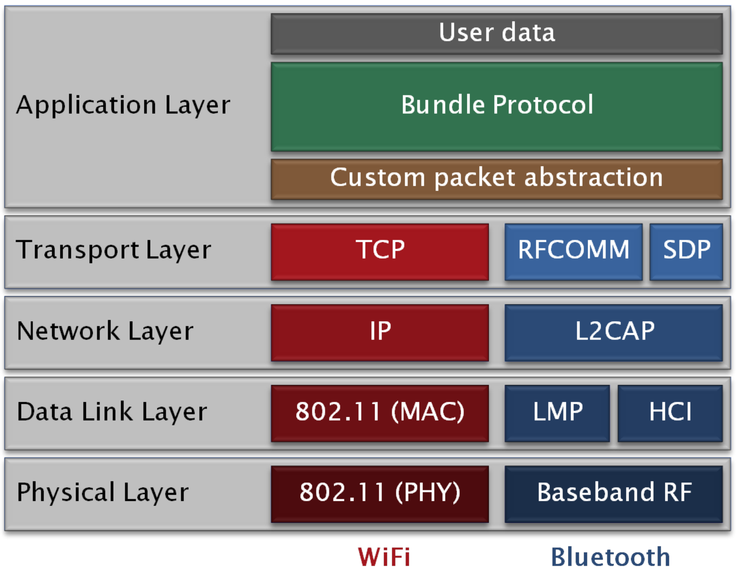


Figure 4.3: Protocol stack used by the DTN framework. For the lower layers, a simplified view is given.

On top of the two stream-based transport protocols TCP and RFCOMM, a simple packet format is used. Its sole purpose is to split the stream into packets for higher-level application logic. This protocol is briefly described in Subsection 4.2.2. Above it comes the bundle layer with the Bundle Protocol (BP) as specified by the DTN research group [19]. The bundle layer is responsible for transport of bundles in delay-tolerant networks. Actual application data that is sent over a DTN is packed into the payload of a bundle.

Connecting to a WiFi access point

IEEE 802.11 conforming access points (APs) broadcast a service set identifier (SSID) that can be read by all WiFi-capable devices in signal range. When the Android WiFi adapter is enabled, the system performs an automatic SSID scan every few seconds. Because this behavior is not clearly specified in the Android developer guidelines and the interval may vary between different phones, our framework performs additional WiFi scans in predefined intervals.

Once a scan is performed, the scan results are checked for an SSID that starts with a "dtn-ap" prefix, which is our way of identifying DTN throwboxes. If an access point matches, a connection to it is initiated. If multiple access points match, the one with the highest signal strength is selected. The access points are secured using a WPA2 key of which the application is aware.

Before two nodes are ready for data exchange, connectivity on several layers has to be established:

1. The 802.11 handshake is complete and the client device is authorized.
2. The smart phone is assigned an IP address by the access point's DHCP server.
3. A TCP connection is initiated from the client device and accepted by the access point.
4. Both peer and throwbox exchange metadata containing information about the DTN nodes.

In every step, an error may occur, which has to be handled accordingly. If connection establishment on the TCP layer fails (possibly because no DTN framework is running on the throwbox to accept incoming connections), a reconnection attempt is made after a fixed time. If connection on the 802.11 or IP layers fails, the device is disconnected and reconnects only after the next AP discovery.

Connection to a remote Bluetooth device

A smart phone running the DTN framework discovers neighboring Bluetooth devices in intervals of a few minutes. A Bluetooth discovery is explicitly triggered by the framework; it is a resource-consuming process where the device broadcasts a message that can be seen by other devices. Bluetooth devices that are configured to be in *discoverable* mode answer these broadcasts to express their presence.

For Bluetooth, every device runs both an initiating and an accepting thread, as a result of the symmetric communication in Bluetooth. After a remote Bluetooth device with the relevant service has been discovered, the framework attempts to initiate a connection to it. If the remote device accepts that connection and the handshake is successful, both Bluetooth devices will be connected after a few seconds. If the connection establishment fails, the initiating device immediately retries for a limited number of times. If all attempts fail, the next opportunity to connect is after the next discovery cycle of the initiating or the remote device. Since all devices running the DTN framework are performing Bluetooth neighbor discoveries, the average time to find a neighbor is actually shorter than the discovery interval.

Connection management

Once a connection has been successfully initiated or accepted, further communication happens symmetrically through network sockets. Both devices start to transmit metadata containing information about the DTN node. For every connection over WiFi or Bluetooth, two threads are run: one for transmitting data, one for receiving. The TCP and RFCOMM sockets provide blocking read and write operations, meaning that a thread waits until data has been received or transmitted, respectively.

When either the receiving or transmitting thread reports a problem (mostly due to leaving the wireless range of the remote device), the connection is immediately torn down. The contact history is updated with statistics about the connection, such as the contact time or the number of bytes transmitted and received. A *contact left* event is triggered, which cancels pending bundles forwarded to the other nodes and updates the routing table.

By default, TCP and RFCOMM sockets have a very long timeout. A disconnection may only be triggered when writing data to the socket (in the case of TCP not even then, depending on protocol settings). That is why a mechanism to detect disconnection quickly, ideally within seconds, is required. A simple approach is the transmission of *keep-alive* packets, whose sole purpose is to make sure that the connection is still up. In addition to keep-alive messages, a time-out mechanism resets the socket, when it has not received any data for a long time. It is important to consider that there will be no incoming keep-alive messages during the receipt of a large bundle (since packets are queued in a FIFO manner), thus the time-out is kept long enough to avoid this problem.

Packet-based communication protocol

The framework uses the transport protocols TCP (over WiFi) and RFCOMM (over Bluetooth) to communicate with other nodes. Both TCP and RFCOMM represent reliable stream-based protocols, such that the developer needn't take care of acknowledgments, retransmission, packet ordering or packet duplication. The Java APIs for both protocols are very similar; both are represented as a socket with corresponding input and output streams. It is thus possible to direct a memory buffer or a file directly into a socket stream, and it will be transmitted over the corresponding wireless interface.

JDTN provides a very sophisticated and complex TCP convergence layer, however we were not able to fully exploit its capabilities and had to restructure major parts to fit our purposes of initial metadata exchange. Eventually we decided to write our own simple convergence layer for TCP, also because it allowed us to reuse a lot of functionality for Bluetooth and WiFi. Because of the similarities between RFCOMM and TCP, an abstraction on top of the sockets was written. This allowed us to treat connection initiation, acceptance and maintenance as well as metadata exchange in a uniform way, without duplicating the code for both convergence layers.

Above the transport protocols, a simple protocol is used to exchange metadata and DTN bundles. The main purpose of this protocol is to provide a packet abstraction on top of the socket streams. The protocol specifies a message format which begins with a 5 byte header, followed by the payload. The header includes the type of packet and the payload length. The following types of packets are used:

- **Keep-alive.** A message that is sent every few seconds to avoid socket timeout and to check whether the connection is still available. Without keep-alive messages, it is possible that the link has long broken, but the framework does not know because there has been no attempt to transmit data, and the socket timeout has not elapsed yet.

- **Node announcement.** Exchanged when two nodes encounter, this packet contains information about the DTN node name, the contact history and the bundle history. Upon receiving this packet, a decision for forwarding of local bundles can be met.
- **Contact.** Exchanges information about an entry of the contact history containing node name, node type and popularity. This is used for updates of the transitive function in the PRoPHET extension.
- **Bundle ID.** Transmission of a bundle identifier states that the transmitting node has knowledge of the corresponding bundle and does not need to receive the bundle again.
- **DTN bundle.** This packet wraps an encoded bundle in the Bundle Protocol. It is used for transmission of user-data in the delay-tolerant network. Unlike other packets, DTN bundles are forwarded over multiple hops.

The receiver side dispatches the type of the packet and informs the corresponding components. For example, when a bundle is received, the JDTN Bundle Protocol agent will be notified and can further process the bundle.

4.2.3 Routing protocol

The DTN framework uses a popularity update mechanism based on Section 3.2. Depending on a neighbor's popularity, the state of connection and the list of bundles known by the neighbor, a route including him as a next-hop is created. This route is inserted in JDTN's routing table. The Bundle Protocol agent then takes care of forwarding bundles appropriately.

In order to support a store-carry-forward paradigm with bundle duplication, the Bundle Protocol agent in JDTN had to be extended. It was conceived with having only one copy per bundle – the bundle is either stored or forwarded. With the extension, a bundle can on one hand be stored **and** forwarded, retaining the local copy; on the other hand it can be forwarded to multiple neighbors instead of just one.

4.2.4 Serialization

The whole contact history is persistently stored in a SQLite database. This makes sure that the information is not lost if the DTN framework is stopped and later restarted.

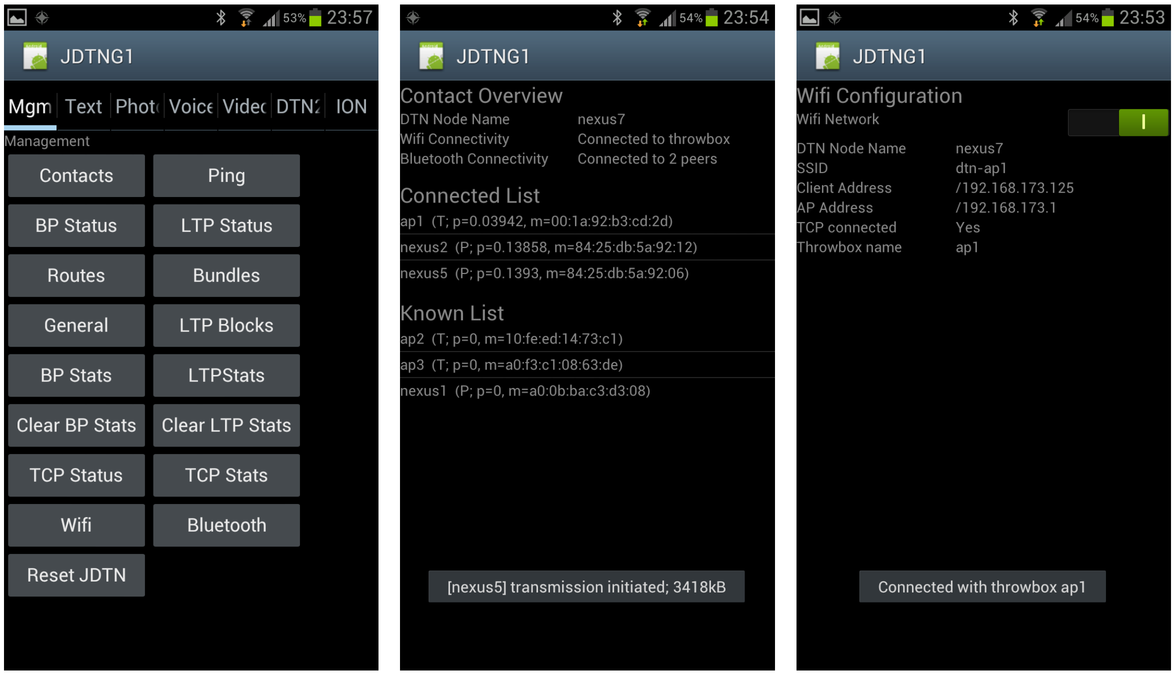
Configuration of the current node is stored in XML files. For Desktop throwboxes, the configuration includes parameters related to the bundle generation used during the experiment (interval, size and destination). For Android peers, WiFi- and Bluetooth-specific settings are serialized. Every configuration file also includes the DTN node name, which is set manually and must be globally unique.

4.2.5 User interface

The Android UI has been extended to give access to the new functionality. The application starts with a screen that allows to specify the node name and to start the background service. Once started, the main menu can be entered, which gives an overview over all the possible configuration options. New are the configurations for Bluetooth and WiFi, an overview of all the contacts, and an option to reset the framework to its original state, deleting all bundles and contacts.

On network events (encounter/leave of a node, transmission of a bundle), the user is notified by the Android service using a toast message.

Figure 4.4 shows the Android app in action. The application is an extension of JDTNG1; while the main menu existed already but with fewer buttons, the Contact and WiFi configuration screens are new.



Contact Overview

DTN Node Name	nexus7
Wifi Connectivity	Connected to throwbox
Bluetooth Connectivity	Connected to 2 peers

Connected List

```
ap1  (T; p=0.03942, m=00:1a:92:b3:cd:2d)
nexus2  (P; p=0.13858, m=84:25:db:5a:92:12)
nexus5  (P; p=0.1393, m=84:25:db:5a:92:06)
```

Known List

```
ap2  (T; p=0, m=10:fe:ed:14:73:c1)
ap3  (T; p=0, m=a0:f3:c1:08:63:de)
nexus1  (P; p=0, m=a0:0b:ba:c3:d3:08)
```

[nexus5] transmission initiated; 3418kB

Wifi Configuration

Wifi Network

DTN Node Name	nexus7
SSID	dtn-ap1
Client Address	/192.168.173.125
AP Address	/192.168.173.1
TCP connected	Yes
Throwbox name	ap1

Connected with throwbox ap1

(a) Main menu. At the top, various JDTN apps are visible. The buttons lead to other configuration and inspection screens, two of which are shown on the right.

(b) Overview of current and past contacts. The letter `T` or `P` differentiates between throwboxes and peers; `p` stands for popularity and `m` for MAC address. A toast message notifies the user that a transmission just started.

(c) Configuration screen for the WiFi network interface. The lines contain information about the connectivity on different OSI layers (IP address received, TCP connection established, BP node name and metadata exchanged). A toast message confirms the encounter of a throwbox.

Figure 4.4: Screenshots of JDTNG1

Chapter 5

Experiments and Results

In order to investigate the operation of the JDTN-based framework that was developed during this thesis, a real-world experiment involving people carrying smart phones has been conducted. The experiment's setup and its evaluation are described in Section 5.1. To test the routing protocol, a simulation software has been written in C++. The way it models the DTN network as well as the evaluation of DTN routing protocols is discussed in Section 5.2. Section 5.3 summarizes the results from the experiment and the simulation. A conclusion is drawn based on commonalities and differences between the real-world and simulation scenarios.

5.1 Real-world experiment

The intent behind the real-world experiment was to test the operation of the DTN framework and to get insights about the performance of WiFi and Bluetooth communication in a DTN scenario.

The experimental setup included three throwboxes that were realized as standard laptop computers equipped with an external USB wireless antenna. Two of them had a Linux Ubuntu 12.04 operating system installed, one was set up with Windows 7. Dedicated software was run to turn the computers into wireless access points – on one hand for 802.11 SSID broadcasting and handshaking on the link layer, on the other to provide IP addresses via DHCP on the network layer. On the Linux operating system, the tools `hostapd` and `dhcpd` have been used, respectively. On Windows, the built-in `netsh` command line interface was able to provide both access point and DHCP server in a simple way.

Eight smart phones of type Galaxy Nexus were distributed among members of the Communication Systems Group at ETH Zurich. The members had their offices in two adjacent floors. Two throwboxes were placed in offices, one on each floor, and the third throwbox was located at a break room on one of the floors, in the opposite corner of the building. The DTN framework software was run on every smart phone. Bluetooth and WiFi adapters were enabled throughout the whole duration of the experiment. The participants were instructed to do nothing but charge the phone over night or when the battery is low; the whole operation was automatic and required no user interaction.

Figure 5.1 shows a laptop computer acting as a throwbox, with an external WiFi antenna plugged in.

Next to it, several Galaxy Nexus smart phones are visible; they represented the mobile peers in the experiment.

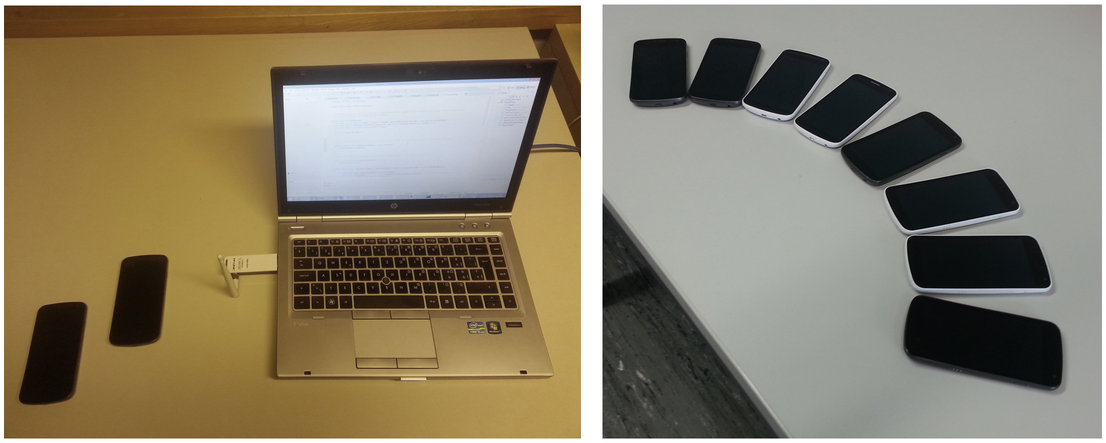


Figure 5.1: Laptop as access point and Galaxy Nexus phones.

Network load was generated by the throwboxes. In intervals of half an hour, every throwbox generated a file filled with random data. The file size was uniformly distributed between 1 KiB and 8 MiB. These sizes should model typical DTN applications such as e-mail or exchange of image or audio files. The generated file was then used as a payload for a DTN bundle, the bundle's destination was one of the other two throwboxes chosen at random. The smart phones were used only to forward bundles generated by the throwboxes.

As the purpose of this experiment was the investigation of the DTN framework's general operation and its performance in WiFi and Bluetooth networks, and since using PRoPHET or its extension requires a fine-tuning of parameters based on multiple runs, an epidemic routing protocol was used.

The experiment ran for approximately three days; the time span between the first device starting and the last device ending the DTN framework was 75.96 hours. Every smart phone and every access point wrote events regarding the network and the physical state of the device into a log file; these log files were merged after the experiment and processed using Matlab. Every event was annotated with a time stamp in milliseconds. Initially, the clocks of all devices were loosely synchronized using the Internet. The logged events include:

- Connection establishment and loss on the BP layer. This corresponds to a node *encounter* and *leave* event, correspondingly.
- Physical WiFi and Bluetooth state: connected access points or devices, MAC addresses, WiFi signal strength.
- Routing-related events: bundle generation, store or forward decision.

- Communication-related events: bundle transmission initiated, transmission complete, transmission failed and bundle received.
- Battery status: remaining power and whether the device was charging.

In the following sections, results from the experiment are presented. The evaluation focuses on the performance of the network under load of DTN bundles and the comparison between WiFi and Bluetooth communication for delay-tolerant networking purposes.

5.1.1 Contact times

This section investigates the contact times, that is, the durations during which two devices have fully established connection on all network layers. This time can be potentially used for the exchange of user data. Every time the connection was established or lost, a corresponding node encounter/leave event was logged. Afterwards, every encounter/leave pair of events resulted in a sample. The contact time was simply the difference of the time stamps. The data collected on all nodes was merged together. Since both events were measured on the same device and contact time is only dependent on the time difference, time synchronization is not an issue here.

Figure 5.2 gives an overview of the contact times between smart phones and access points, which were connected through WiFi. There are a few phones that have been connected for very long to an access point (almost 50 hours), which we assume is due to the lack of movement. When a smart phone remains in the same office during two days and nights, it is very likely that it remains connected to the same access point.

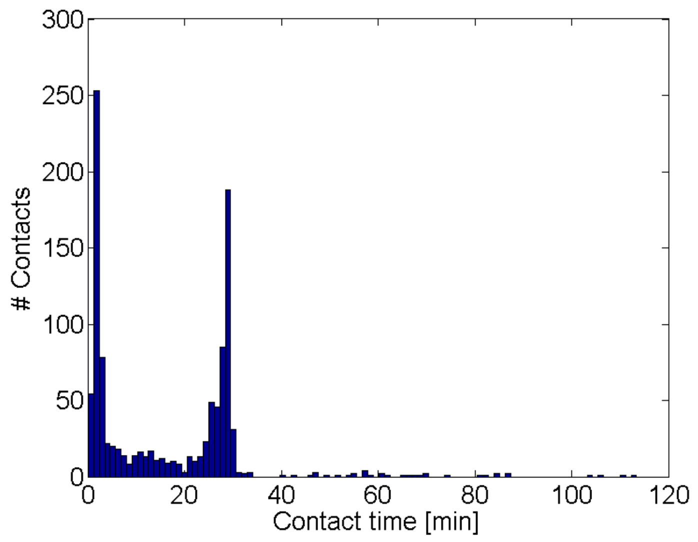


Figure 5.2: Distribution of WiFi contact times; lower 95% of all samples.

The peak around 30 minutes can be explained by the fact that the phones were distributed during a meeting, and they were located in the same place for half an hour. During that time, they have been connected to one access point. The peak around 2-3 minutes could be attributed to people who walk by an access point or carry out short errands, possibly meeting other people in their offices.

Figure 5.3 shows the distribution of contact times between two smart phones, which have been connected via Bluetooth.

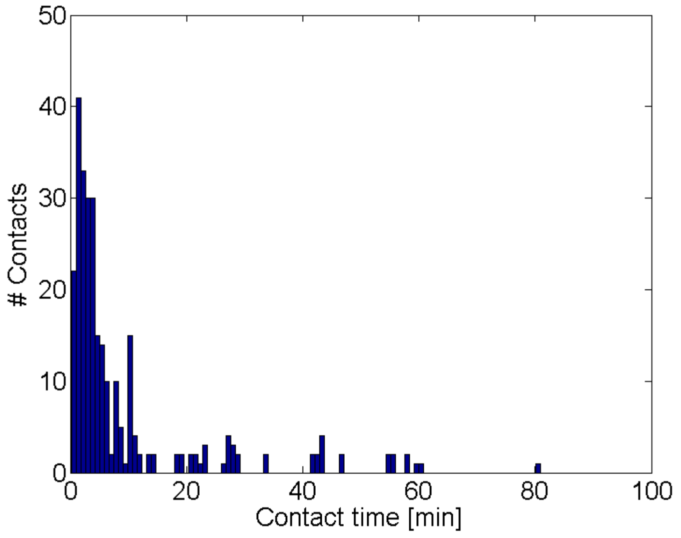


Figure 5.3: Distribution of Bluetooth contact times; lower 90% of all samples.

From the graph it can be noted that unlike WiFi, Bluetooth does not show a peak of contact times around 30 minutes, as one would expect since the phones have been gathered closely in the initial phase. During our tests, we noticed that Bluetooth connections are not very stable when there are many peers around; the number of simultaneous Bluetooth connections is limited. A device that discovers another Bluetooth device tries to establish a connection to it, which drains resources from the Bluetooth adapter and may lead to loss of existing connections. As a result, even though all devices have initially been in Bluetooth range, their mutual connections did not last for more than a few minutes.

The statistics about both WiFi and Bluetooth contact times are gathered in Table 5.1. From the Figures 5.2 and 5.3, it can be seen that there are a few extreme values. As a consequence, the mean values are relatively high and do not represent a typical connection duration. The median values on the other hand are much more realistic.

The maximum contact times are similar for Bluetooth and WiFi, suggesting that the phones that have been connected to access points for almost two days have also been connected among each other. They were probably lying around in adjacent offices. While the longest connections lasted for about two days, it can also be seen that the shortest connections were disrupted only after few seconds. This happens when a person carrying the phone walks by an access point or another device, establishes connection in exactly that moment, but goes out of range only shortly after.

	WiFi	Bluetooth
Mean	49.6 min	101.5 min
Median	18.27 min	4.23 min
Minimum	12 s	5 s
Maximum	49.81 h	44.80 h

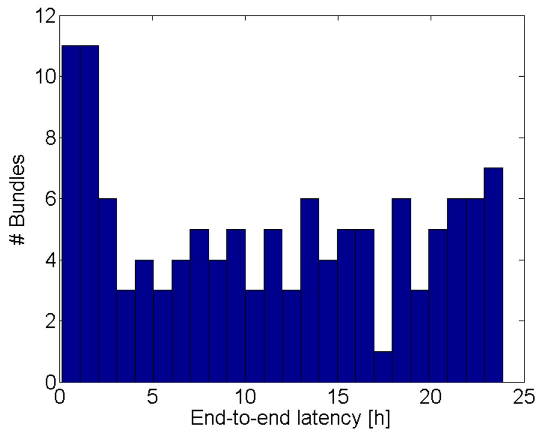
Table 5.1: Comparison of WiFi and Bluetooth contact times

5.1.2 Bundle delivery ratio, latency and hop count

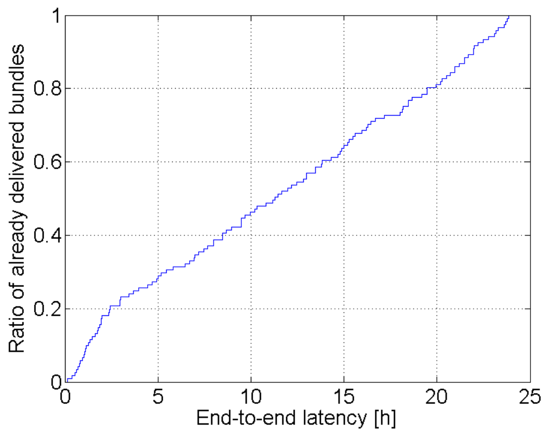
The bundles have been generated on one of three throwboxes and were destined to one of the other two. The bundle expiration time was 24 hours, meaning that after a day, a bundle is discarded, whether it has arrived or not.

In total, 405 bundles were generated during the experiment. 121 bundles have arrived at their destination during the expiration time of a day, which results in a delivery ratio of 29.9%.

Figure 5.4 shows how the delivery latency of each bundle, which corresponds to the difference between the time of arrival at the destination node and the time of generation at the source node. From the histogram, it can be seen that more bundles arrive in the first few hours, while the latency over the rest of the day is roughly uniformly distributed. The cumulative distribution function shows that already 30% of the bundles have arrived after 5 hours, and 80% have arrived after 20 hours. The average latency is 11.35 hours.



(a) Histogram.



(b) Cumulative distribution function.

Figure 5.4: Distribution of bundle latencies.

It is noteworthy that the delivery ratio of 29.9% is biased, because bundles have been generated until the very end of the experiment. With an average latency of 11.35 hours, it is apparent that bundles that were created within the last few hours of the experiment had a very low chance of delivery. 11.35 hours corresponds to 23 bundle generation cycles per throwbox, that is, 69 bundles in total. These 69 bundles are expected to not arrive because they were generated too late. Subtracting them from the total number of bundles, we get a corrected delivery ratio of $121/(405 - 69) = 36.01\%$.

In order to compute the number of hops a bundle takes, its path through the network has to be traced. Since the bundle itself does not memorize the nodes it has seen, this information must be extracted from the log files of multiple nodes. For every distinct bundle (identified by the bundle ID), the transmission and receipt times on different nodes are compared, leading to a list of nodes where the bundle has been located over time. From this list of nodes, only those are considered that form a path from the source to the destination throwbox. The number of nodes on this path is then the hop count. Source and destination node are included in the count, therefore the minimum hop count for a bundle is three, which represents the case where a device directly carries a bundle from one throwbox to the other. In the case where a bundle takes multiple paths on its way through the network, only the path with the shortest latency is considered, i.e. the one where the bundle arrived at the earliest time at its destination.

Of the 121 successfully delivered bundles, 119 have used 3 hops (one intermediate smart phone), while 2 have used 4 hops (two intermediate smart phones). Thus, the vast majority has been delivered directly over WiFi.

A reason for this large difference could be found in the placement of the access points, which did not leave a lot of space in between uncovered. As a result, there is a high chance that smart phones are in range of a WiFi access point. Or, more precisely, when a smart phone is close to another smart phone, it is very likely that it is also close to an access point, and in that case the latter would be preferred. Important factors are the considerably higher range and throughput of WiFi compared to Bluetooth, which encourage network traffic to flow via WiFi.

5.1.3 Bundle transmission statistics

In order to recognize the bottlenecks in the network, observing where bundle transmissions succeed and where they fail can give valuable insights. Every link type (WiFi and Bluetooth) is evaluated separately with respect to the number of successful transmissions and receipts as well as the number of failed transmission attempts. Both the number of transmitted bundles and the data size in MiB are compared.

Figure 5.5 represents the transmission and receipt statistics for WiFi. The three access points, marked with the “ap” prefix, have each transmitted roughly 150 bundles of 400 MiB total size. The smart phones, prefixed with “nexus”, show greater difference in their statistics: while some transmit almost nothing over WiFi, others are transmitting around 50 bundles of 200 MiB total size each. These differences can emerge when some smart phones are moved around more often and therefore act more actively as carriers than others. For example, the phones “nexus3” and “nexus8” received much more bundles than they transmitted, which leads to the conclusion that they have been sitting around in range of a WiFi access point for a long time, receiving a lot of bundles without delivering them to other access points. On the other hand, “nexus4” has received only few bundles, which can be the result of being located outside the covered WiFi range for most of the experiment’s duration.

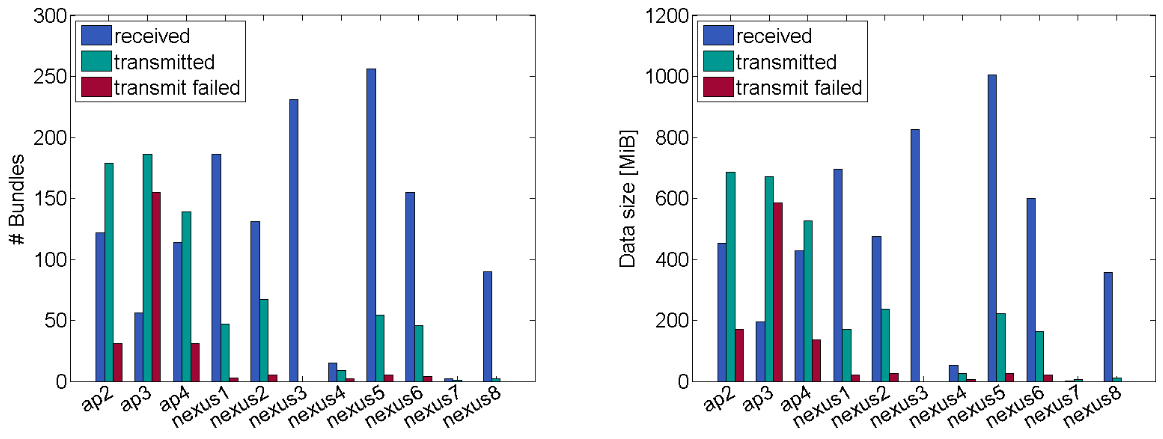


Figure 5.5: WiFi transmission statistics: number of bundles and data size.

In Figure 5.6, the transmission and receipt statistics for Bluetooth are plotted.

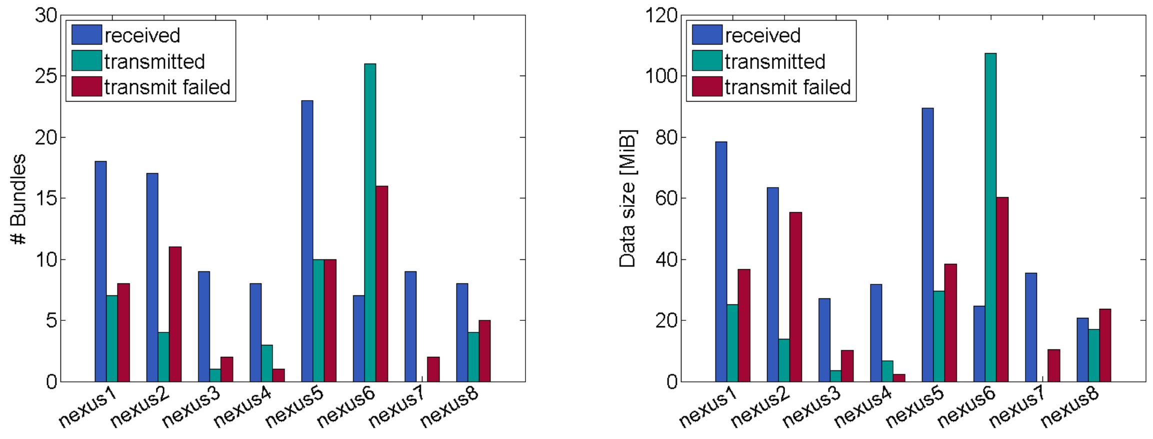


Figure 5.6: Bluetooth transmission statistics: number of bundles and data size.

The diagram plotting the number of bundles and the one plotting the amount of data in bytes follow a very similar trend, which indicates that there are no significant differences between bundles of smaller and larger size.

It can be observed that peers receive much more data than they transmit, while the throwboxes experience the opposite. Throwboxes act as bundle generators and intermediate nodes, while peers are pure carriers, thus throwboxes typically hold more bundles than the peers. Since every contact opportunity is used to transmit the bundles, throwboxes transmit more data than peers, which can also be seen by comparing the absolute heights of the bars. Additionally, peers must first move to another throwbox before they can transmit the data, while throwboxes simply transmit to all devices in range – for bundles that they generate themselves, they can be sure that the peers have not already seen them.

During the experiment, we noticed that the throwbox “ap3” was suffering from issues related to the WiFi connection; it often happened that a smart phone has successfully associated to the access point, but was then disconnected after a few seconds. Since bundle transmission happens directly after connection establishment, a lot of transmissions were aborted due to the early disconnection, which explains why the failure ratio for throwbox “ap3” is considerably higher than for “ap1” and “ap2”.

5.1.4 Node throughput

For every successful transmission of a bundle, the throughput was measured. The throughput is computed as the ratio of transmitted bundle size and transmission time. The bundle size was measured as the number of bytes transmitted in the payload block of every bundle. That is, it excludes the size of protocol headers for BP, as well as those for underlying protocols such as TCP or IP. As a result, the throughput expresses how much user data (payload) can be transmitted via DTN bundles per time. For the time, two time points are recorded: once immediately before the bundle’s byte stream is written to the socket, and once immediately after the socket’s blocking write function returns. As the underlying transport protocols TCP and RFCOMM are reliable, we know that data has arrived by the time the socket finishes writing. This also allows to measure both time points on the same device, which avoids the need for time synchronization. The time and bundle size measurements of all nodes were merged and then separated by the network type (WiFi or Bluetooth).

WiFi throughput

To measure WiFi throughput, the data of 1588 successful transmissions was recorded. The mean value computed across all throughputs is 1.82 MiB/s. However, closer investigation of all obtained throughputs showed that four measured times are zero and several are very close to zero, leading to unrealistically high results. The ten highest throughputs with non-zero time are listed in Table 5.2.

Throughput [MiB/s]	449.75	430.27	387.14	202.91	105.63
Bundle size [B]	471594	902343	811892	1063821	1772235
Transmission time [ms]	1	2	2	5	16
Throughput [MiB/s]	69.33	23.10	23.10	23.10	15.26
Bundle size [B]	218095	48437	48437	48437	48003
Transmission time [ms]	3	2	2	2	3

Table 5.2: Ten highest WiFi throughput measurements

A possible reason for this distortion may be the limited precision of system clocks. Even when measurements with time zero are not considered, there are still some values that are significantly higher than the rest. Since these extreme values affect the average throughput considerably, leaving them as is would distort the results that will later be used for the simulation. Their influence on the average WiFi throughput is observed and displayed in Figure 5.7.

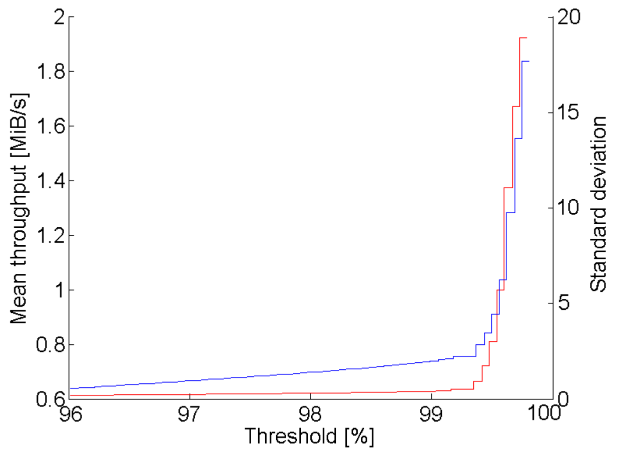


Figure 5.7: Influence of the extreme values on throughput mean and standard deviation. The X axis represents the percentile of the data, above which samples are excluded. The Y axes represent the resulting mean (blue) and standard deviation (red), depending on the amount of included data.

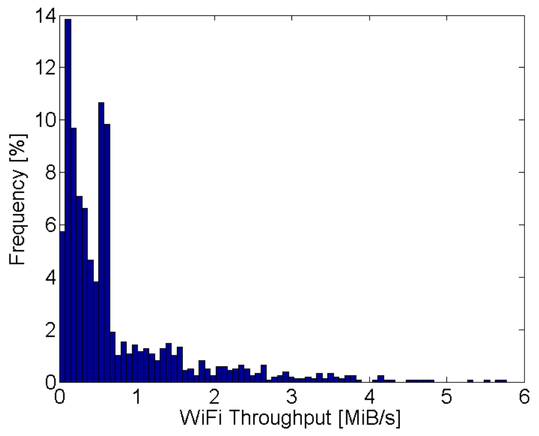
Until 99.15%, both mean and standard deviation are increasing at a constant rate. Above 99.15%, every further sample has a significant influence on the mean and standard deviation, expressed by the large steps. From a semantic standpoint, even the IEEE 802.11n standard supported by the Galaxy Nexus phones does not allow higher throughputs than 600 Mbit/s, which corresponds to roughly 72 MiB/s. This threshold is still far below some measurements. We therefore decided to omit these measurement artifacts in further evaluation.

Using only the samples below the percentile of 99.15%, the WiFi throughput's mean value is 0.71 MiB/s, which is significantly smaller than the previous 1.82 MiB/s. The throughputs are distributed according to Figure 5.8:

Bluetooth throughput

For Bluetooth, only 135 successful transmissions were measured. There have not been any problems with unrealistic extreme values for the Bluetooth transmission. The mean of all Bluetooth throughputs is 64.2 KiB/s.

Figure 5.8 shows the distribution of throughputs for both WiFi and Bluetooth. Both follow a similar structure, although the WiFi histogram is more smooth because of more available samples. Both histograms



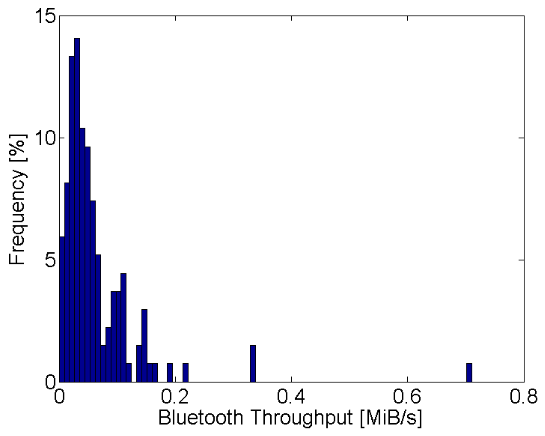


Figure 5.8: Distribution of throughputs for WiFi and Bluetooth. For WiFi, only data below the percentile 99.15% is plotted.

resemble a Gaussian shaped curve: most of the measurements are closely gathered, the number of samples decreases quickly for higher or lower throughputs. Slightly varying throughputs depending on the device can lead to small spikes in the curve.

5.1.5 Power consumption

The power consumption of the JDTN-based framework on the Android phones was measured, by logging the battery state whenever it changed. Three scenarios were analyzed: the experiment and two separate measurements for comparison. These are the three scenarios:

1. Running a simple application that does nothing more than logging the battery state.
2. JDTN service running, but WiFi and Bluetooth networks offline.
3. The experiment itself. Measurements during periods where smart phones were not recharged are considered.

Each of the scenarios was measured on 3 different devices for 24 hours. Figure 5.9 shows the battery drain for every scenario, where the three samples per scenario are averaged. The graphs are shifted so they start in the same origin.

As can be seen from the figure, simply running JDTN has no big effect on the power consumption. However, enabling Bluetooth and WiFi networks and working under actual network load (bundle transmission) drains the battery to a much higher extent.

The difference between initial and eventual battery status in percent is written down in Table 5.3.

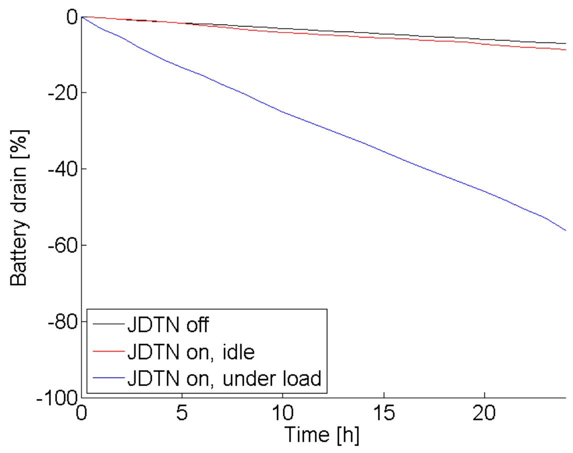


Figure 5.9: Battery drain during 24 hours.

1. JDTN off	7%
2. JDTN on, idle	8.8%
4. JDTN on, under load	56.2%

Table 5.3: Battery drain during 24 hours.

5.2 Simulation

In addition to the real-world experiment, a simulation software for delay-tolerant networks was developed. The software handles the movement of nodes, connections between nodes in wireless range, and bundle routing. Simulation was chosen because it allows a quick analysis of different DTN scenarios and a detailed investigation of the influence of certain parameters on the network. This is of great advantage for the evaluation of the routing protocol.

Two setups are considered: a random model that has been enhanced by introducing community-based mobility, and real-world mobility traces based on GPS-annotated data. Subsection 5.2.1 gives a brief overview about the implementation and modeling. It discusses the two mobility models as well as a list of parameters that have been selected for the simulation. The remaining subsections evaluate the two mobility models.

5.2.1 Modeling assumptions

The simulation is written in C++11. It has been modeled to fit our DTN scenario as closely as possible.

Its basic components are nodes and links. A node represents either a mobile peer or a stationary throwbox. Like in the experiment, peers communicate with each other via Bluetooth, and throwboxes communicate with peers via WiFi. The ranges and throughputs are determined according to the experience gained in the experiment, their values are shown in Table 5.4. When a node enters the wireless range of another node, it waits some time until a link is established. The link represents the connection between two nodes and carries the exchanged packets. When the nodes leave their wireless range, the link breaks.

After an initial metadata exchange, nodes are ready to forward bundles. Bundles are generated by throwboxes and destined to other throwboxes, in analogy to the experiment. The routing and forwarding algorithms correspond to those described in Section 3.2.

Various processing delays are modeled using *delayed queues*. A delayed queue is a FIFO queue constrained by a throughput and a latency. When packets are pushed to the queue, they can only be extracted after some time, depending on their size and the queue properties. Given packet size S_P, queue latency L, queue throughput C, the packet delay D_P is computed as follows:

$$D_P = L + \frac{S_P}{C}. \tag{5.1}$$

If there are multiple packets in the queue, their respective delays add up. Queues are used in the following places:

- **Forwarding queue.** This queue models local processing delays. It has a fixed latency but infinite throughput.
- **Outbound queues.** There is one outbound queue for Bluetooth and WiFi, respectively. Both model the transmission of the packet based on the throughputs measured in the experiment.
- **Link queue.** This queue models packets in flight, after they have left the outbound queue. It uses only a very small latency.

Figure 5.10 gives an overview of the interaction between queues. A bundle that is considered for forwarding is first inserted into the forwarding queue. Then, depending on the type of link, it is moved to either the Bluetooth or the WiFi outbound queue, which model the respective wireless adapters. When the whole bundle has been processed by the outbound queue (i.e. transmission of the last byte has started), it is inserted to the link queue. At the remote node, an inbound bundle is dispatched and – if it is not directly forwarded, delivered or discarded – stored locally, so the process repeats.

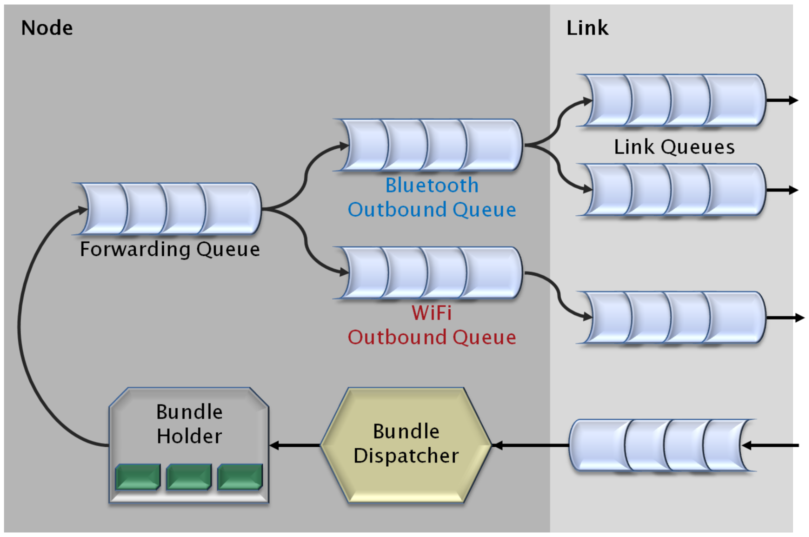


Figure 5.10: Model of queues used in the simulation. The left part contains the forwarding and outbound queues which belong to the node, the right part models the link with its link queues (one per node-node connection).

Storage is limited on every node, the corresponding capacities are listed in Table 5.4. The *storage allocation* $\sigma(t)$ of a node at time t is the sum of the sizes of all bundles the node holds at that time. The *maximum storage allocation* σ_{max} and the *average storage allocation* $\bar{\sigma}$ are obtained as follows, where T is the total simulation duration:

$$\begin{aligned} \sigma_{max} &= \max_{t\in[0,T]} \sigma(t) \\ \bar{\sigma} &= \frac{1}{T}\cdot\int_0^T \sigma(t)\,dt \end{aligned} \tag{5.2}$$

Storage allocation as a global property of the network is obtained by computing the mean over the storage allocations of all nodes.

Parameter selection

Bluetooth and WiFi throughputs, ranges and connect times are based on the experience from the real-world experiment. For the measured throughputs, the mean value was applied to the simulation. The parameter values are listed in Table 5.4.

Parameter	Value
WiFi throughput	1.82 MiB/s
WiFi range	30 m
WiFi connect time	5 s
Bluetooth throughput	64.2 KiB/s
Bluetooth range	12 m
Bluetooth connect time	7 s
Minimal peer speed	0 m/s
Maximal peer speed	0.7 m/s
Refresh constant P_{init}	0.001
Age constant γ	0.9993
Transitive constant β	0.2
Node type factor for peers C_{peer}	0.5
Bundle duplication limit Λ	3
Popularity update interval T_U	5 s
Contact time window T_W	60 s
Minimal bundle size	1 KiB
Maximal bundle size	8 MiB
Storage capacity for throwboxes	40 GiB
Storage capacity for peers	1 GiB

Table 5.4: Summary of selected parameter values.

Since the GPS-based mobility model was simulated during three weeks and the random mobility model only during two days, some parameters related to the frequency of bundles have been chosen differently in each mobility model. The parameters and their corresponding values for each of the two mobility models are listed in Table 5.5. The *throwbox cooldown time* denotes the time span at the end of the simulation, during which no more bundles are generated. This parameter leaves bundles some time to be delivered.

Random mobility model

To have a realistic setting that can be used for the evaluation of routing protocols, an approach very similar to the one in the PRoPHET proposal was chosen [34]. The world is divided into seven *communities* that

Parameter	Value in random model	Value in GPS model
Bundle generation interval	0.5 h	3 h
Bundle lifetime	24 h	72 h
Throwbox cooldown time	6 h	24 h

Table 5.5: Specific parameter choices for each mobility model.

are aligned in a star-shaped topology. A community denotes a circular area to which nodes can be assigned. The community in the middle, called *midtown*, is supposed to represent a village or town center where a lot of people meet. Six communities surround the midtown, they represent quarters where people live. In addition, so-called *gathering points* are inserted, which intend to model people that decelerate because they know someone, they have a deeper look at a store, they move in a crowd or similar. Gathering points are circles that slow down every node that traverses them by a constant factor (currently 0.7), thus increasing the likelihood of DTN contacts.

To analyze the performance of the routing protocol developed during the Master Thesis, the world was populated with 50 peers and three throwboxes that were placed in the centers of randomly chosen home communities. The simulation was run for two days with a time step of one second. The network topology is illustrated in Figure 5.11.

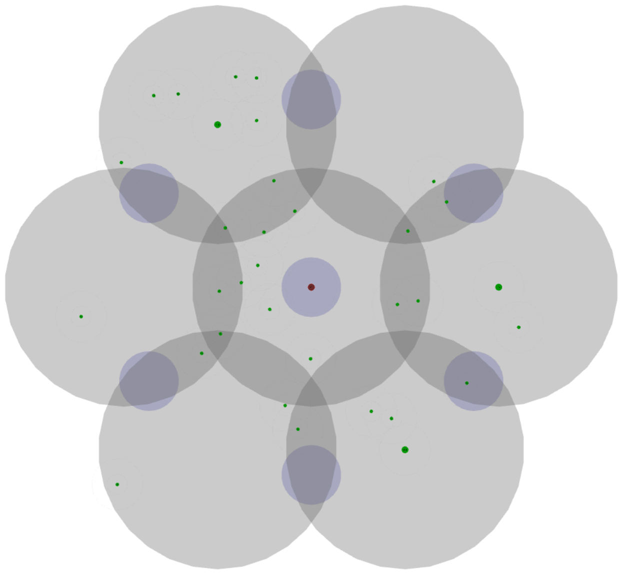


Figure 5.11: Network topology with communities (big gray circles) and gathering points (purple circles). The green dots represent nodes, where the small ones are peers and the larger ones throwboxes. The red throwbox in the middle is initially not placed; it will be in Subsection 5.2.5.

Initially, every node is distributed randomly to one of the six surrounding communities. The community where a node starts is said to be its *home community*.

The mobility model allows nodes to be in one of two states:

1. **Stay.** A node remains in the community where it is currently located. It moves around randomly; once it approaches the edge of the community, it is directed inwards again, so that it cannot leave the community. After 5 minutes, the state changes automatically into *commute*.

2. **Commute.** A node chooses any community except its current one by random. When a node is in its home community, the most likely choice is the midtown; otherwise the most likely choice is the home community. With a small probability, other communities are visited.

   Once a community is chosen, a point is randomly selected within that community (using a uniform distribution over the area). This point is the node's destination. The node travels in a straight line to the destination at its current velocity; once it reaches it, a transition to the *stay* state is performed.

The random movement of a node in *stay* state is implemented using the *wander* steering behavior introduced by Craig W. Reynolds [41, 42]. Steering behaviors are a very popular approach to model artificial intelligence characters in games. For the wander behavior, a circle is located in front of the character (or network node in our case). A point on the circle's boundary moves by a random offset in every time step. The vector from the character's position to the point on the circle represents the new velocity vector. The advantage of this approach is that state of the curvature is kept over multiple frames: a character taking a curve will not completely change its direction in every time step, thus resulting in more realistic movement behavior. The wander steering behavior is visualized in Figure 5.12.

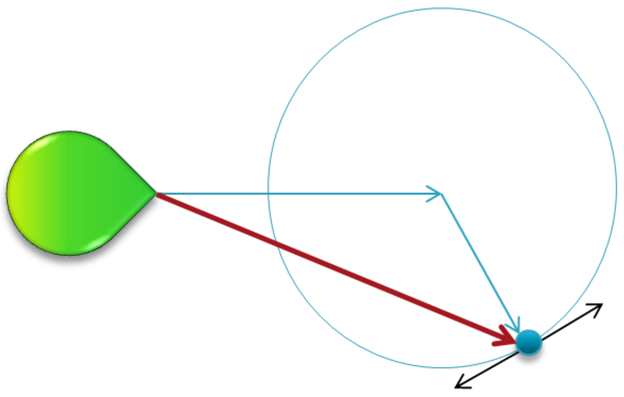


Figure 5.12: Wander steering behavior. In front of the green node, a circle is located, on which a point moves. The red vector is the direction of the new node velocity.

The magnitude of the node velocity (speed) is constrained to specified bounds (currently [0, 0.7] m/s). Initially, the speed is chosen uniformly in this range. During the simulation, it is offset by a small random acceleration in every time step. The velocity vector resulting from the wander behavior is not immediately applied, but set to the length of the current speed.

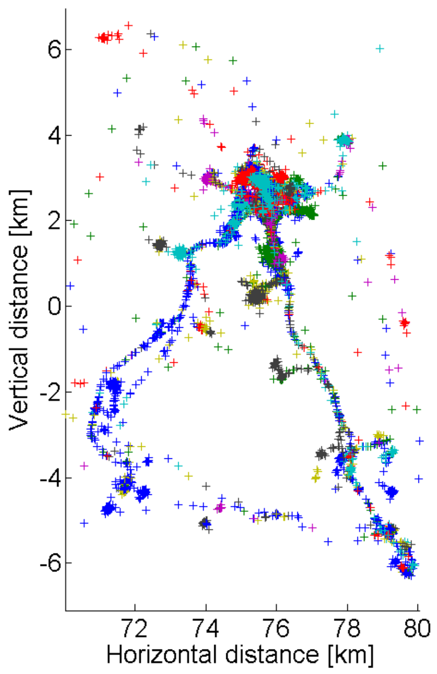
GPS-based mobility traces

To simulate using real-world data, the CenceMe dataset from the Darthmouth university was used [43]. The data was gathered by students and employees carrying smart phones. A total of 18 mobile DTN nodes were used. The log files in the data set contained information concerning acceleration, GPS positions or labeled activities; only the logged GPS coordinates were of interest for this work. By integrating the location data into our DTN simulation, it is possible to observe the network behavior in a real-world mobility model.

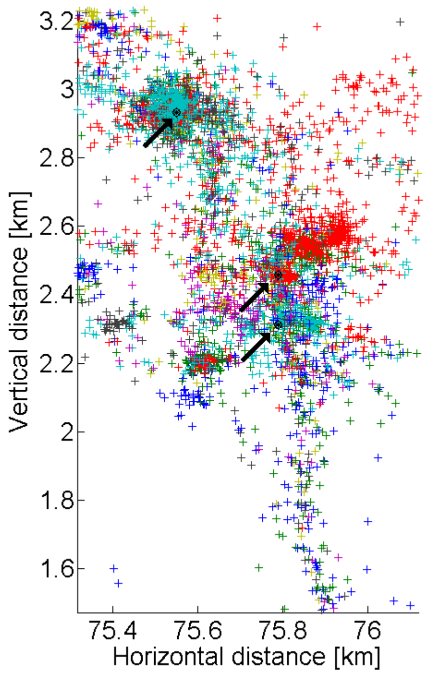
The GPS samples were converted from spherical Earth coordinates (longitude λ and latitude ϕ in radians, altitude a in meters over sea) to a Cartesian 2D coordinate system in meters (x towards east, y towars north). This was done by projecting each position sample onto a plane tangential to the Earth. This relation is described in Equation (5.3), where R_{earth} is the average Earth radius. The intersection point (λ_0, ϕ_0, 0) of plane and Earth sphere was chosen as the mean of all GPS position samples of all times.

$$\begin{aligned} x &= (R_{earth} + a) \cdot sin(\lambda - \lambda_0) \cdot cos(\phi - \phi_0) \\ y &= (R_{earth} + a) \cdot sin(\phi - \phi_0) \end{aligned} \tag{5.3}$$

Figure 5.13 displays the position samples in 2D coordinates from a top view.



(a) Top view onto the traces.



(b) Closer look at the upper region, with throwbox positions.

Figure 5.13: Position samples from the GPS trace dataset. Different colors refer to different nodes.

Every GPS sample was annotated with a time stamp. From it, a time difference relative to the beginning of the experiment was computed. The position of each node at a time t was either the Cartesian position inferred from the GPS sample, when there is exactly a sample for time t; otherwise, the position is linearly interpolated between two samples.

The total duration of the experiment was 501 hours, 15 minutes and 40 seconds, which corresponds to almost 3 weeks. This time is measured from the very first GPS sample to the very last one, on any device. Some devices have been started later or stopped earlier, thus not all nodes are active during the whole duration.

Initially, throwboxes were placed by a K-means clustering algorithm across all position samples that was supposed to find points where a lot of people gather. However, since it is difficult to strictly separate the map into clusters, the clustering centroids were shifted to locations that nobody ever visited. Instead, three throwboxes were placed manually in locations of high density. As can be seen from Figure 5.13, every throwbox is surrounded by many nodes. Two throwboxes are relatively close, but are frequented by different nodes (hence the red and teal clusters).

Most of the parameters for the random simulation have been adopted by the GPS-trace simulation. Since the whole data collection spans a longer duration and the graphical visualization has shown that some nodes stay longer at their place before they eventually move to a different location, some time-related parameters have been increased. The bundle expiration time was set to 72 hours instead of 24 hours. The throwbox cooldown was changed from 6 to 24 hours.

5.2.2 Contact times

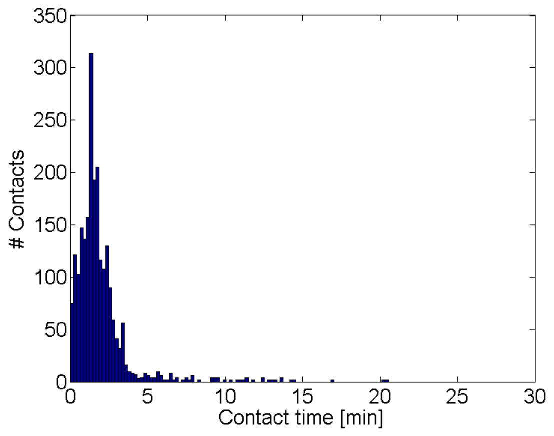
Both mobility models have been analyzed with respect to contact times. Figure 5.14 compares times during which two nodes have been connected over a WiFi link for both mobility models. The figures suggest that in the GPS model, WiFi contact times are generally shorter than in the random model.

The comparison of Bluetooth contact times is visualized in Figure 5.15. The same tendency as for WiFi is visible: contact times are generally shorter.

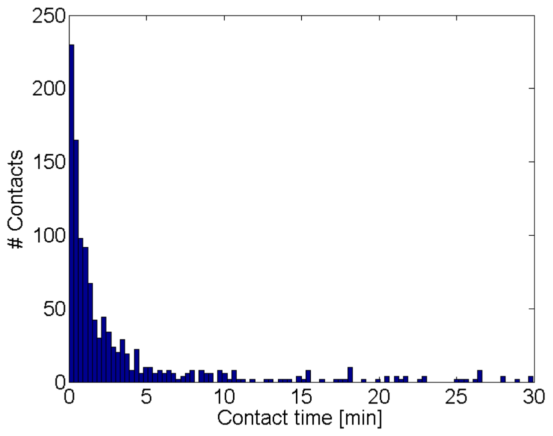
Table 5.6 lists quantitative measurements of the contact times for both mobility models.

	WiFi		Bluetooth	
	Random	GPS	Random	GPS
# Contacts	2267	1274	26123	1925
Mean duration	2.10 min	32.28 min	29 s	3.21 min
Median duration	1.53 min	1.45 min	23 s	44 s
Minimum duration	1 s	1 s	1 s	1 s
Maximum duration	27.87 min	75.02 h	8.20 min	10.89 h

Table 5.6: Comparison of WiFi and Bluetooth contact times.

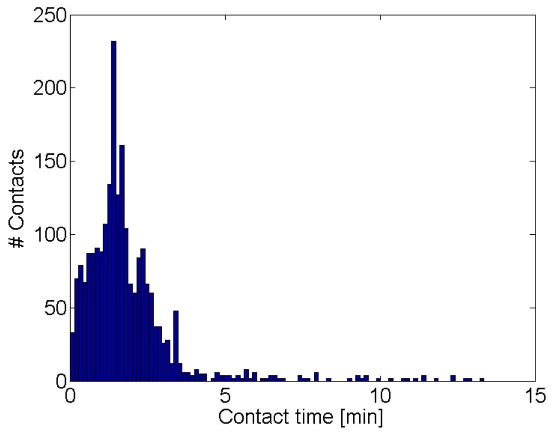


(a) Lower 99.5% of data for random model.

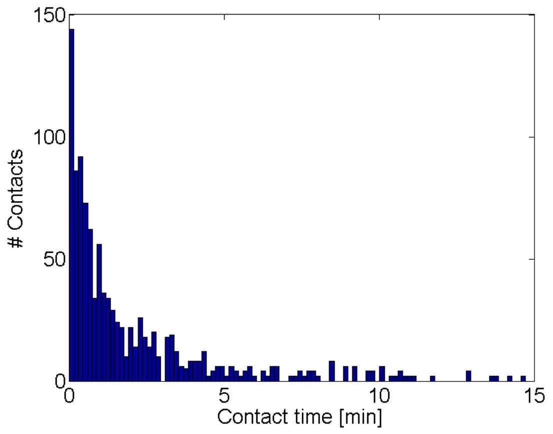


(b) Lower 90% of data for GPS model.

Figure 5.14: Distribution of WiFi contact times.



(a) Lower 99% of data for random model.



(b) Lower 93% of data for GPS model.

Figure 5.15: Distribution of Bluetooth contact times.

The random mobility model shows considerably more contacts than the GPS model, even though the simulation lasts only for two days instead of almost three weeks. This is a direct result of the much denser setting in the random model (crowded village scenario).

Maximum contact times are much shorter in the random model, because the nodes stay only for five minutes within a community; after that they start to travel to another community. The GPS traces have no such constraints, and the numbers indicate that some have stayed very long at the same place. The maximum WiFi contact time of more than three days could be a result of leaving the tracked device at the same workplace.

The minimum durations amount to one second because this is the time step of the simulation; no shorter connections are possible. Such short contact times can occur when nodes pass by another node and have just enough time to establish the connection, but not to exchange any data.

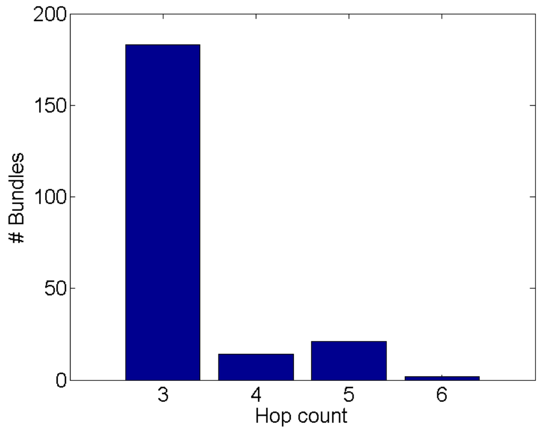
Mean contact times vary a lot; because there are extreme values that have a huge influence on the mean, the median was also computed. In this case, the median is expected to give a more precise impression of a typical contact time. The median contact times are in the same order of magnitude in both mobility models, for both WiFi and Bluetooth. In the random mobility model, the Bluetooth median contact time is roughly half the duration of the GPS model.

In conclusion, it can be noted that the random mobility model offers more contact opportunities, however it does not represent long-lasting contacts. Median contact times are similar in both mobility models.

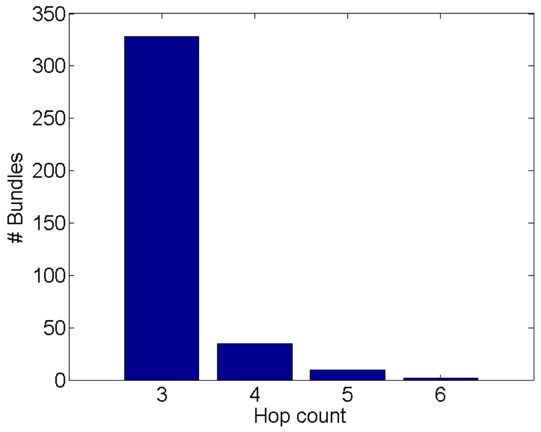
5.2.3 Bundle delivery ratio, latency and hop count

In the GPS model, three throwboxes have generated a total of 477 bundles in 3-hour-intervals. 375 of the generated bundles have arrived at their destination, which corresponds to a delivery ratio of 78.62%. In the random model, three throwboxes generated 243 bundles, one every half an hour. 220 bundles have been delivered, yielding a delivery ratio of 90.53%.

The average hop counts are 3.28 (random mobility) and 3.16 (GPS mobility). Figure 5.16, which shows the distribution of hop counts, confirms that the majority of bundles uses 3 hops, that is, one intermediate hop between source and destination throwboxes. These low numbers indicate that most of the bundles have been delivered directly by one carrier, without traveling through multiple hops. So, they confirm the results obtained in the real-world experiment (Subsection 5.1.2).



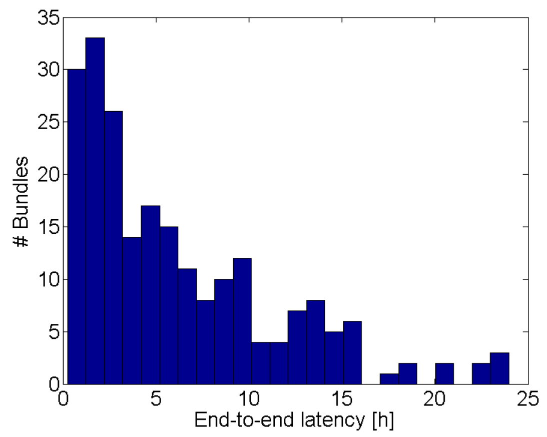
(a) Hop count distribution for random model. The mean is 3.28.



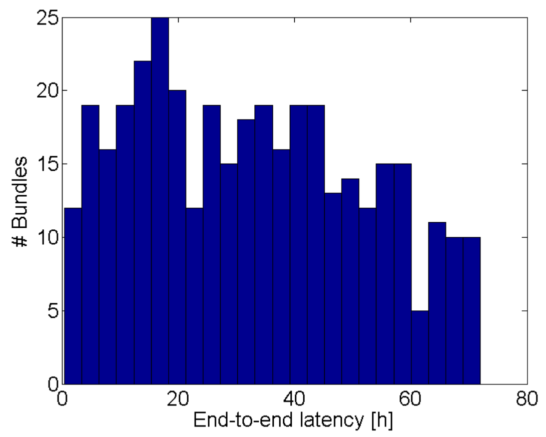
(b) Hop count distribution for GPS model. The mean is 3.16.

Figure 5.16: Distribution of hop counts

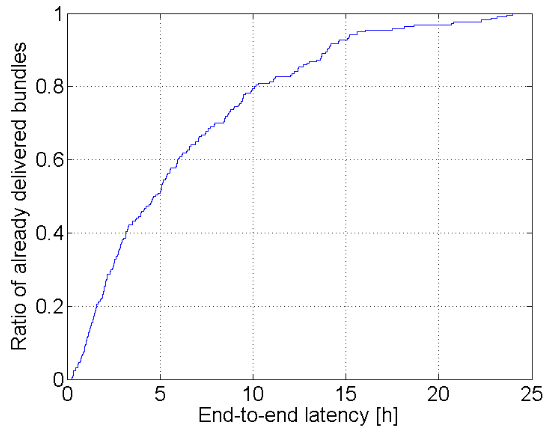
Figure 5.17 shows the distribution of end-to-end latencies for both mobility models. In the random model, the histogram shows a rapid decrease in the number of bundles as the latency increases; most bundles arrive during the first few hours. This differs from the GPS model, where the bundles are more or less uniformly distributed over the whole range of latencies. Although the GPS model also shows a slight decrease in the number of bundles as latency increases, this effect is far less prominent than in the random model.



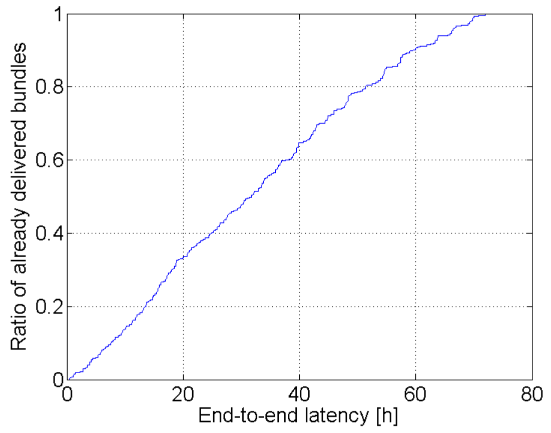
(a) Latency histogram for random model.



(b) Latency histogram for GPS model.



(c) Latency CDF for random model.



(d) Latency CDF for GPS model.

Figure 5.17: Distribution of bundle latencies.

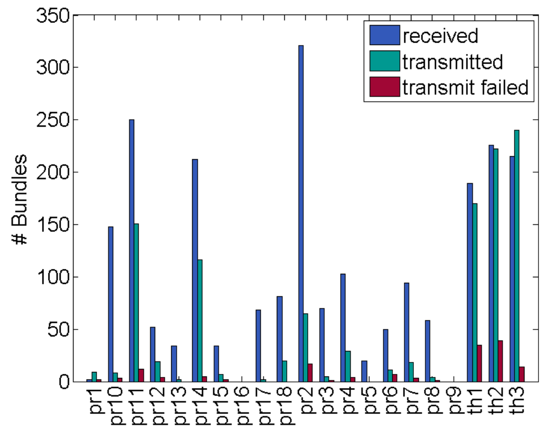
These effects are closely related to the bundle lifetime. The random model does not exploit the lifetime, as most of the bundles have already arrived by the time they would start to expire. In the GPS model however, the almost linear cumulative distribution function indicates that the trend continues after the lifetime of 72 hours. If the lifetime exceeded 72 hours, this would most probably lead to a notable increase in the number of delivered bundles.

A consequence of these observations is that metrics such as latency can strongly vary between different scenarios, and that they can inherently affect the design of the delay-tolerant network. With respect to latency in particular, the bundle lifetime must be chosen to match the characteristics of the underlying scenario as closely as possible. While a too short lifetime leads to the loss of bundles as seen here, a too long lifetime increases the storage requirements in dense settings – possibly up to the point where the network is not operational anymore, as new bundles are discarded because of storage constraints.

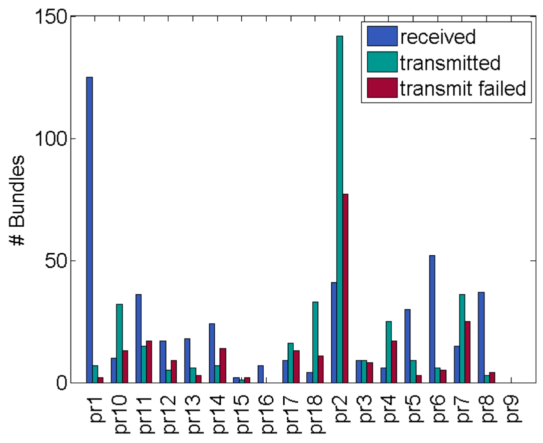
5.2.4 Bundle transmission statistics

The statistics of successful and failed bundle transmissions are not discussed for the random mobility model, because every peer starts with the same initial conditions and underlies the same constraints. As nodes cannot be meaningfully differentiated, no conclusions can be drawn based on behavior or environmental factors specific to a node. The only useful information could be taken from comparing peers with throwboxes, however there have not been results that are not also available in the GPS model. As such, evaluation is focused on the latter.

In Figure 5.18, the transmission and receipt statistics of the GPS model for both WiFi and Bluetooth links are plotted.



(a) Number of bundles transmitted via WiFi.



(b) Number of bundles transmitted via Bluetooth.

Figure 5.18: Transmissions and receipts for both WiFi and Bluetooth.

A substantial difference between the number of bundles that nodes attempt to transmit is recognizable. For example, peer "pr2" transmits almost four times as many bundles as other nodes via Bluetooth; this node hence plays an important role as a forwarder in the network. Linked to that observation is the fact that node "pr2" has received the highest number of bundles via WiFi from throwboxes. On the other side, some nodes do not interact with the network at all ("pr9"), others play a minor role ("pr15", "pr16"). Node "pr1" receives the most bundles via Bluetooth, however it has almost no opportunities to forward them, which suggests that it is often located in a dead-end path of the network (i.e. distant from most nodes, and only surrounded by nodes that have already seen the bundle).

The three throwboxes “th1”, “th2” and “th3” show very similar results with respect to the number of transmitted and received bundles. This suggests that they have been placed in spots that contribute equally to the network.

For Bluetooth transmission, the ratio of failure is much higher than for WiFi, while the absolute number of exchanged bundles is lower. This confirms the results obtained in the real-world experiment.

5.2.5 Effect of throwbox placement

This section investigates the influence of throwboxes on the network performance. For the random mobility model, two simulations were run, where one contained a throwbox in the center of the midtown community, and the other did not. The purpose of this comparison is to see whether throwboxes placed in crowded locations are able to improve the number of contact opportunities, and to which extent.

The results are shown in Table 5.7. After placing the throwbox, the delivery ratio has increased by 7%.

	Without throwbox	With throwbox
Delivery ratio	90.53%	97.53%
Average latency	6.23 h	2.85 h
Average hop count	3.28	3.76
# WiFi transmissions	2267	5698
# Bluetooth transmissions	26123	23426

Table 5.7: Effect of placing a throwbox in the center

The latency with the throwbox is less than half of the original latency. The hop count has slightly increased, which is most probably a result of the new intermediate hop on the center throwbox. When looking at the number of transmissions, the number of bundles transmitted via WiFi has grown by a factor of 2.5 because of the new contact opportunity. The number of bundles transmitted via Bluetooth has slightly decreased, probably because contacts that would have exchanged data via Bluetooth in the center now prefer the faster and wider-range WiFi communication.

According to those values, a throwbox placed in a very crowded region can have a largely positive effect on the network performance.

5.2.6 Evaluation of the routing protocol

This section analyzes the performance of the routing protocol that was developed during this work. A comparison with existing routing protocols such as epidemic routing and PRoPHET is made.

The GPS-based mobility traces have shown to provide very few multi-hop transmissions, and thus do not leave many options for a routing protocol to decide where to forward bundles. In order to benefit from smart forwarding strategies, nodes must experience the situation where not all bundles can be forwarded, and a forwarding priority is assigned. We thus use the random mobility model to evaluate the routing protocol.

Bundle duplication limit

A fundamental parameter in the routing protocol is the bundle duplication limit Λ, which constrains how many times each node (both throwbox and peer) is allowed to forward each bundle. This parameter is essentially a trade-off between probability of arrival and utilization of network resources. With high values, the routing degrades to epidemic routing. A value of 1 is the other extreme, where the bundle only travels along a single path through the network.

Figure 5.19 visualizes the effects of varying the bundle duplication limit on the delivery ratio and the allocated storage. Also shown are the effects on transmission success ratio for Bluetooth and WiFi.

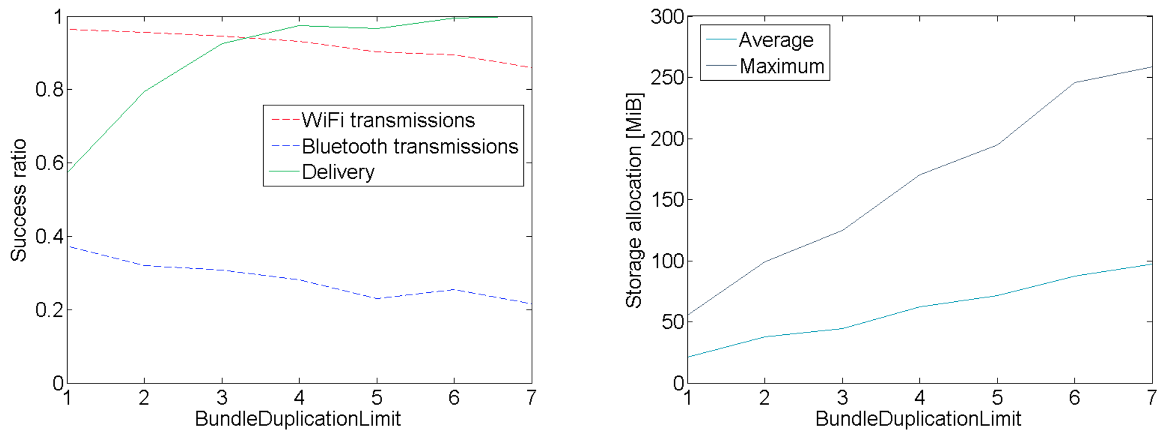


Figure 5.19: Effects of bundle duplication limit on delivery ratio, transmission success and storage allocation.

The delivery ratio raises continuously from 60% to 98% until $\Lambda = 4$, which is expected because a bundle arrives with higher probability when more nodes carry it. Higher values for Λ do not improve the delivery ratio significantly. The success rate of WiFi and Bluetooth transmission on the other hand decrease slightly, but constantly. This can be attributed to the fact that when nodes have more bundles to transmit, it is more likely that the contact time is too short to transmit all of them, as a result of which some will fail. The average and maximum storage allocation increase constantly, as one would expect with more bundles in the network.

Figure 5.20 shows the average latency and hop count of bundles depending on the bundle duplication limit. The average latency decreases by 3.5 hours between $\Lambda = 1$ and 4, which shows that the additional node carriers contribute significantly to a shorter delivery delay. The hop count also decreases until $\Lambda = 1$, but later it doesn't show a clear direction. A possible explanation for increasing hop counts is the decreasing WiFi/Bluetooth transmission success rate.

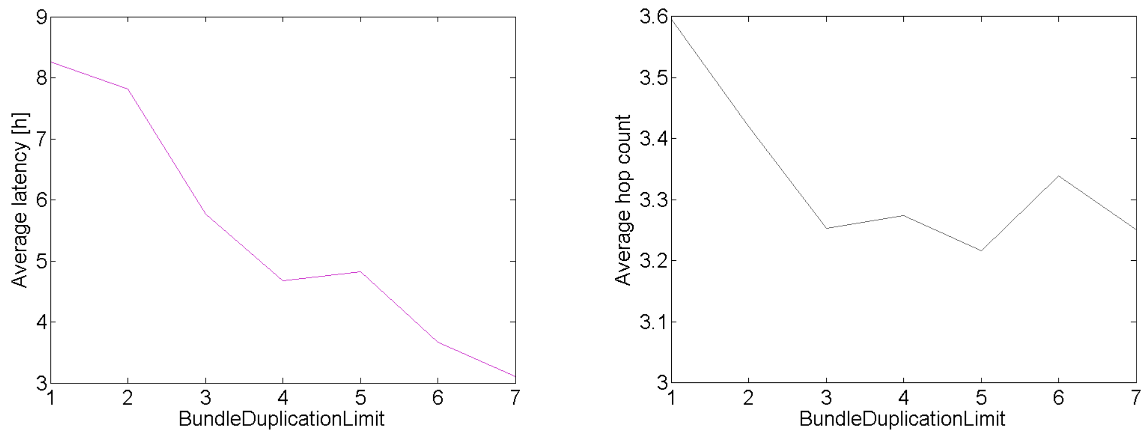


Figure 5.20: Effects of bundle duplication limit on bundle latency and hop count.

According to the figures, a value of $\Lambda = 3$ or $\Lambda = 4$ seems to show a reasonable trade-off between storage allocation and delivery ratio or latency. While the required storage continues to grow with higher values, the ratio of delivered bundles and the latency improve only slightly. This shows that our protocol can achieve almost the performance of epidemic routing with a significantly lower overhead. For further evaluation, the parameter is fixed to $\Lambda = 3$.

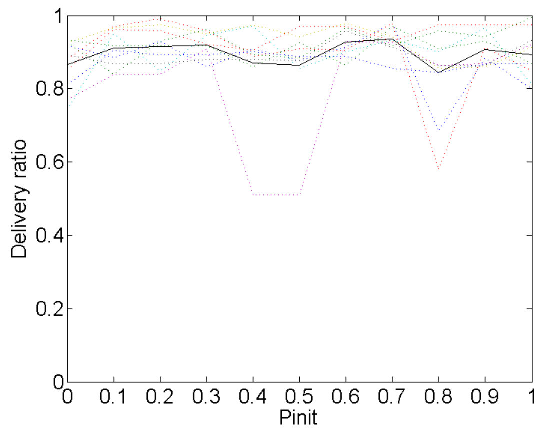
Selection of refreshing, aging and transitivity parameters

Our routing protocol updates popularities using the refresh, age and transitive functions, where the idea originates from PRoPHET. The parameters used by these three functions are P_{init}, γ and β, correspondingly (see Section 3.2).

Finding ranges where these parameters perform well turned out to be a challenging task, since each parameter on its own had no systematic influence on the network performance. Instead, behavior was chaotic: a small change in the parameter led to a completely different outcome. As an example, Figure 5.21 shows the change in delivery ratio and latency when P_{init} is varied. The dotted colored lines represent different seeds of the random number generator; the black line shows the mean value across 10 runs.

There is no clear pattern recognizable. A sweep across a fraction of the possible parameter values with more samples led to similar results; the curve does not smooth out. The same behavior has appeared for γ and β.

It seems like minor changes in the way how node popularities are computed lead to a completely different assignment of popularities, of which the effects are not obvious. One problem is that our routing protocol is also dependent on further parameters: the bundle duplication limit Λ, the node type factor for peers C_{peer}, the popularity update interval T_U and the contact time window T_W. This leads to a very large design



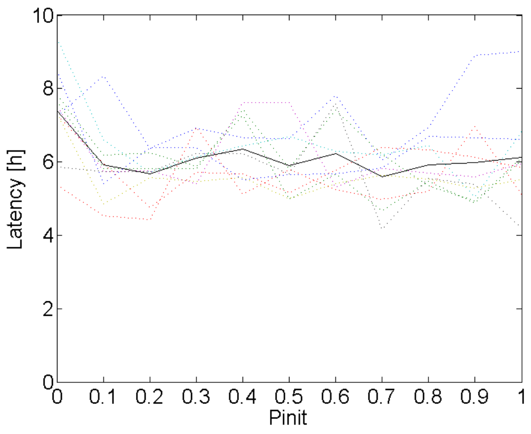


Figure 5.21: Effects of P_{init} on delivery ratio and latency.

space, of which the exploration by means of one-dimensional sweeps does not scale. The original papers on PRoPHET routing [34, 35] do not describe the way how their parameters were chosen, either.

In the simulation, we chose $P_{init} = 0.001$, $\gamma = 0.9993$ and $\beta = 0.2$, because these values lead to a popularity that does not increase too quickly and stay at 1; aging was chosen to be weak enough to not negate the effects of refreshing.

Comparison between different routing protocols

Besides epidemic routing, it is interesting to see how routing protocols behave if the bundle duplication limit is kept constant. This comparison reveals whether some protocols are able to meet smarter forwarding decisions, constrained by the limited number of times a bundle can be forwarded.

Three routing protocols that all use a fixed bundle duplication limit $\Lambda = 3$ are compared:

- The protocol developed during the thesis.
- PRoPHET routing.
- *Oblivious routing*: this protocol does not consider node popularities and treats all nodes equally.

Each of the three protocols has been evaluated in five simulation runs, of which the results were averaged. A throwbox was placed in the midtown, otherwise the original parameters apply. The results are shown for 30 and 50 peers in Tables 5.8 and 5.9, respectively. Choosing two different numbers of peers allows comparison of results in different network densities and excludes effects specific to one density.

The results are very similar for 30 and 50 peers. Our protocol outperforms the oblivious protocol in terms of delivery ratio by 3-4%. Latencies are in range of 61-67% of the oblivious protocol. PRoPHET has even higher delivery ratio (difference of 5%) and lower latency (43-47% of oblivious routing). Both protocols show higher average hop counts compared to oblivious routing, which indicates that longer paths

	Oblivious	Our protocol	PRoPHET
Delivery ratio	94.51%	97.53%	99.69%
Average latency	4.76 h	2.92 h	2.25 h
Average hop count	3.04	3.81	3.94
Average storage allocation	64.0 MiB	135.4 MiB	166.9 MiB
Maximum storage allocation	154.9 MiB	348.8 MiB	426.9 MiB

Table 5.8: Comparison of routing protocols, 30 peers.

	Oblivious	Our protocol	PRoPHET
Delivery ratio	94.63%	98.27%	99.57%
Average latency	4.56 h	3.06 h	1.97 h
Average hop count	3.04	3.81	3.96
Average storage allocation	40.5 MiB	92.9 MiB	124.3 MiB
Maximum storage allocation	99.4 MiB	240.7 MiB	312.4 MiB

Table 5.9: Comparison of routing protocols, 50 peers.

in the network are considered. This can be partially attributed to the transitive property which takes into account indirectly reachable nodes.

The fact that storage allocation is higher is mainly a result of more nodes carrying the bundle. One possible explanation is as follows. Since both PRoPHET and our routing protocol prioritize bundle forwarding, they can prefer nodes that are slower and thus more local to one community to those that frequently visit various communities. These slower nodes typically have fewer contact opportunities, so they carry fewer bundles. Therefore, the chance that they can accept a new bundle is higher because a) the storage limit is not reached as quickly, and b) because of their slowness, the link will not break as fast, making more transmissions successful.

Regarding the comparison of our routing protocol and PRoPHET, we believe that more thorough and systematic investigation of the involved parameters could improve the performance of our protocol.

5.3 Scenario comparison

In this section, the most important results gained from the real-world experiment, the simulation based on random mobility in communities and the simulation based on GPS traces are summarized. Table 5.10 lists measurements in the three different scenarios next to each other for direct comparison.

	Experiment	GPS simulation	Random simulation
Total duration	76 h	501 h	48 h
Bundle generation interval	0.5 h	3 h	0.5 h
Delivery ratio	31.60%	78.62%	90.53%
Average latency	11.35 h	32.46 h	6.23 h
Average hop count	3.07	3.16	3.28
# WiFi transmissions	1139	1274	2267
WiFi transmission success	73.33%	84.56%	92.64%
Mean WiFi contact time	49.64 min	32.28 min	2.10 min
Median WiFi contact time	18.27 min	1.45 min	1.53 min
Minimum WiFi contact time	5 s	1 s	1 s
Maximum WiFi contact time	44.80 h	75.02 h	27.87 min
# Bluetooth transmissions	312	1925	26123
Bluetooth transmission success	46.07%	57.31%	28.23%
Mean Bluetooth contact time	1.69 h	3.22 min	29 s
Median Bluetooth contact time	4.23 min	44 s	23 s
Minimum Bluetooth contact time	12 s	1 s	1 s
Maximum Bluetooth contact time	49.82 h	10.89 h	8.20 min

Table 5.10: Comparison of real-world experiment with two simulations.

Because the three measurements originate from fundamentally different settings, it is not meaningful to directly compare all of the results. For example, a comparison of delivery ratios between the three scenarios is not expressive; it is more reasonable to compare this metric between different adaptations of the same scenario, as we did in the previous sections. End-to-end latency depends largely on the scale and the number of contact opportunities in the network: the GPS simulation where nodes are several kilometers apart imposes the highest latency. The following paragraphs focus on observations that are general enough to be applicable to different settings.

The three scenarios have several commonalities. For example, the hop count is always close to 3, implying that most bundles have been transmitted directly via WiFi. For the random simulation, the hop count is slightly higher, but still very small; with such a large number of Bluetooth transmissions, one could expect that more bundles would use multiple hops via Bluetooth. In all scenarios, the bundles transmitted via Bluetooth do not really contribute to more hops, which we assume has two reasons:

1. Bluetooth plays a minor role compared to WiFi and will be the second choice when both are available. Because of the higher throughput and range of WiFi networks, more bundles can be exchanged via WiFi than Bluetooth in an equal amount of time. This difference is amplified by our routing protocol which prefers throwboxes over peers.

2. The throwboxes have been placed prominently to encourage direct delivery. The result would look different, if peers didn't travel the whole distance between two throwboxes, and communication between peers were essential to the success of delivery.

In the real-world experiment, contact times are rather long, which is a direct consequence of the lack of mobility. The GPS model too contains a few nodes that do not change their location for a very long time, however half of the devices are connected via WiFi for only 1.45 minutes or less. For Bluetooth, the median contact time is 44 and 23 seconds in the simulated scenarios, indicating that half of the contacts between peers have lasted for less than a minute. Given the throughput measured in the experiment, this short contact time does not suffice to transmit bundles with the size of several megabytes.

Chapter 6

Conclusion

In this Master Thesis, a delay-tolerant networking framework for Desktop and Android systems based on the JDTN library was developed. We could show that the framework is operational, and that it it possible to carry bundles from one throwbox to another without any user interaction.

6.1 Throwboxes and routing protocol

This work extends the PRoPHETv2 and PRoPHET+ routing protocols and proposes several improvements to address shortcomings in the computation of node popularities. With respect to throwboxes, a new parameter is introduced to differentiate between node types and to account for the impact of throwboxes on the network. Instead of updating popularities based on encounter events, the metric is updated in regular time intervals.

A simulation using a random mobility model based on communities showed that the proposed routing protocol achieves delivery ratio and latency comparable to epidemic routing while drastically reducing storage allocation on the nodes. By forwarding every bundle at most three to four times per node, a performance close to that of epidemic routing was achieved.

Furthermore, our popularity-based routing protocol outperformed a protocol that duplicates the same number of bundles, but is oblivious to the popularity metric and thus does not distinguish between nodes. This implies that considering throwboxes and contact times leads to an improvement in forwarding decisions.

A comparison of our protocol to PRoPHETv2 showed that the latter achieved slightly better performance in the network. We believe this difference is a consequence of the difficulty to select an optimal combination of the involved parameters.

With respect to throwboxes, it was shown that placing a throwbox in frequented areas can lead to an increase in bundle delivery ratio, as well as a significant reduction of end-to-end latency. When supporting delay-tolerant networks by means of throwboxes, choosing appropriate locations can have large impacts on the performance.

6.2 WiFi vs. Bluetooth

We conducted a real-world experiment using laptop-based access points as throwboxes and smart phones as mobile peers. Communication among peers was achieved using Bluetooth, between peers and throwboxes using WiFi. Our DTN framework was assigned the task of delivering bundles autonomously in the network. The data collected during the experiment allowed us to get deeper insights about communication between the nodes, especially with respect to specifics of WiFi and Bluetooth.

Regarding the comparison of the two wireless technologies, the experiment has shown that most of the communication happened over WiFi. While there have been several bundle transmissions over Bluetooth, they often have not contributed to a better path in the network, and have thus not improved the delivery ratio or latency. This may also have to do with the lack of mobility in the experiment, however similar results were obtained in the simulation.

A fundamental problem with Bluetooth communication is the short range and the low throughput. Designed for PANs, Bluetooth is not a technology that aims to provide the highest possible data rates, it rather focuses on low energy consumption. For bundle sizes between 1 KiB and 8 MiB, which represent typical sizes of media files, transmission may take several minutes using Bluetooth. This is clearly a limitation for DTNs, as it makes it impossible to transmit such media to other people when walking past them, or meeting them only for a few seconds.

Another important factor to consider is that Bluetooth discovery happens rarely, in our case in the order of minutes. While it *is* possible to decrease that interval, this is not necessarily a good idea, because the Bluetooth adapter spends most of its resources on the discovery process during that time. We have experienced that connection establishments fail during discovery. Besides inhibiting possible traffic, short-interval Bluetooth discovery also drains the battery faster. In cases where the network topology does not change for a long time, frequent discoveries are effectively a waste of power and network resources.

WiFi, on the other hand, allows very high throughputs and decent ranges, depending on the used IEEE 802.11 standard. The fact that access points broadcast their SSID constantly makes WiFi scanning less expensive than Bluetooth discovery and yields faster results. Combined with the higher range and the short scanning intervals, this increases the probability of exchanging data with a throwbox just by walking by significantly.

We therefore conclude that Bluetooth as a convergence layer is of limited utility in general delay-tolerant networking. However, Bluetooth is still helpful in various scenarios. If we know that people gather for a longer time – e.g. in meetings, train, bus, restaurants – the throughput and discovery time will suffice to exchange bundles. Also, applications based on bundles of the size of a few KiB, such as e-mail, text messages or newspapers, can still be transmitted during short contact times (once the connection is established). Furthermore, Bluetooth can be the preferred choice when energy consumption is a critical factor.

6.3 Future work

As future work, we recommend deeper investigation of the parameters used by our routing protocol. Instead of one-dimensional parameter sweeps, systematic optimization algorithms could be used. Given the large search space, simulated annealing could be a promising optimization approach.

An interesting extension would be the analysis of more delay-tolerant networking scenarios. In addition to the community-based random model and the GPS traces, more real-world situations could be modeled, for example people commuting between cities, sparsely populated areas or transit networks. By laying a focus on more network topologies, the routing algorithm is not optimized specifically for one possible situation.

The DTN framework's basic functionality is available, but it can be extended in many ways. A possible addition would be further convergence layer adapters. Since Bluetooth has showed questionable performance for delay-tolerant networking in highly dynamic scenarios, a potential alternative would be WiFi Direct, which has been available on an increasing number of Android devices lately. Given its higher range and throughput, WiFi Direct could be used for peer-to-peer communication to overcome the limitations of Bluetooth. Even an adaptive mode that switches technology based on battery usage policy could be considered. Furthermore, the DTN framework does currently not exploit all the features specified by the Bundle Protocol and implemented by JDTN. Examples are custody transfer and bundle fragmentation. Especially the latter could mitigate the situation where one node is encountered many times, but not for long enough to transmit complete bundles (as it was often the case with Bluetooth).

The C++ simulation software that started as a simple visualization of network nodes has been constantly enhanced with new functionality. It has reached a feature-richness at which it can be applied to test further DTN scenarios. Based on insights from this thesis, modeling could be made more realistic; for example, Bluetooth and WiFi scanning could be implemented, or a probabilistic failure model could be used to reproduce sporadic real-world communication issues.

Bibliography

[1] Anton Bekkerman and Gregory Gilpin. High-speed internet growth and the demand for locally accessible information content. *Journal of Urban Economics*, 77:1–10, 2013.

[2] Scott Burleigh, Adrian Hooke, Leigh Torgerson, Kevin Fall, Vint Cerf, Bob Durst, Keith Scott, and Howard Weiss. Delay-tolerant networking: an approach to interplanetary internet. *Communications Magazine, IEEE*, 41(6):128–136, 2003.

[3] Ian F Akyildiz, Özgür B Akan, Chao Chen, Jian Fang, and Weilian Su. Interplanetary internet: state-of-the-art and research challenges. *Computer Networks*, 43(2):75–112, 2003.

[4] Meenakshi Bansal, Rachna Rajput, and Gaurav Gupta. Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. *The Internet Society*, 1999.

[5] Stefano Basagni, Marco Conti, Silvia Giordano, and Ivan Stojmenovic. *Mobile ad hoc networking*. John Wiley & Sons, 2004.

[6] N4C. Networking for Communications Challenged Communities. http://www.n4c.eu. Accessed: March 2014.

[7] Vasco NGJ Soares, Farid Farahmand, and Joel JPC Rodrigues. A layered architecture for vehicular delay-tolerant networks. In *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, pages 122–127. IEEE, 2009.

[8] Wenrui Zhao, Yang Chen, Mostafa Ammar, Mark Corner, Brian Levine, and Ellen Zegura. Capacity enhancement using throwboxes in DTNs. In *Mobile Adhoc and Sensor Systems (MASS), 2006 IEEE International Conference on*, pages 31–40. IEEE, 2006.

[9] Nilanjan Banerjee, Mark D Corner, and Brian Neil Levine. An energy-efficient architecture for DTN throwboxes. In *INFOCOM 2007. 26th IEEE international conference on computer communications. IEEE*, pages 776–784. IEEE, 2007.

[10] Xiaolan Zhang, Jim Kurose, Brian Neil Levine, Don Towsley, and Honggang Zhang. Study of a bus-based disruption-tolerant network: mobility modeling and impact on routing. In *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, pages 195–206. ACM, 2007.

[11] Shimin Guo, Mohammad Derakhshani, Mohammad Hossein Falaki, Usman Ismail, Rowe Luk, Earl A Oliver, S Ur Rahman, Aaditeshwar Seth, Matei A Zaharia, and Srinivasan Keshav. Design and implementation of the KioskNet system. *Computer Networks*, 55(1):264–281, 2011.

[12] University of Waterloo – Tetherless computing lab. VLink. http://blizzard.cs.uwaterloo.ca/tetherless/index.php/VLink. Accessed: July 2014.

[13] Rajesh Krishnan, Prithwish Basu, Joanne M Mikkelson, Christopher Small, Ram Ramanathan, Daniel W Brown, John R Burgess, Armando L Caro, Matthew Condell, Nicholas C Goffee, et al. The spindle disruption-tolerant networking system. In *Military Communications Conference, 2007. MILCOM 2007. IEEE*, pages 1–7. IEEE, 2007.

[14] Alex (Sandy) Pentland, Richard Fletcher, and Amir Hasson. DakNet: Rethinking connectivity in developing nations. *Computer*, 37(1):78–83, 2004.

[15] W. Zhao, M. Ammar, and E. Zegura. Controlling the mobility of multiple data transport ferries in a delay-tolerant network. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 1407–1418 vol. 2, March 2005.

[16] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *ACM Sigplan Notices*, volume 37, pages 96–107. ACM, 2002.

[17] DTNRG. Delay Tolerant Networking Research Group. http://www.dtnrg.org. Accessed: July 2014.

[18] RFC 4838. Delay-Tolerant Networking Architecture. http://tools.ietf.org/html/rfc4838. Accessed: March 2014.

[19] RFC 5050. Bundle Protocol Specification. http://tools.ietf.org/html/rfc5050. Accessed: March 2014.

[20] RFC 6257. Bundle Security Protocol Specification. http://tools.ietf.org/html/rfc6257. Accessed: March 2014.

[21] Margaret Butler. Android: Changing the mobile landscape. *Pervasive Computing, IEEE*, 10(1):4–7, 2011.

[22] Hervé Ntareme, Marco Zennaro, and Björn Pehrson. Delay tolerant network on smartphones: Applications for communication challenged areas. In *Proceedings of the 3rd Extreme Conference on Communication: The Amazon Expedition*, page 14. ACM, 2011.

[23] HURRywalla. Android implementation of HURRy and BPQ over Bytewalla for DTN. http://sourceforge.net/p/hurrywalla/wiki/Home. Accessed: January 2014.

[24] NUS DTN Middleware. Middleware for accelerated DTN Application development. http://www.comp.nus.edu.sg/ kartiks/nusdtn. Accessed: January 2014.

[25] BP-RI. Long-delay and fault-tolerant network research. http://irg.cs.ohiou.edu/ocp/index.html. Accessed: January 2014.

[26] RFC 5326. Licklider Transmission Protocol - Specification. http://tools.ietf.org/html/rfc5326. Accessed: February 2014.

[27] IBR-DTN. A modular and lightweight implementation of the bundle protocol. http://trac.ibr.cs.tu-bs.de/project-cm-2012-ibrdtn. Accessed: January 2014.

[28] Michael Doering, Sven Lahde, Johannes Morgenroth, and Lars Wolf. IBR-DTN: an efficient implementation for embedded systems. In *Proceedings of the third ACM workshop on Challenged networks*, pages 117–120. ACM, 2008.

[29] Waldir Moreira, Ronedo Ferreira, Douglas Cirqueira, Paulo Mendes, and Eduardo Cerqueira. SocialDTN: a DTN implementation for digital and social inclusion. In *Proceedings of the 2013 ACM MobiCom workshop on Lowest cost denominator networking for universal access*, pages 25–28. ACM, 2013.

[30] Amin Vahdat, David Becker, et al. Epidemic routing for partially connected ad hoc networks. Technical report, Technical Report CS-200006, Duke University, 2000.

[31] John Burgess, Brian Gallagher, David Jensen, and Brian Neil Levine. MaxProp: Routing for vehicle-based disruption-tolerant networks. In *INFOCOM*, volume 6, pages 1–11, 2006.

[32] Aruna Balasubramanian, Brian Levine, and Arun Venkataramani. DTN routing as a resource allocation problem. *ACM SIGCOMM Computer Communication Review*, 37(4):373–384, 2007.

[33] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259. ACM, 2005.

[34] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic routing in intermittently connected networks. *ACM SIGMOBILE mobile computing and communications review*, 7(3):19–20, 2003.

[35] Samo Grasic, Elwyn Davies, Anders Lindgren, and Avri Doria. The evolution of a DTN routing protocol – PRoPHETv2. In *Proceedings of the 6th ACM workshop on Challenged networks*, pages 27–30. ACM, 2011.

[36] Ting-Kai Huang, Chia-Keng Lee, and Ling-Jyh Chen. Prophet+: An adaptive prophet-based routing protocol for opportunistic networks. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 112–119. IEEE, 2010.

[37] IEEE Standard for Information Technology – Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput. *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)*, pages 1–565, Oct 2009.

[38] Bluetooth special interest group. https://www.bluetooth.org. Accessed: July 2014.

[39] Bluetooth Special Interest Group (SIG). RFCOMM with TS 07.10. Serial Port Emulation.

[40] JDTN. Java-based implementation of Delay Tolerant Networking. http://sourceforge.net/projects/jdtn. Accessed: July 2014.

[41] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[42] Craig W. Reynolds. Steering behaviors for autonomous characters. http://www.red3d.com/cwr/steer. Accessed: July 2014.

[43] The dartmouth/cenceme dataset. Dataset of sensor data collected by the CenceMe system. http://crawdad.org/ crawdad/dartmouth/cenceme/. Accessed: July 2014.