



Bytewalla IV

Routing and Application Layer Optimizations for Delay-Tolerant Networks

Implement the P_{Ro}PHET's queuing mechanism and design a solution for applications over DTN

MICHEL HOGNERUD

Master's Thesis at TSLab
Supervisor: Hervé Ntareme
Examiner: Peter Sjödin

Résumé

Recently, a new technology known as Delay-Tolerant Networking (DTN) has emerged. DTN seeks to address technical issues in networks that may lack continuous network connectivity. For instance, remote villages which do not have a permanent connectivity due to the lack of infrastructure. Several practical projects have been developed based on DTN. One of them, named Bytewalla, is a project developed at KTH since 2009. It is a DTN implementation running on the Android-platform and its goal is to bring Internet connectivity to remote villages. However, the DTN applications are still very few compared to the ones available for Internet. They are also difficult to integrate as compared to regular Internet applications. This can be explained by the fact that nowadays protocols were not designed for partly connected and disruptive environments. This thesis aims to improve the DTN implementation in Bytewalla for better performances and to design and implement an architecture to offer better support for regular network applications. As part of this thesis, a SMTP application (mail client) will be integrated over DTN as a proof-of-concept for the Android and the Ubuntu operating systems.

This report is prepared as a partial fulfillment of my Master's thesis on "Routing and Application Layer Optimizations for Delay-Tolerant Networks".

Keywords : DTN, PRoPHET, Android, Bytewalla, SMTP

Acknowledgments

I am grateful to my supervisor Hervé Ntareme for assisting me during this thesis. Also, I would like to offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Stockholm, June 29, 2011

Michel Hognerud

Abbreviations and Acronyms

RFC	Request for Comments
IPND	DTN IP Neighbor Discovery
IP	Internet Protocol
TCP	Transmission Control Protocol
DTN	Delay-Tolerant Networking
DTNRG	Delay-Tolerant Networking Research Group
SMTP	Simple Mail Transfer Protocol
PRoPHET	Probabilistic Routing Protocol for Intermittently Connected Networks
KTH	Kungliga Tekniska högskolan
IPN	Interplanetary Internet
IPNRG	IPN Research Group
SDNV	Self-Delimiting Numeric Values
OSI	Open Systems Interconnection
JPL	Jet Propulsion Laboratory
SQL	Structured Query Language
EID	Endpoint Identifiers
RIB	Routing Information Base
TLV	Type-Length-Value
ADU	Application Data Unit
API	Application Programming Interface
CCSDS	Consultative Committee for Space Data Systems
DNS	Domain Name System
FIFO	First-In First-Out
HTML	Hyper-Text Markup Language
HTTP	Hyper-Text Transfer Protocol
JPL	Jet Propulsion Laboratory
MX	Mail Exchange Record
MOFO	Most Forwarded First Out
MIME	Multipurpose Internet Mail Extensions
POP	Post Office Protocol
RIB	Routing Information Base
SNC	Sámi Network Connectivity
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UDP	User Datagram Protocol
SNMP	Simple Network Management Protocol

Contents

Contents	vi
1 Introduction	1
1.1 Overview	1
1.2 Problem Statement	1
1.3 Criteria	3
1.4 Thesis Organization	5
2 Background and Related Work	7
2.1 Motivation	7
2.2 DTN Concept	7
2.2.1 Early research	8
2.2.2 NASA and IPN	8
2.2.3 DTNRG	8
2.3 DTN Applications	9
2.3.1 DakNet	9
2.3.2 N4C	9
2.3.3 Sámi Network Connectivity (SNC)	10
2.4 Bytewalla	10
2.4.1 Bytewalla I	10
2.4.2 Bytewalla II	10
2.4.3 Bytewalla III	11
2.5 Routing in Delay-Tolerant Network	11
2.6 Summary	12
3 Specifications	13
3.1 The Bundle Protocol	13
3.1.1 DTN Architecture	13
3.1.2 Application Data Units, Bundles, Blocks	14
3.1.3 Bundle Status Reports	15
3.2 Routing	16
3.2.1 Epidemic routing	17
3.2.2 PRoPHET	17

3.3	Neighbor Discovery	18
3.4	Summary	18
4	Contribution	19
4.1	Queuing mechanism	19
4.1.1	Queuing Policies	19
4.2	Applications over DTN	20
4.2.1	Requirements	21
4.2.2	Application to DTN Interface	21
4.2.3	Identifying the bundles	23
4.2.4	Synchronous Data Access	24
4.3	Case: SMTP over DTN	26
4.3.1	SMTP over DTN Architecture	26
4.3.2	SMTP Protocol Spoofing	26
4.4	DTN Management	27
5	Implementation	31
5.1	Software Development Approach	31
5.2	Queuing Mechanism	31
5.2.1	Design in Bytewalla	31
5.2.2	Queuing Policies	32
5.3	Applications over DTN	33
5.3.1	Ubuntu	33
5.3.2	Android	37
5.4	DTN Management	39
6	Testing & Analysis	41
6.0.1	Test environment	41
6.0.2	Methodology	42
6.0.3	Measurements	42
6.0.4	Observations and Summary	42
7	Conclusion	45
7.1	Summary	45
7.2	Future Work	45
A	Testing FIFO and MOFO	47
	Bibliography	51

Chapter 1

Introduction

1.1 Overview

Internet allows people to communicate from far distances. It is a great opportunity for many people and the economy. Nevertheless, not everyone has access to these technical facilities. Some areas, especially developing countries and rural areas do not have this chance, hence increasing the gap between developed and developing countries. In other situations, such as recently in the Arab world, access to Internet is disabled and prevents people from communicating with the rest of the world.

In order to provide connectivity to remote areas, Bytewalla was started at KTH in Fall 2009. Bytewalla provides an application which helps to carry data in mobiles from the source to the destination, similar to a postman collecting letters to offices.

Several applications have already been developed with Bytewalla. For example mail integration and a healthcare application. In this thesis however, DTN has been optimized for developing applications easily on it. A solution has been studied and implemented in order to reply to this issue. Also, optimization techniques have been considered and partially implemented to improve applications communications over DTN. These improvements should help DTN to penetrate in real-world situations and make it easier to implement applications in DTN environments.

1.2 Problem Statement

Since the birth of computer-mediated communication was first implemented in the US in the early 1960s, Internet has known a high adoption rate in the industrialized countries. It has now become part of many people's lives as a convenient real-time communication solution, and plays a major role in the economy.

However, there are great disparities in opportunity to access the Internet between developed and developing countries. This has been described as the term "global digital divide". Global digital divide points out the geographical division for Internet access. The emergence of the information revolution in countries like Swe-

den and United States has reinforced their lead in the economy, while developing countries did not get profit from it, increasing the gap between these countries. [32]

Recent surveys show great differences in Internet usage between world regions [41]. Hence, 77.3% of the population of the United States has access to Internet. Scandinavia has even greater penetration, with 86.1%, 92.5%, 94.8% for Denmark, Sweden and Norway respectively. On the opposite, countries such as Burkina Faso, Congo and Bangladesh have respectively 1.1%, 0.5% and 0.4% of their population which have access to Internet. On a more global scale, we see that Europe, North America and Oceania have the highest rates on contrary to world regions such as Africa and Asia.

In order to provide connectivity to rural areas and challenged networks, a new approach known as Delay-tolerant networking (DTN) was developed. DTN is meant to provide connectivity in heterogeneous networks that may lack continuous connectivity due to disruptions or considerable delay.

However, in these challenge environments, popular ad hoc routing protocols such as AODV (Ad hoc On-Demand Distance Vector Routing) fail to establish routes. A routing protocol named PROPHET for “Probabilistic Routing Protocol for Intermittently Connected Networks” was developed since 2003 by Lindgren, et al [2]. In realistic situations, data mules encounters are rarely random. They move in a society and tend to have greater probabilities to meet certain mules than others. Hence, PROPHET makes use of their history of encounters to maintain a set of probabilities for successful deliveries to known destinations and to route the data through the mules which have the best chances of delivering the data to its final destination.

The Bytewalla project was started at KTH in Fall 2009. Bytewalla is the DTN implementation on the Android-platform. The purpose is to connect African rural villages using Android phones with delay-tolerant networking [13]. The idea behind it is that people traveling between villages and cities while carrying their phones will carry data along their movements. The scenario is explained in the Figure 1.1. A "mule" (an Android phone) will connect a WiFi access point located in a village with no connectivity to Internet, and download the data. Once it reaches the city, the data's destination, the phone connects to the local WiFi access point and uploads the data. This works also on the other way, city to village.

Bus and cars doing regular trips between villages and cities could be used as mules by carrying an Android-phone running Bytewalla.

Bytewalla also includes the PROPHET routing protocol. However the queuing mechanism which is necessary to know which bundles to drop when the storage gets full is missing from the implementation. Mobile phones storage space may be overloaded in case the mule receives a lot of data and it should have mechanisms to determine what data to drop first according to characteristics such as their priority or their delivery probabilities.

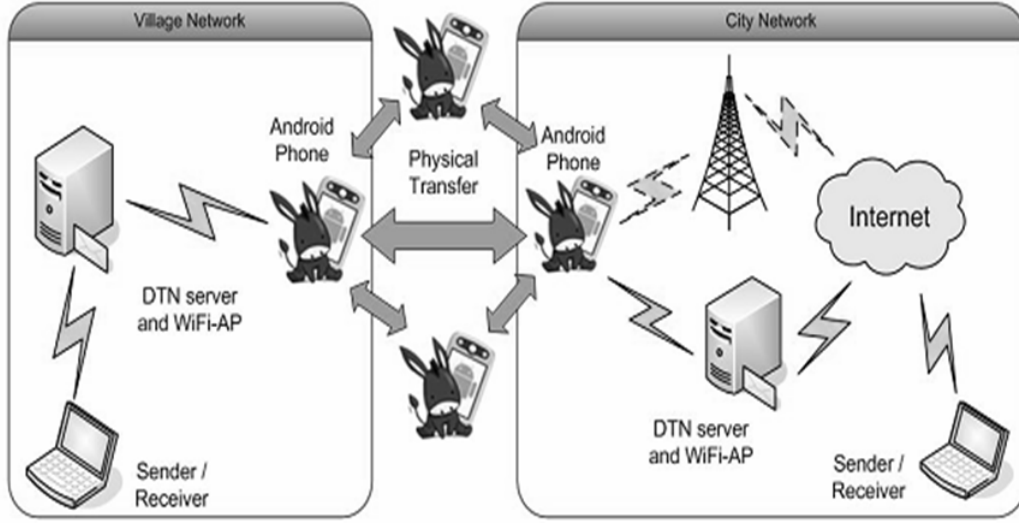


Figure 1.1. The Bytewalla System

Along with the implementation of DTN on the Android-platform, a few applications such as emails, management tool and healthcare have been developed making use of DTN. However, integrating applications over DTN is still not a convenient process as most applications are optimized for TCP/IP communications.

In his paper [38], Jörg Ott talks about the difficulties met with integrating an application protocol with a mobile Internet. As he says, "the semantics of many of today's non-real-time-applications are perfectly compatible with disruptive mobile environments, it is just the protocol designs that are not". He points out some issues such as working with intermediaries and protocol operations. For example, sending a mail on Internet requires to go through intermediaries known as mail servers. In order to avoid single points of failures, applications should be designed for direct end-to-end operation, while the intermediaries would only focus on message forwarding. Also, entities need to be more proactive and communicate all their intentions at once rather than iteratively interacting with a peer, as each iteration would bring much more delay to the operation.

With Bytewalla, using emails with DTN requires to setup and configure many tools such as DNS, Postfix and Python [22] on the machine from which the mail is being sent. This is a complicated and a very resource hungry system, especially for devices which do not have regular energy supply.

1.3 Criteria

A set of requirements have been established as defined below. But first, a literature study is carried out considering DTNs specifications.

The main objective is to provide an architecture which improves applications integration on DTN, their management as well as their reliability and delays. At the end, the system is tested to check that its functionalities are working accordingly with our goals. The other objective is the improvement of the PROPHET implementation with queuing mechanisms. Hence, the main requirements are:

1. Implementation of PROPHET's queuing mechanisms.

The implementation will strictly follow the PROPHET specification from "Lindgren, et al" and it will be a part of the existing Android application Bytewalla. The implementation must be flexible enough to be able to use different queuing policies according to the application's configuration. We will also implement two queuing policies depending of their efficiency which will be discussed in Chapter 4.

2. Design and implementation of the Application Layer Optimization.

The application layer stands over the Bundle Protocol layer. The thesis goal is to ease the development of applications on top of DTN.

- As for now, it is difficult to use DTN with multiple applications. The Bundle Protocol does not provide any information about the applications to which the bundles is intended to be delivered, as ports do in the TCP/IP model. Hence, we will provide a mechanism for using multiple applications in the same DTN network.
- Developing an application for DTN is not an easy process as the email integration in Bytewalla 1 shows. Hence, we will propose an interface to run applications on a DTN network (e.g. Bytewalla) and to help building applications on top of DTN without having to worry about the underlying DTN layers.
- The Application Layer Optimization will also give the ability to include optimization techniques to improve delay, reliability such as protocol spoofing. Protocol spoofing will let the applications communicate "all-at-once" instead of iteratively.
- A management interface will be developed to provide administrators with statistics and information about the application communications over DTN.
- Based on this system, it will also be possible to implement more tools. For example a subscription service: a village in a remote area could receive courses on a regular basis with DTN.

In order to achieve this we will present its design and implement a proof-of-concept tool for the DTN nodes. This tool will include optimization techniques and interfaces to the applications willing to communicate through DTN. Besides, we will deliver a management interface allowing the administrator to view and configure the network. This is more thoroughly described in Chapter 4.

3. Performance analysis.

The existing tools (Bytewalla, Application Layer, DTN daemon) will be tested in order to ensure it works efficiently.

1.4 Thesis Organization

The thesis is organized in 8 chapters. The Chapter 2 presents the background of the DTN concept and the related work. The related work is a set of DTN related projects which have been conducted in the past or are still continued. One of this related project is named Bytewalla or is the project on which this thesis is based. Bytewalla has been through three iterations already and the work performed as part of this project is presented in Section 2.4. Then in Chapter 3, we present the technical background and the specifications required as part of this thesis. This includes the Bundle Protocol and its some of its companion concepts such as the DTN routing protocols and the Neighbor Discovery mechanism. Following this chapter, we present in Chapter 4 the design of our implementation. There we discuss about the available solutions and we explain our choices. Logically, we then explain the implementation part in Chapter 5. Finally we test the implementation and measure its performance in Chapter 6.

Chapter 2

Background and Related Work

This chapter first explains the need for the DTN technology in 2.1. Then we identify the different research groups involved in the field, and we present their contributions which led to the current standards of DTN. Later on, we give an overview of some the practical applications which have been developed with the idea of delay and disruption-tolerant networks. Finally, we focus on some of the research which was performed for routing protocols in Delay-Tolerant Networking.

2.1 Motivation

The current Internet protocols do not perform well in some environments because of some of their fundamental assumptions which are built-on their architectures:

1. An end-to-end path exists between a data source and its peer
2. The maximum round-trip time between any node pairs in the network is not excessive
3. The end-to-end packet drop probability is small

Unfortunately, challenged networks may not be able to meet these assumptions. Such examples are Inter-planetary networks and Terrestrial Mobile Networks (unexpected partition due to nodes mobility).

In an effort to adapt Internet to unusual environments, research in this area was conducted since a few decades ago as described in the next section. It has first been focusing on Inter-planetary networks, but some research groups have recently decided to work on terrestrial networks.

2.2 DTN Concept

The DTN history goes back to many years ago. It started with a project led by several space agencies and later evolved into a terrestrial network based on the previous work. Today, a whole research field has emerged around the DTN concept.

2.2.1 Early research

The Consultative Committee for Space Data Systems (CCSDS)[21] composed of world's space agencies was created in January 1982 at an International Workshop on Space Data Systems held in Washington DC, USA. The CCSDS's goal was to develop advanced standardized solutions for exchanging space mission data. As part of this, the members created the final CCSDS Recommendations which served to guide the internal development of standards by each of the members.

2.2.2 NASA and IPN

In 1998, Vint Cerf and scientists from NASA's Jet Propulsion Laboratory (JPL) started working on Interplanetary Internet (IPN). In the IPN scenario, transmission is subject to significant delays and intermittent connectivity due to planets and spacecrafts movements.

In August 2002 the IPN research group (IPNRG) published the draft "Delay-Tolerant Network Architecture: The Evolving Interplanetary Internet" [16] which describes the architecture designed for IPN. This work led to the concept of bundles as a way to address the Store-and-Forward problem. Bundles are an area of new protocol which sits above the Transport layer in the OSI model.

2.2.3 DTNRG

The DTNRG (DTN Research Group) was formed in 2002 to generalize the IPNRG's work to networks other than those operating in deep space. It proposes an alternative to the Internet TCP/IP end-to-end interactive delivery model and employs hop-by-hop storage and retransmission as a transport-layer overlay [28]. The main difference between interplanetary-networks and terrestrial networks is that IPN works in a scheduled manner while terrestrial networks are in general opportunistic. However they have in common that they both deal with delays and disruptions.

The DTNRG released a description of the architecture of DTN [19] in 2003. Since then, the DTNRG published more documents. Some of the more important ones are:

1. RFC 4838 "Delay-Tolerant Networking Architecture" [14]
2. RFC 5050 "Bundle Protocol Specification" [10]
3. Delay Tolerant Networking TCP Convergence Layer Protocol (Internet Draft) [37]
4. UDP Convergence Layers for the DTN Bundle and LTP Protocols (Internet-Draft [36])

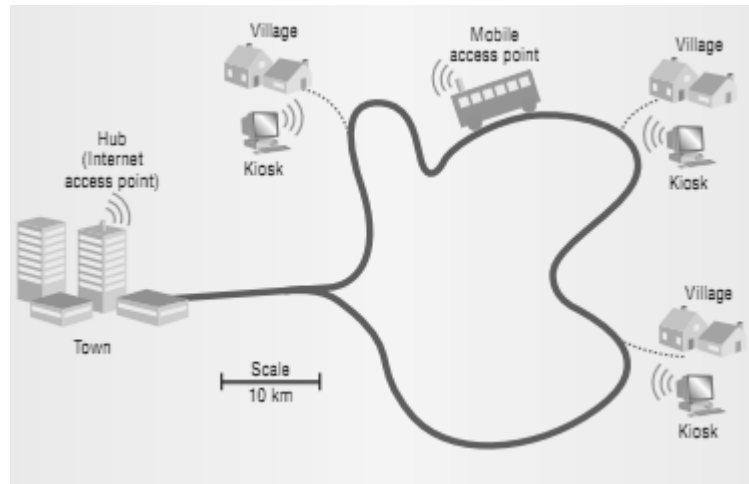


Figure 2.1. The DakNet concept

2.3 DTN Applications

Here we present some of the practical applications for delay and disruption-tolerant networks. All these applications have different goals such as animal tracking or providing Internet connectivity but they are all more or less working with delay-tolerant networks.

2.3.1 DakNet

DakNet [4], developed by MIT Media Lab researchers, was one of the first practical application with DTN. Its goal is to provide very low-cost digital communication to remote villages. It has been deployed in remote parts of both India and Cambodia. In Figure 2.1, a bus carrying a mobile access point travels between village kiosks and a hub with Internet access. Data automatically uploads and downloads when the bus is in range of a kiosk or the hub.

It has been used to send voice messages and emails.

2.3.2 N4C

The Networking for Communications Challenged Communities (N4C) project is funded by the European Union to provide connectivity to remote European regions. Indeed, many regions do not have links to the world networks because they are sparsely populated or have a relatively poor economic base.

Hence, with the help of DTN, N4C aims to create an ‘opportunistic networking architecture. Two testbeds are set up in Swedish Lapland and Slovenian’s mountain.

Three application tests were carried out on animal tracking, pod cast application and meteorological data. [35]

2.3.3 Sámi Network Connectivity (SNC)

SNC is a project which seeks to establish Internet communication for the Saami population who live in remote areas in Northern Scandinavia. This community is nomadic and has poor infrastructure to access information.

SNC is conducted at the Luleå University of Technology (Sweden).

2.4 Bytewalla

Bytewalla is the project which serves as a basis for this thesis. Its objective is to connect rural and remote areas to Internet. In order to achieve this, the DTNRG implementation was ported on the Android platform along with the development of some DTN applications.

In this section we present the three steps that compose the Bytewalla project. Bytewalla consists of two semester group projects and one thesis which precisely focused on the security considerations. The two other iterations led to the implementation of DTN on the Android platform along with a couple of DTN applications. Each iteration includes the features from the previous iteration.

2.4.1 Bytewalla I

Bytewalla I was the first iteration of the Bytewalla Project. It was held in Fall 2009 in KTH. The main objectives were:

1. Porting the standard DTN implementation on the Android platform.
The standard DTN implementation was developed by DTNRG [29].

2. Email integration.

Bytewalla chose to develop an email application which serves as a proof-of-concept for their DTN implementation. The application can support MIME types ([25], [26], [33], [34], [23]) as payload. Hence, the users can attach any digital files such as images, videos, voices to the email as well.

The integration relies on the mail system POSTFIX and a DNS server which need to be installed to send and receive emails. Some Python scripts are also required in order to convert the mails into bundles and vice versa. The setup must be configured following two documents: Postfix DTN2 Integration [7] and Postfix and DNS [6]. It is a long process (17 pages of instruction in total) and requires many tools (Postfix, DNS, Python).

2.4.2 Bytewalla II

Bytewalla II was a thesis conducted in Spring 2010 [18].

The thesis main objective was to deploy a standardized security solution for DTN networks in Android, which could be implemented in Bytewalla. The work

was based on several documents such as DTN Security Overview [20] and Bundle Security Protocol Specification [42].

2.4.3 Bytewalla III

Bytewalla III was a group project conducted in Summer 2010 and was focusing on several objectives:

1. Implementation of PRoPHET
PRoPHET was integrated into Bytewalla, following the PRoPHET Internet-Draft version 8 [3].
2. Neighbor Discovery
Before then, the addressing was static. Thus, nodes were not able to discovery each other. With Neighbor Discovery, nodes can discover their neighbors and start exchanging information. This work was an implementation of the DTN IP Neighbor Discovery (IPND) InternetDraft [9].
3. Network Management Tool [11]
One of the two applications that were developed in Bytewalla 3 is the Network Management Tool. This tool informs the administrators about the statistics of the DTN network, such as the number of bundles delivered, transmitted, and so on. Besides that, it also let the administrator generate a new configuration file for the DTN daemon. However the administrator would need to replace the actual configuration file manually. Also, this tool only gives global statistics and does not provide information specific to a bundle such as whether it was delivered.
4. Sentinel Surveillance Application [12]
The other one is the Sentinel Surveillance Application. This is a healthcare application whose goal is to provide communication to doctors in remote areas. Doctors can register records about their patients and everything is synchronized with a remote server. Every time a record is added to the database, the SQL query is bundled and sent to the remote host through the Bundle Protocol. The receiver then unpacks the bundle and execute the SQL query on its own database.

2.5 Routing in Delay-Tolerant Network

Traditional TCP/IP routing protocols cannot be used with DTN. These protocols try to establish and complete end-to-end route, and then forward the data. In the DTN case, this is not possible as end-to-end paths are difficult or impossible to establish. Hence, a “store and forward” approach is adopted.

Several routing protocols have been designed based on this approach. Epidemic routing was the first routing protocol designed for DTN [43]. It is flooding-based in nature: nodes continuously replicate the data to other nodes as they meet, so that

the data eventually reaches its destination. Epidemic routing is resource hungry as it makes duplication of the data without attempting to eliminate the duplications which do not improve the delivery probability. With mobile devices having limited storage and energy capacities, resources should be used wisely.

Hence, P_{Ro}PHET is a variant on the epidemic routing protocol and aims to reduce resource usage and still attempt to achieve the best case routing capabilities for epidemic routing. The key idea is that in real-world situations, encounters between data mules are not random as the mules move in a society. So if a mule has already met another mule, it is likely that they will meet again. So P_{Ro}PHET keeps track on the encounters a mule makes and computes the delivery probabilities for each known node. This way, data is passed from a node to another one only if it increases the chances of delivery.

2.6 Summary

In this chapter we introduced the concept of Delay Tolerant Networking. First we explained the motivation for developing Delay Tolerant Networking, then we an overview of its historical background, and finally, we showed some of its practical applications along with the development of the routing protocols in DTN.

The next chapter will focus on Bytewalla, on which this thesis bases itself.

Chapter 3

Specifications

This chapter presents the specification necessary to understand this thesis. This includes the DTN architecture and the Bundle Protocol, as well as the DTN routing protocols and the Neighbor Discovery mechanism. All these concepts will be involved in the design and the implementation part of this thesis.

3.1 The Bundle Protocol

3.1.1 DTN Architecture

The RFC 4838 describes the architecture for Delay-Tolerant Networks. As stated before, the Internet architecture relies on assumptions like end-to-end connectivity and low round-trip delays. To circumvent these requirements, the DTN architecture has adopted a store-and-forward approach. Data are packed into bundles which are saved with persistent storage. Hence the nodes can keep the data even over long network disruptions.

According to the DTN architecture, the bundle layer is above the transport layer. Not all transport protocols provide the exact same functionality, so some adaptation is required between the transport protocols and the bundle protocol. This is accomplished by a set of convergence layers placed between the bundle layer and underlying protocols. The convergence layer takes care of the specificities of the transport protocol and presents a consistent interface to the bundle layer. The complexities of the convergence layers depend on the transport protocol. For example the TCP convergence layer [37] would not have to worry about reliability as it is already implemented in TCP, while the UDP convergence layer [36] may handle it itself.

Nodes are identified with Endpoint Identifiers (EID). Each node is required to have a unique EID. An EID is a name using the syntax of URI [8].

For reliability, the bundler layer provides two options: end-to-end acknowledgments (Bundle Status Reports) and custody transfer. DTN applications may also

Version	Proc. Flags (*)
Block length (*)	
Destination scheme offset (*)	Destination SSP offset (*)
Source scheme offset (*)	Source SSP offset (*)
Report-to scheme offset (*)	Report-to SSP offset (*)
Custodian scheme offset (*)	Custodian SSP offset (*)
Creation Timestamp time (*)	
Creation Timestamp sequence number (*)	
Lifetime (*)	
Dictionary length (*)	
Dictionary byte array (variable)	
[Fragment offset (*)]	
[Total application data unit length (*)]	

Figure 3.1. The Bundle Primary Block

implement their own reliability mechanism.

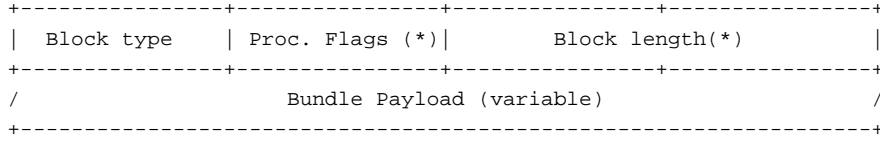
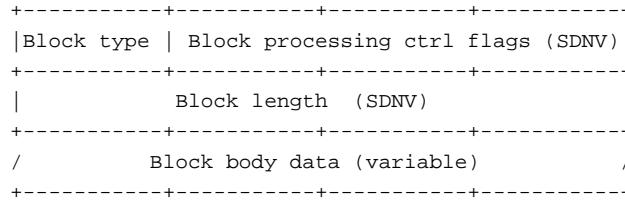
3.1.2 Application Data Units, Bundles, Blocks

Applications communicate with the bundle layer to send and receive data. When an application sends an application data unit to the bundle layer, the bundle layer will pack the data into one or more bundles (Bundle Protocol Data Units).

Each bundle is composed of at least two blocks:

1. **Primary Bundle Block** (see Figure 3.1)
This is the first block. It contains information such as the source, the destination, lifetime, creation timestamp.
2. **Bundle Payload Block** (see Figure 3.2)
This block contains the payload received from the application layer.
3. **Extension blocks** (see Figure 3.3)
These blocks are optional. They are used for specific cases.

Extension blocks and the Bundle Payload Block all follow a common format (block type, flags, length, content). The Bundle Payload Block's type code is 1.

**Figure 3.2.** The Bundle Payload Block**Figure 3.3.** The Bundle Extension Block

3.1.3 Bundle Status Reports

As we saw in the Subsection 3.1.1, the end-to-end reliability mechanism is ensured by Bundle Status Reports. Bundle Status Reports are standard bundles whose payload is a status report. The reports can inform the node indicated in the Report-To field about different types of events:

1. The reception of a bundle
2. The forwarding of a bundle
3. The delivery of a bundle
4. The deletion of a bundle

It includes the time of the event and provides the identifiers of the bundles concerned by the status reports.

The Figure 3.4 shows the representation of a bundle status report.

The status flag informs about what the status report is reporting (e.g. received bundle, forwarded bundle, etc.).

The reason code informs about the reason for the event that is being reported (e.g. lifetime expired, transmission canceled, depleted storage, etc.).

The fragment fields are only used for fragmented bundles.

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

Figure 3.4. Bundle Status Report

The "Time of" fields report the time of the event. The status flag informs about what the status report is reporting (e.g. received bundle, forwarded bundle, etc.).

The "Copy of bundle X's Creation Timestamp time" field is a copy of the creation timestamp time of the bundle that the status report concerns. It helps to identify the bundle whose the status report is originating from.

The "Copy of bundle X's Creation Timestamp sequence number" field is a copy of the creation timestamp sequence number of the bundle that the status report concerns. It helps to identify the bundle whose the status report is originating from.

The last fields (i.e. "Length of X's source endpoint ID" and "Source endpoint ID of bundle X" gives the source of the bundle for which the status report was sent.

3.2 Routing

Routing is a really important part of DTN. It impacts on the delay and the bundles delivery success rate. The simplest one is known as Epidemic routing but some other solutions have been developed. One of them, briefly presented in the previous chapter, is known as PROPHET. Hence we will focus especially on the PROPHET specification.

3.2.1 Epidemic routing

Epidemic routing is a flooding-based type of routing protocol. Nodes will continuously replicate and transmit bundles to the other nodes they meet. No effort is made to limit the resource usage; however it offers high delivery probabilities.

3.2.2 PROPHET

On contrary to epidemic routing, PROPHET aims to a more efficient routing protocol which reaches as good delivery predictabilities as epidemic routing while using less resource.

When two nodes discover each other, they start the Information Exchange Phase. First, the node will send a Routing Information Base Dictionary (RIB Dictionary) TLV (type-length-value) to the node it is peering with. This is a dictionary of the Endpoint Identifiers (EIDs) of the nodes which will be referenced in the Routing Information Base. The next step is to send the Routing Information Base (RIB) TLV. This contains the list of EIDs that the node has knowledge of with corresponding delivery predictability. Upon reception of the RIB, the node updates its delivery predictabilities and determines which of its stored bundles it wished to offer. After the decision is made, it sends a Bundle Offer TLV containing the bundle identifiers and their destination that the node wishes to offer.

PROPHET contains an algorithm to calculate the delivery predictabilities according to the node's history. The mule A stores delivery predictabilities $P(A, B)$ for each known destination B. If the mule A has no delivery predictability stored for mule B, the value is assumed to be zero. The delivery predictabilities are recalculated according to three rules:

1. When the mule A encounters the mule B, the predictability for B is increased:

$$P(A, B) = P(A, B)_{old} + (1 - \delta - P(A, B)_{old}) * P_{encounter}$$

, where $0 \leq P_{encounter} \leq 1$ is a scaling constant setting the rate at which the predictability increases on encounters after the first and delta is a small positive number that effectively sets an upper bound for $P(A, B)$.

2. If the mule A does not encounter another mule B during some interval, the predictability is "aged":

$$P(A, B) = P(A, B)_{old} * \gamma^K$$

, where γ is the "aging constant" and K is the number of time units that has elapsed since the last aging.

3. Predictabilities are exchanged between A and B and the transitive property of predictability are used to update the predictability of destinations C for which B has a stored $P(B, C)$:

$$P(A, C) = \text{MAX}(P(A, C)_{old}, P(A, B) * P(B, C)_{old} * \beta)$$

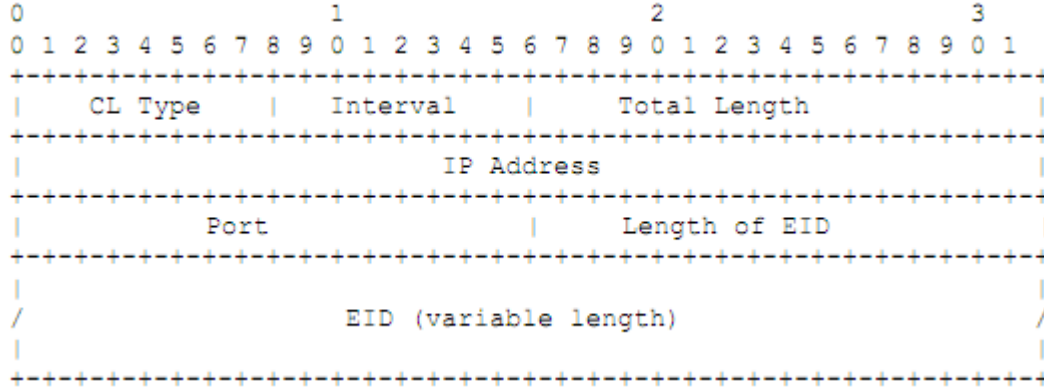


Figure 3.5. The beacon format

, where β is a scaling constant

3.3 Neighbor Discovery

DTN IP Neighbor Discovery (IPND) is documented in the Internet-Draft [9]. Shortly, it is a method for nodes to learn of the existence, availability and addresses of other nodes. IPND periodically sends (broadcast) and receives beacons to/from other nodes. These beacons are small UDP messages and contain information such as the address of the node. Upon reception of a beacon, a node will establish contact with the discovered node. IPND has been implemented in the DTNRG implementation.

In the DTNRG implementation, the beacon format is presented in the Figure 3.5.

CL Type: The convergence layer type informs the type of convergence layer option advised.

Interval: The interval for periodically sending beacons.

Port: Usually set to 4556 (the standard DTN port)

Length of EID/EID: The EID of the node sending the beacon

3.4 Summary

In this chapter we have presented the specification and the technical aspects of the Bundle Protocol, the routing in DTN and the Neighbor Discovery mechanism. Next chapter show the design of our solutions for the Queuing mechanism and the Application Layer

Chapter 4

Contribution

This chapter focuses on the solutions and the designs investigated by the author of this thesis. This chapter first explains the queuing mechanism architecture and the queuing policies that are included as part of the implementation. Then we focus on the Application Layer. First we go through the possible solutions and then we move on to a specific case with SMTP over DTN. Finally we discuss the DTN management improvements.

4.1 Queuing mechanism

Every time a bundle is added to the storage, the application should check for maintaining the quota. This is where the queuing mechanism intervenes. This should be developed in a modular way, so that the user can easily switch from one queuing policy to another one, or even add its own queuing policy. Each policy will be responsible for returning the "first" bundle in the queue, i.e. the bundle to be deleted first. The quota maintenance and the bundle deletion procedure is common to all queuing policies.

Hence, as the Figure 4.1 shows, the queuing policies will be implemented in their own classes and they will all inherit from a common ProphetQueuing class.

4.1.1 Queuing Policies

We chose which policies to implement according to their efficiency in terms of delay and deliverability. Lindgren, A. and Phanse, K.S. have proposed and evaluated some policies [31]. The queuing management policies which are evaluated in this paper are:

1. FIFO – First in first out.
2. MOFO – Evict most forwarded first.
3. MOPR – Evict most favorably forwarded first. The node keeps a value FP for each message. Each time a message is forwarded, its value FP is updated:

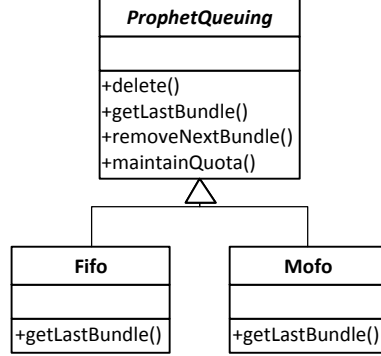


Figure 4.1. Queuing Classes UML

$FP = FP_{old} + P$, where P is the deliverability predictability for the receiving has for the message. The message with the lowest FP value will be dropped first.

4. SHLI – Evict shortest life time first.
5. LEPR – Evict least probable first. Drop the message which has the destination with the lowest predictability.

According to the paper, each queuing policy has been tested with 5 forwarding strategies and different queue sizes. However MOPR and LEPR are not applicable to two forwarding strategies. Hence we focus only on the three other remaining queuing policies.

About the deliverability, it appears that MOFO is the most efficient one no matter the queue size with three of the 5 forwarding strategies. Of the three policies applicable to all forwarding strategies, SHLI is in most cases the least efficient one.

Regarding the average delay, MOFO and FIFO reach similar results while SHLI is in any case less efficient than the two others.

Therefore, we decided to implement the MOFO and the FIFO queuing policies.

4.2 Applications over DTN

In this section we go through the possible solutions for supporting regular applications over DTN. We decide on the best solutions available and we give a clear picture of its design.

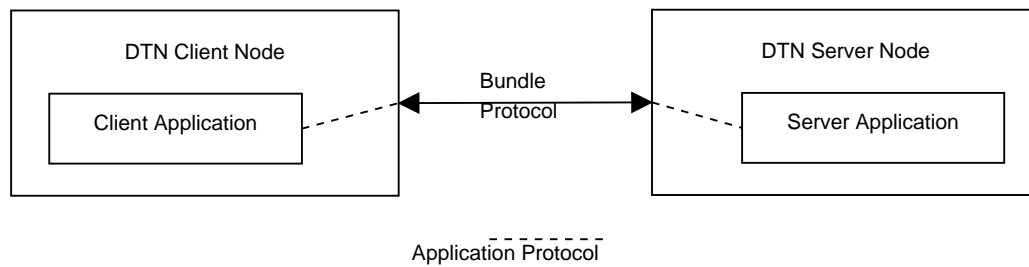


Figure 4.2. Basic DTN Setup

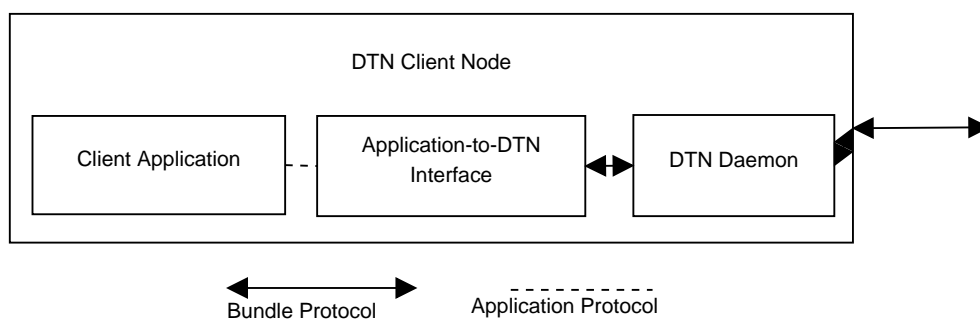


Figure 4.3. The DTN Client Node

4.2.1 Requirements

Regular applications should be able to communicate over DTN. This means sending and receiving messages to and from other DTN nodes, wherever they are. Moreover it must be easy to integrate applications with DTN. This would help having more applications available for delay-tolerant networks. As for now there is no way to identify the bundles and classify them according to the type of data they contain. Hence it is difficult to run multiple applications on the same DTN network. In fact, we need a solution to identify the bundles and process them accordingly. Finally, the solutions should be able to run on Linux and Android. Bytewalla is meant to be used both with Linux and Android.

4.2.2 Application to DTN Interface

Figure 4.2 gives a basic overview of a DTN setup with two applications (client-server) trying to communicate to each other. The client sends the request to the server, which process it upon reception. But as we have seen in the Introduction chapter, many applications are not designed for DTN networks, although their semantics are compatible with DTN. Their designs often require no delay and no disruption which DTN can't offer.

One solution is to rebuild the applications for DTN. This includes rebuilding the application protocol for a better integration with DTN. The new protocol would

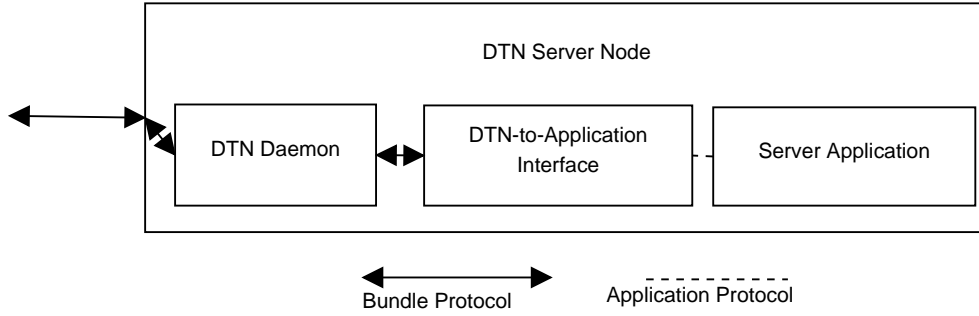


Figure 4.4. The DTN Server Node

take disruptions and delays into account and be suitable with the Bundle Protocol. In fact, this isn't always possible. This requires a lot of effort for a seldom usage on DTN, compared to Internet. It also duplicates the application for the same purpose but for two different type of networks. However, this would certainly makes the application more suitable for DTN. Anyway, it requires to redesign the application protocol and setup an interface with the Bundle Protocol.

Another solution is to use protocol spoofing, as shown in Figure ?? . The application remains the same and keeps using the same protocol as before, but the messages are reformatted before going through the delay-tolerant network. For examples, several messages or requests may be rewritten and packed together to fit in a single bundle, ready to traverse the delay-tolerant network. This leads to better performances as we can reduce the number of messages, so the overall delay and the risk of bundle loss, and this is completely transparent to the application. Only the Application protocol/Bundle protocol interface needs to be developed. Protocol stacks for popular applications are often available, making this task even easier.

In some cases, where the application is complex, the first solution would probably be the best. But it requires more effort as the whole application needs to be rewritten on contrary to the second solution, where only the messages are reformatted. In order to encourage the support of applications for DTN, we will focus on the solution which makes this task the easiest possible, the second one. For simple and popular applications such as mailing and web, it is sufficient. However for more complex situations, the first solution may be the only one available. But in any case, the design we will build, will be helpful to the first solution as well.

Figure 4.2 clearly shows that we need an interface between the applications and the Bundle Protocol. In fact, this interface must convert bundles into messages for the server application, and messages into bundles for the Bundle Protocol. Hence, for a single application protocol, we need two interfaces. One for sending messages over DTN and one for receiving them.

Figure 4.3 shows the representation of the client component. It consists of the client application (e.g. a mail user agent), the interface and a DTN daemon such

as Bytewalla (Android) or DTND (Linux). The client application connects to the interface as it would directly connect to the remote server on Internet, and the interface connects to the daemon for sending bundles. Each application has its own interface to the Bundle Protocol, so if they are multiple applications running on the machine, there must be an interface for each of this application.

About the server side now, Figure 4.4 gives an overview of how is it designed. It is pretty much similar to the client node, but on the other way. The DTN daemon receives the bundle, which is forwarded to the interface and then unpacked for transmission to the server application. At this stage, the message has traveled all through the delay-tolerant network to reach its destination as on Internet.

The Application-to-DTN interface must be able to handle requests from the client application. Hence it must have the same capabilities as the server application, however, all it does is to listen for new messages and output them into bundles. The interface is encouraged to reformat the messages in order to reduce the storage consumption and to fit it into as few bundles as possible to limit the loss and the delay.

The DTN-to-Application interface, on contrary, is called upon reception of an incoming bundle. The interface which acts here as the client, recovers the messages from the bundle and recreate the communication between the client application and the server application.

On both the client and the server side, this is totally transparent to the applications. Only the client application must be configured to connect to the Application-to-DTN interface.

When building these interfaces, the developer should pay attention not to lose information or to alter the semantics of the messages.

4.2.3 Identifying the bundles

As explained in the first chapter, there is no identifier which helps to identify the applications. Hence for example, when a bundle is received, nothing informs us about whether it is intended for the email or the healthcare application.

For this, there are two known solutions. The type of application could either be indicated in the payload by adding an extra header containing such information or in an extension block as described in 3.1.2.

Adding the application information in the payload is certainly not a good solution in that not all bundles may respect this format. On the contrary, adding an extension block will help to efficiently identify the bundles. If the extension block is missing, then we simply ignore the bundle or process it separately. This new extension block will be called application block from here.

This new type of bundles (the ones containing the application block), identified as application bundles in this document, will now be composed of three blocks. The

Primary Block, the Payload Block and the Application Block which tells about the type of service the bundle is intended to be used for. [10] states that the block type codes 192 through 255 are available for private and/or experimental use. For this experiment, we will use the block type 200.

At this moment the application block payload consists of only one field called the "application type". This field contains an integer corresponding to a particular application (e.g. 10 for HTTP).

4.2.4 Synchronous Data Access

It is good to notice that in some cases, the applications may have to be redesigned for DTN due to the long-delays and the low-reliability compared to interconnected networks such as Internet. For example, the application must not wait for an instantaneous response (e.g. a web browser requesting a page). The request and the response are not synchronized. A response may be received and handled at any time. So the application must be always be ready to handle a message containing the response to one of its previous requests.

This also implies that the user-interface may need to be redesigned as well. The user may not be able to have instantaneous feedbacks from the application, as the response will in most cases be delayed.

In the case of SMTP, we are simply sending a mail, so the application does not wait a particular response. However, it would be different with the POP protocol [24] for example. POP aims to retrieve emails. In the case of DTN, POP is not applicable because it asks for emails and hopes to get an immediate response which is not guaranteed in DTN. This should then be replaced by a mechanism where we first send a request for emails through the Bundle Protocol, and then wait for handling the response bundle containing the emails whenever we receive it. This is represented in the Figure 4.5. The request may also be replaced by a subscription service, sending emails periodically instead of having to explicitly request for them. However, once the emails have been received, it would be possible to access them through a regular POP application (e.g. Thunderbird [17]) connecting to a local POP server delivering the emails received through the Bundle Protocol.

Some solutions include rebuilding the application for better compatibility with DTN as seen previously, or developing a light local server. In the case of the POP application, we could have a light local application acting as a POP server to which the POP client would connect. The POP server would be responsible for requesting new mails (automatically on a regular basis or upon demand) and storing them until they are request by the POP client. To take another example with HTTP, the POP server would be replaced by a simple HTTP server with caching enabled.

Requesting data upon demand involves a lot of delay because it requires a full round trip to get the data. This may be improved with some techniques such as common caching or subscription services. The caching technique is described in the

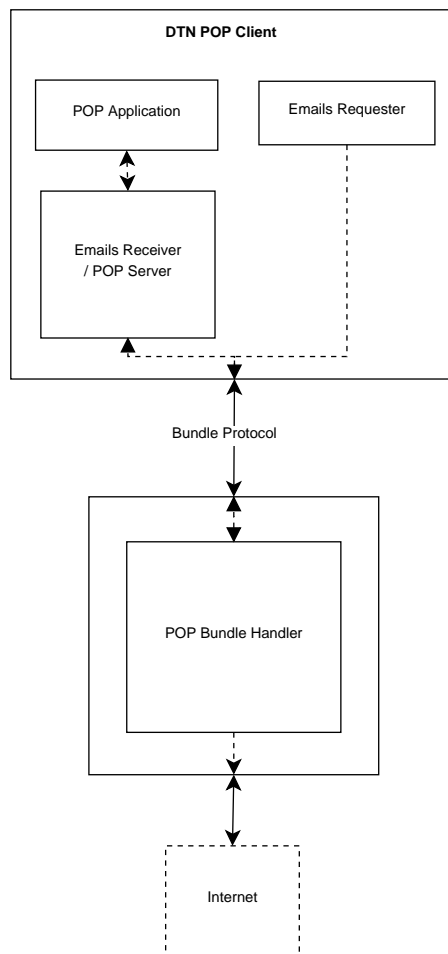


Figure 4.5. How to retrieve emails over DTN

DTN Management section (see 4.4). The subscription service consists of a node connected to Internet which would keep the non-connected DTN nodes updated with their last information such as news, medical records, mails, etc.

It should also be noted that more efforts should be put on reliability, as the DTN mail client can not know instantaneously whether its request could be sent. For example in the case of SMTP, we forward all SMTP requests into bundles. However, if the identifiers are wrong, we will be aware of it only when the DTN server will be able to forward the SMTP message. If it fails, the client should be informed by an application feedback mechanism, that the mail message failed to be sent. This was specifically investigated as part of this thesis but this can be a future work.

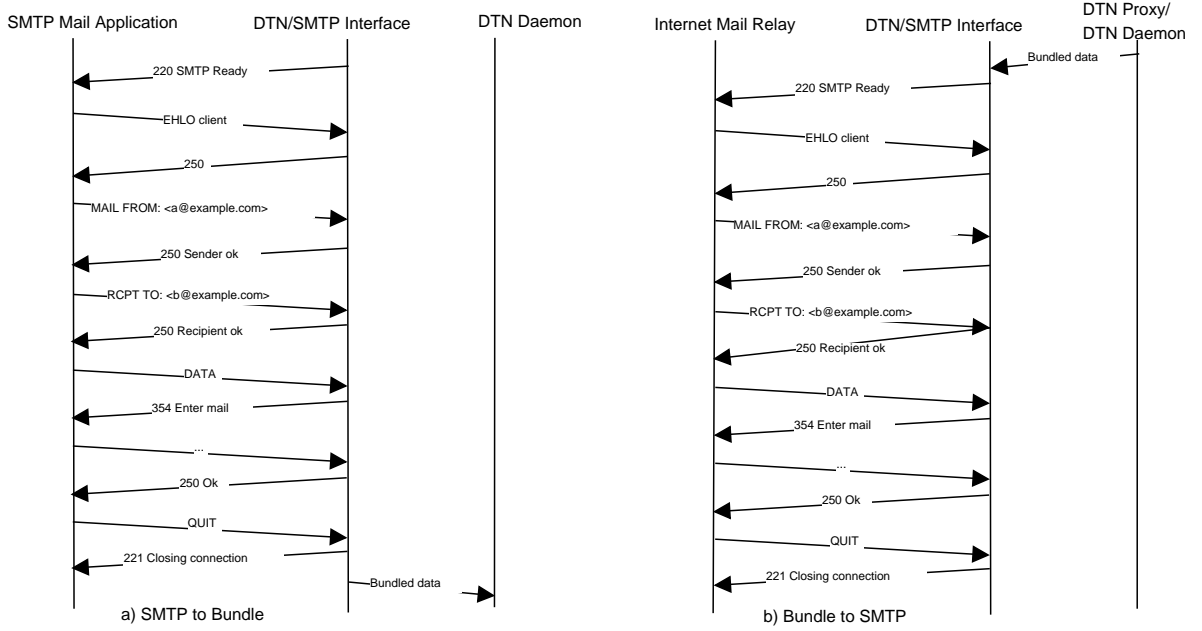


Figure 4.6. SMTP Protocol Spoofing

4.3 Case: SMTP over DTN

In this section we present how to run SMTP over our designed DTN architecture. We first show how the communication between a mail client and a SMTP server connected to Internet is performed, and then we focus on the data model used for carrying the SMTP data within the Bundle Protocol.

4.3.1 SMTP over DTN Architecture

The purpose of implementing SMTP over DTN is to be able to communicate by emails with a peer connected to Internet. As Internet connectivity is not guaranteed, we send the SMTP data through the Bundle Protocol to a remote DTN server node which will be responsible forwarding the mail to a mail relay server on Internet. This node must of course be connected to Internet, but not the DTN client node. To identify SMTP bundles, we will use the value 1 as the "application type" in the application block (see 4.2.3).

Postfix [1] was chosen as for the local mail relay server.

4.3.2 SMTP Protocol Spoofing

Upon the exchange with the mail application, the DTN/SMTP interface prepares the data for the bundle, see Figure 4.6. The Figure A shows the conversion from SMTP to bundle while Figure B shows the conversion from bundle to SMTP.

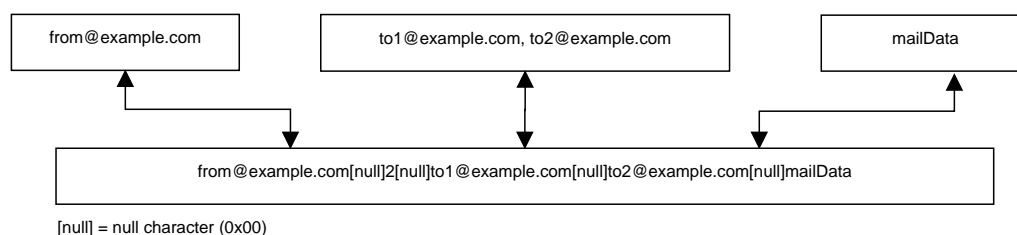


Figure 4.7. The SMTP Bundle Payload

The SMTP messages need to be packed up by the local SMTP server application into bundles before being transmitted with the Bundle Protocol. The SMTP/DTN interface takes three parameters from the mail client: the "from address", the "recipients" and the data containing the mail (header, subject, body, ...). In order to keep the proof-of-concept simple and the bundles as short as possible due to the limited storage capacity of the mobile devices, we concatenate the three parameters, separated by a null character (see Figure 4.7). As there might be several recipients, the list of recipients is also concatenated with a null character as a separator. The number of recipients is added to the front of the resulting string, again with a null character to separate both.

Note that the MX records (mail exchanger records) should be updated by the SMTP/DTN interface [39].

4.4 DTN Management

The objective here is to improve the DTN management system for better control and reliability. We want to be able to manage the bundles of a local DTN network. Along with management features, we also aim at improving DTN performances. For example, if there are not enough resources to carry the bundles onward another location, the bundles could be queued up until some resources become available. Priorities could also be given to bundles according to the type of application they serve.

The first question is should the management system be centralized or decentralized? Centralized means that all the bundles would go through a single node (the gateway) responsible for the management features. Decentralized means that each node will be responsible for managing its own bundles.

A centralized system is difficult to setup because of the mobile nodes. The nodes may spread apart and increase their distance to the gateway, thus increasing the delay for the bundles. However, for a close and small community such as a small village or a small group of nomad nodes stay together, this solution remains possible. Besides, it gives better control and flexibility for the management of the DTN network because of the global perspective. The system can make decisions based on

application type	application source	application destination
------------------	--------------------	-------------------------

Figure 4.8. The Application Block

the whole DTN network, such as setting priorities according to the overall traffic. Finally, it's easier to concentrate the resources like storage capacity on a single gateway than having them distributed over all the nodes.

A decentralized system is easier to setup in a DTN environment. For a management system, the nodes could be managed with a protocol similar to SNMP [15] but designed for DTN. However, the nodes would have a limited perspective compared to a gateway.

Because Bytewalla was first developed for small villages in rural areas in Africa, and because the first solution brings advantages that the second solution can hardly offer, the author of this thesis decided to go for the centralized management system.

Figure [x] gives an overview of the management system. All the bundles travel through the gateway before leaving or entering the village. The gateway process each bundle before it is transmitted to the next node.

In order to transmit all application bundles through the gateway, the bundle's destination is set as the gateway EID. However the gateway then needs to know where to send the bundle after it has been processed. Hence, two fields are added to the Application Block to keep track of the application bundle's source and the application bundle's destination. The two fields are the application data unit (ADU) source and the ADU's destination. The first one identifies where the application ADU is sent from while the second one identifies to whom it is sent. On the DTN level, the bundle will first be sent to the gateway which will then retransmit it to the ADU's destination after it has been processed. The bundle may go through other intermediary nodes but its final destination is always the ADU's destination. The final Application Block payload is shown in Figure 4.8. All fields are separated by a null character (0x00).

The gateway is not mandatory (the bundles may be send directly to their final destination) but it brings new features and can to improve the performances.

When the DTN Management gateway is used as an intermediary for all the bundles, it offers a couple of functionalities. For example, all the bundles may be stored on the gateway for future retransmission. Whenever a bundle transits through the gateway, the gateway keeps a copy and requests for a delivery status report. If the delivery status report isn't received within a timeout frame or if it receives a deletion status report, the bundle is resent if its lifetime hasn't expired.

Moreover the gateway can furnish the administrators with all the statistics. With the help of status reports, an administrator is aware of which bundles could

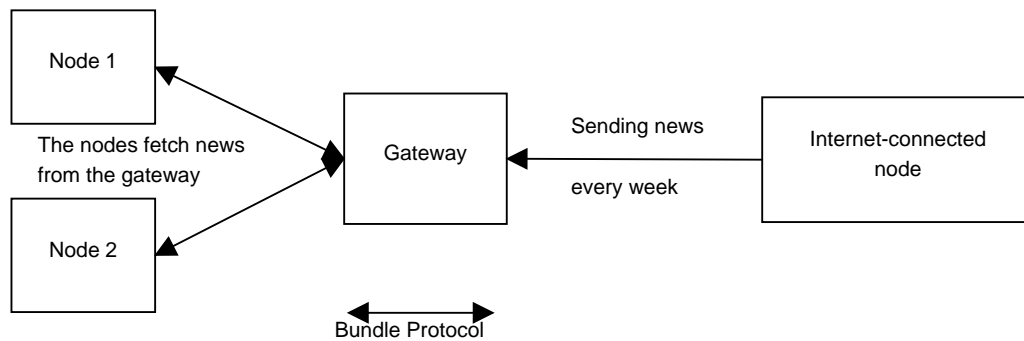


Figure 4.9. Subscription Service

be delivered successfully or failed to do so.

When there is only a little capacity for transmitting bundles to their destination, the gateway may decide on which bundles to deliver first based on their priority. The priority may be based on the type of applications the bundle serves, on its size, etc. As we can now learn about the type of data the bundles contain, this becomes an easy task.

Finally, it can also offer features like caching and subscription service. As in Figure 4.9, the gateway could for example request and store a single copy of a record, and make it available for all the nodes, instead of having each node requesting its own copy. A useful application would be for sending news or educational content.

Chapter 5

Implementation

The Implementation chapter is divided into three sections. First we introduce the reader to the development approach. The second section concerns the implementation of the queuing mechanism. We discuss about the possible queuing policies, the design of the queuing mechanism in Bytewalla and its implementation process. Finally, the third section focuses on the Application Layer. There we thoroughly present the technical challenges and how they are solved.

5.1 Software Development Approach

For the implementation, the author chose to follow the Evolutionary Prototyping Model [40]. This model is an incremental approach. The system is developed in increments so it can easily be modified according to its results and to follow the objectives of this thesis.

The implementation was done with Python 2.6 [22] and it only aims to be a proof-of-concept, not a robust, optimized and definitive implementation. Especially because of the limited time allocated to the implementation.

MySQL[5] is used as the database management system.

5.2 Queuing Mechanism

5.2.1 Design in Bytewalla

The purpose of implementing queuing mechanism in Bytewalla is to keep the total size below the specified quota. Hence, every time a bundle would be stored on the disc, we would maintain the quota by deleting bundles if necessary and according to priority order depending on the queuing policy.

Overview

The core resides in the ProphetQueuing class. It is an abstract class and it

provides common functions such as `getInstance`, `maintainQuota`, `delete` and `removeNextBundle`. The queuing policies are implemented in their own file, inherit from `ProphetQueuing` and have only one function “`getLastBundle`”. This function which is unique to each policy returns the last bundle id according to their priority in the queuing policy. For example the FIFO policy will return the id of the bundle which was added at first.

Small modifications were made to the `DTNConfigurationParser` and `BundleStore` classes. Respectively to add a policy setting in the configuration file and to give the handle to the queuing policy when it comes to maintain the quota after a new bundle was stored on the disc.

Configuration

The user can specify the queuing policy that he wants to use in the configuration file. This is achieved in the `DTNConfigurationParser` by simply adding a new setting “`Queuing_policy`”.

Storage and maintaining the quota

Inside the `BundleStore`’s `add` function, after the bundle has been stored and if the type of router being used is `PRoPHET`, we give the handle to the `PRoPHET`’s queuing mechanism. This checks if the quota has not been exceeded, and if this is the case, delete as many bundles as needed to free enough space.

5.2.2 Queuing Policies

Two queuing policies have been implemented. “First In First Out” (FIFO) and “Evict most forwarded first” (MOFO).

First In First Out

This policy has been implemented in the `Fifo` class. The `getLastBundle()` function simply takes the smallest id (the oldest one) from the database and returns it.

Evict Most Forwarded First

This policy has been implemented in the `Mofo` class. The `getLastBundle()` function simply returns the id of the bundle which has the greatest `forwarded_times` which contains the number of times the bundle has been forwarded.

This is very similar to the `Fifo`’s `getLasBundle()` function, however, we also need to keep track of the number of times the bundle has been forwarded.

This is handled inside the `BundleDaemon` class. There we simply increment the `forwarded_times` field for each bundle being transmitted.

5.3 Applications over DTN

The DTN-to-Application interface has only been implemented in Python on Ubuntu. The Application-to-DTN interface has been implemented both on Ubuntu in Python and on Android in Java. The reason for developing the DTN-to-Application on Ubuntu is that the server will usually be running on a immobile node connected to Internet (for example to contact SMTP mail relays). Also it is quite resource consuming as many services may be running on the machine (MySQL, multiple types of application, etc.). The reason for developing the DTN application client both on Ubuntu and Android is that it was easier to test and create prototypes on a scripting language such as Python, and along with the development of the server on the same system. However, the final objective is to be able to use such applications on any type of devices, especially mobile ones. Hence it was decided to implement it on the Android phones as well with the Bytewalla application.

The whole implementation is divided into modules (python files) and they are described below, in two sections. One for the Python implementation and one for the Android implementation.

As a proof-of-concept it was chosen to implement SMTP over DTN, replacing the first solution implemented in Bytewalla 1. The components specific to the implementation of SMTP over DTN are also described along the general system implementation.

5.3.1 Ubuntu

5.3.1.1 Overall

The implementation on Linux is able to send application bundles and to receive and dispatch them. Hence there are both an Application-to-DTN interface and a DTN-to-Application interface. The following modules are used for the Application bundles transmission and for the DTN Management component (see 5.4).

Config.py	This is the configuration file. It includes settings such as the proxy TCP/UDP ports, the DTN daemon TCP/UDP ports and the EIDs.
Bundle.py	This module contains the Bundle class and functions to convert between raw data and bundles.
serviceBlock.py	This module contains the serviceBlock class which defines the Application Block (type of application, the application source and the application destination).
Sdnv.py	Handles the conversion between integers and Self-Delimiting Numeric Values (SDNV). SDNVs are used in the Bundle Protocol and PRoPHET.
DTNinterface.py	This module manages the transmission with the local DTN daemon. It receives the packed data, creates the application bundle and delivers it to the DTN daemon to be transmitted through the DTN network.
DTNproxy.py	This is the core file. It eavesdrops the communications between the local DTN node and the other nodes. This way it can capture the application bundles and let the other modules process them.
bundleHandler.py	This is where the application bundle goes through when being processed. If you want to add or remove a processing function, this is where to do so.
Reports.py	The reports.py module takes care of the status reports. It parses them and updates the records for the stored bundles.
statusreports.py	statusreports.py is used to parse status reports and instantiate them with the StatusReport class.
storage.py	This module handles the access to MySQL for storing and updating the bundles records.
Reliability.py	This tool may be used to resend, for example with a cron, the bundles which could not reach their destination.
service.py	This module contains information about the applications running on the machine, and dispatch the bundles to corresponding DTN-to-Application interface.

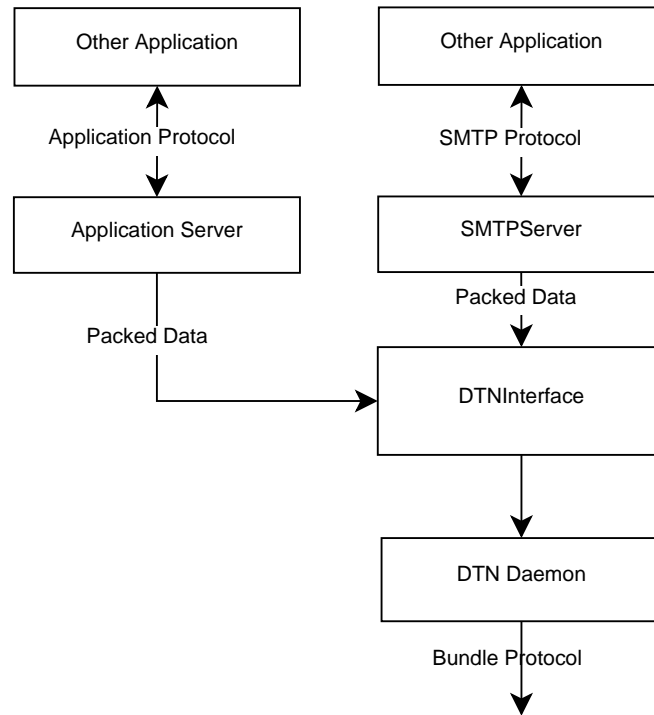


Figure 5.1. Data flow in the client

The Figure 5.1 shows the data flow in the client. The messages are received from the applications, packed in the applications servers and sent as bundles with the DTN Interface.

For capturing the application bundles upon reception, there are two principal solutions:

1. Modifying DTN2
2. Setting up a proxy application

DTN2 is available to everyone as an open-source application. It is written in C and could be modified to include new features. Setting up a proxy application consists of developing an additional application which would be responsible for capturing and processing the bundles before their reach the standard DTN daemon. The proxy can be developed in any language.

As the author is more familiar with scripting language such as Python, it was chosen to set up a proxy application in Python, rather than modifying the existing DTN2 project.

As in Figure 5.2, the proxy replaces the DTN2 daemon. It uses its ports (4556 and 9556) while the DTN2 daemon now runs with different ports (e.g. 4557 and 9447). Hence, the proxy receives and establishes connections with the other DTN nodes. Meanwhile, it establishes connections with the local DTN2. All the messages

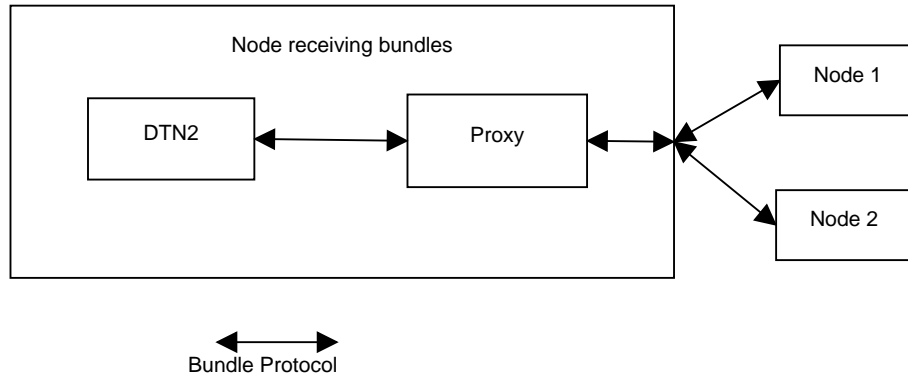


Figure 5.2. The DTN Proxy

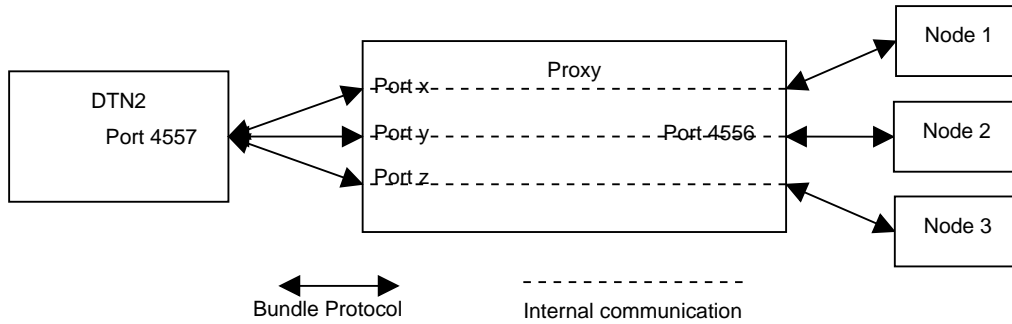


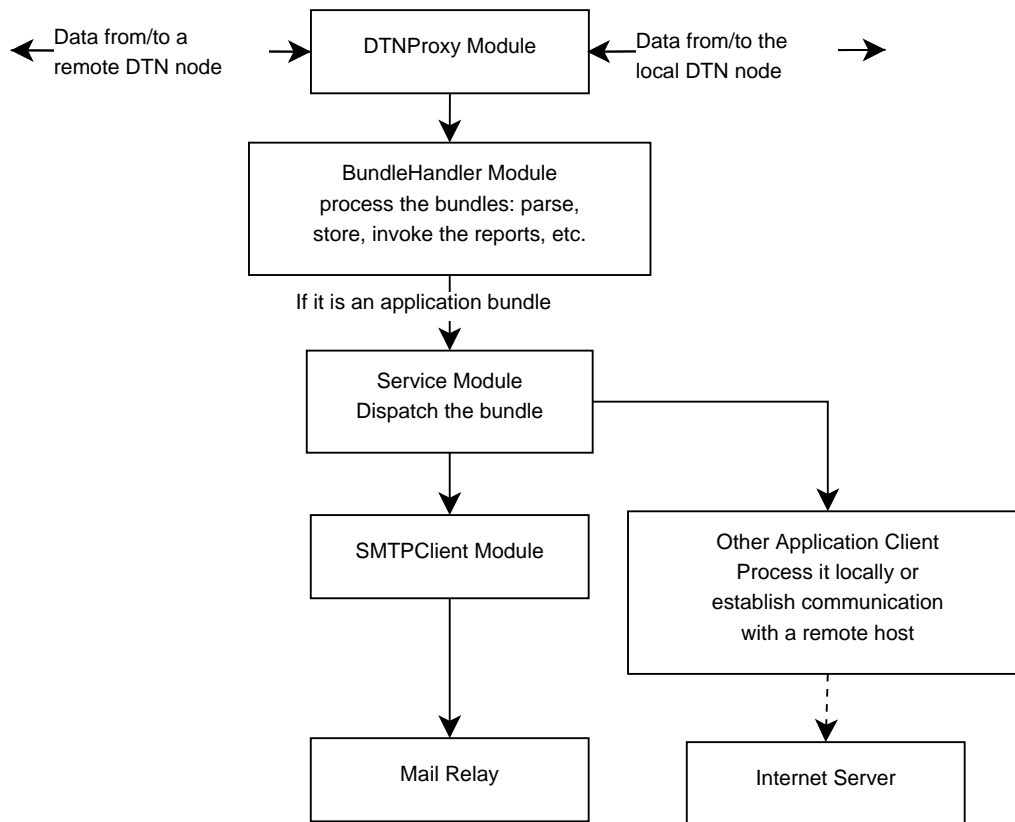
Figure 5.3. Closer view at the Proxy

sent from the other nodes are transmitted to the local DTN2 daemon and vice-versa. Hence, the proxy is transparent to the other nodes and to DTN2. In order to do so, the proxy associates each remote node IP address to a local port which is used to connect to the local DTN2 daemon. This way, the DTN2 establishes links with as many local ports as there are links to remote nodes. Figure 5.3 gives a closer view at the proxy.

The Figure 5.4 shows the data flow in the server (receiving application bundles). All application bundles are captured with the DTN proxy, processed by the bundle handler, and then dispatched to the right application [27].

5.3.1.2 Case: SMTP over DTN

To support SMTP over DTN, only two files are required on addition to the the general purpose modules. These two files are the Application-to-DTN interface and the DTN-to-Application interface.

**Figure 5.4.** Data flow in the server

SMTPServer.py It is the Application-to-DTN interface for SMTP. This module runs a SMTP server which handles the communications with the local SMTP client. After the exchange has been processed, it packs the data and sends it to the `dtinterface.py` module.

smtpclient.py It is the DTN-to-Application interface for SMTP. This module handles the SMTP application bundles. It recreates the messages out of the bundles it receives.

5.3.2 Android

5.3.2.1 Overview

For the Android implementation we are using the Bytewalla application which is responsible for bundling the data and send it through the DTN network. However, the Bytewalla 3 application does not support the application block as presented

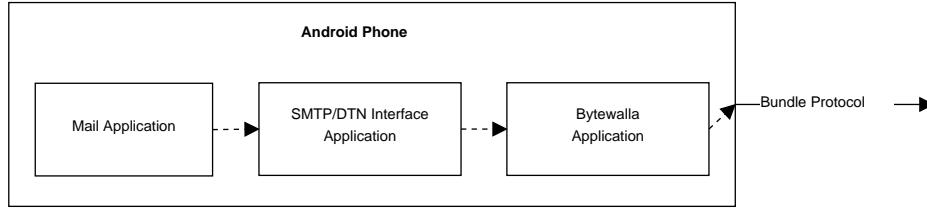


Figure 5.5. Overview of the implementation on Android

in the Chapter 4 and does not provide an interface to the other applications for sending bundles.

Hence the implementation is divided in two parts. How to provide an interface in Bytewalla for sending bundles from other applications, and the development of the application handling SMTP messages.

So finally, there will be three components:

1. **The SMTP client - The default Android mail application**

It is a regular mail application. It must be configured to use the SMTP/Bundle interface as the SMTP server.

2. **The SMTP/Bundle interface application**

It receives the message from the mail application, handles the messages, packs the data, and forward them to the Bytewalla application for the Bundle Protocol.

3. **The Bytewalla application**

It handles the Bundle Protocol.

When running, the system can be represented as in Figure 5.5.

The SMTP/DTN interface application communicates with the Bytewalla by the help of Android broadcasts. The interface application sends a message to the Android system along with some parameters such as the packed data (payload) and the type of service (1 for SMTP). Then the Bytewalla application catches the message and process it.

5.3.2.2 Modifications to the Bytewalla application

All the files added to the Bytewalla as part of this thesis are stored together in the package "se.kth.ssvl.tslab.bytewalla.androiddtm.applications"

Receiving data from other applications

In order to perform this task, we must add a declaration to the Bytewalla application's manifest file. This declaration (see 5.1) informs the Android system about what messages the Bytewalla application is ready to handle.

Listing 5.1. Bytewalla Manifest Declaration

```
<receiver android:name=".applications.PayloadReceiver">
```

```

<intent-filter>
  <action android:name="se.kth.ssvl.tslab.bythewalla.androiddtn.
action.SEND_BUNDLE" />
</intent-filter>
</receiver>

```

The declaration also indicates which class is responsible to handling the message. In our case, the PayloadReceiver does it. Once received, a new service (see [27]) defined in the ProcessPayload class is started to process the message and send it through the Bundle Protocol. The ProcessPayload service defines the bundle specification and ask the Bythewalla application to send the bundle.

Application Block

The Bythewalla 3 application only supports the primary and the payload block. Hence the application block has been integrated with the ApplicationBlockProcessor class.

5.3.2.3 The SMTP/DTN interface application

For this application I have used the SubEtha SMTP library [30]. This library allows the application to receive SMTP mail with a simple, easy-to-understand API.

The application listens to a specific port for new mails. For each processed mail, the application packs up the data as shown in Subsection 4.3.2 and send it to the Bythewalla application along with the type of service.

5.4 DTN Management

The DTN management tools have only been implemented on Ubuntu. It shares the same modules as in 5.3.1.1. Most of the processing such as storing bundles and processing status reports is done in the bundleHandler.py module.

Along with this modules comes a web-based administration interface.

The administration interface is developed in PHP and runs on a HTTP Server such as Apache. The application consists of modules. Each module has its own specific task.

4 modules have been implemented.

1. **bundles.php**: This module shows the application bundles which passed through the gateway. It provides information like the bundle creation date, its source, its payload and whether it was delivered or not.
2. **config.php**: The config.php module helps to generate a new configuration file for the local DTN daemon.

3. **default.php**: This is the default module, which only refers to the other modules.
4. **stats.php**: The statistics module gives an overall information about the gateway. For example the number of stored bundles, the number and percentage of bundles delivered and undelivered, the average delay, and so on.

Chapter 6

Testing & Analysis

A series of tests were carried out on the implementation to measure the bundle transmission delay with and without the Application Layer optimization between two connected nodes. Based on the results we discuss whether the proxy adds significant delays. Although DTN is by definition tolerant to delay, a long transmission delay on a link between nodes could reduce the amount of data two nodes can exchange before the connection is disrupted. Hence it is important to ensure that our work does not add significant delay.

6.0.1 Test environment

The test environment consists of two servers running DTN2. The two servers are connected via Ethernet. Their specification is shown in Table 6.1 and the setup in Figure 6.1.

Table 6.1. Specification of the DTN2 servers

CPU Intel Celeron	1.8 GHz
System Memory	512 MB ECC DDR
Total Hard Disk Size	40 GB
Operating System	Linux Ubuntu
DTN Software DTN2	version 2.6

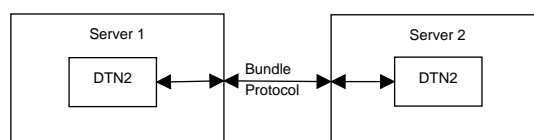


Figure 6.1. test Environment Setup

6.0.2 Methodology

The bundles are sent from a machine to the other one. The later one runs the Application/DTN proxy. It will process the bundle before it is received by DTN2. The time for this processing will be measured.

For the processing we will only perform the following steps:

1. Capture the bundles,
2. Parse them,
3. And forward them to DTN2.

Whenever an application bundle would be processed by a local application, it would run asynchronously to the proxy, so this would not add delay to the transmission.

6.0.3 Measurements

In the Table 6.2, we measure the time elapsed from the moment the data is sent to the proxy, to the moment the proxy forwards the data to the peer node. We made ten simulations runs from which we took the average.

Table 6.2. Time elapsed in the proxy processing

Measure Number	Time Elapsed (ms)
1	1.204
2	1.068
3	1.244
4	1.183
5	1.189
6	1.197
7	1.292
8	1.163
9	1.189
10	1.084
Average	1.181

6.0.4 Observations and Summary

When transmitting only a few bundles as in most situations, the additional processing delay would not be very significant. However in a situation where many bundles are transmitted (e.g. >100) and the disruptions regular, this delay could become significant.

Nevertheless, the implementation may easily be much improved. It is recommended to integrate the application bundles within the dtn daemon in order to reduce the processing involved in internal bundles transmission.

Chapter 7

Conclusion

7.1 Summary

The PROPHET queuing mechanism has been implemented in the Bytewalla application. Two queuing policies, FIFO and MOFO, were chosen according to their efficiency evaluated by Anders Lindgren and Kaustubh S. Phans [31]. These two queuing policies are now part of the Bytewalla application and were tested in the Chapter 6. We saw how they help managing the bundles when the storage consumption becomes too high.

This thesis also replied to some of the issues met when developing applications over DTN. The implementation includes some tools developed with Python. As it has been experienced with SMTP, applications can now transparently, except for delays, communicate over DTN. The application known transmit its requests all-at-once instead of iteratively as it uses to be in Internet. Also, the whole infrastructure has been designed to support more services such as caching with the help of the intermediary servers. The later also support management tools for a better control of the network. The management tools also help to improve the reliability as the bundles are retransmitted until an acknowledgment is returned.

In the Chapter 6, we verified that the implementation was working according to the objectives and that they do not worsen the quality of service. However, it would certainly be interesting to test it in real-world situations.

7.2 Future Work

The PROPHET implementation in Bytewalla is not complete yet. For example, it would be an improvement to implement the Forwarding Strategies mechanism. While the queuing mechanism aims to order the bundles by order of priority for deletion when the storage consumption becomes too high, Forwarding Strategies aim to order the bundles by the order in which they should be transmitted to another mule. The connection between two mules may be disrupted at any time,

and so, the "most important" bundles should be transmitted first.

This thesis is a starting point for developing more tools and applications over DTN. Here, we have only integrated an mail application for testing the concept. Now, it would be interesting to integrate popular and useful applications for specific situations. For example, twitter and youtube for the people where the access to Internet has been shutdown. For education purposes in remote areas, or for mails delivery, it would be useful to have a subscription service implemented on top of this thesis work. This only requires to develop a client and a server application, running on a regular basis to send the most updated data. Caching may also be implemented in a generic fashion as well as by means of application-specific modules to provide better access to the content which was delivered recently in the area.

Appendix A

Testing FIFO and MOFO

For these tests we are using the final version of Bytewalla IV developed along this thesis. The application is running on a HTC Wildfire with Android 2.2.1. We have reduced the storage capacity to only 32Kb so that we reach the maximum storage capacity and trigger the queuing mechanism more easily.

It is difficult to simulate a large DTN network and the encounters, especially with the Android platform, as no tools are available for this purpose. However some simulations have already been conducted to measure the queuing mechanism performances in the paper from Lindgren, A. and K. Phanse [31]. Hence, here, we simply verify that the queuing mechanism implementation is working as it is supposed to.

We test both queuing policies FIFO and MOFO. The FIFO policy deletes the first bundle that was stored on the node. The MOFO policy deletes the bundle which has been forwarded the most. If several bundles have been forwarded the same number of times, it will use a FIFO policy to delete a bundle among the bundles which have been forwarded the most. For each policy we have 20 iterations. For each iteration we create a new bundle (whose id corresponds to the iteration where it was created) and check the occupied disk space after the bundle has been created. The bundles are created manually through the Bytewalla DTNSend application. Whenever a bundle is deleted, we check its attributes such as its ID and the number of times it was forwarded. Hence, we can ensure that the queuing mechanism is deleting the bundles according to the policy.

The storage consumption is retrieved from the Bytewalla application interface. The bundle attributes are retrieved from the Android debugger tool.

The Table A.1 shows that the FIFO queuing mechanism starts deleting bundles when the quota (32KB) gets exceeded by the storage consumption. At this point, each time a bundle is added, another one is deleted, thus, maintaining the storage consumption below the specified quota. As the FIFO policy requires to delete the oldest bundle first, we can notice that the queuing mechanism starts by deleting the oldest bundle (bundle 1), the the bundle 2, and so on.

Table A.1. Storage consumption with the FIFO queuing policy

Iteration	Storage Consumption (MB)	Number of bundles	Deleted bundle's id
0	0.0059	0	-
1	0.0097	1	-
2	0.0136	2	-
3	0.0175	3	-
4	0.0213	4	-
5	0.0252	5	-
6	0.0291	6	-
7	0.0329	7	-
8	0.0329	7	1
9	0.0329	7	2
10	0.0329	7	3
11	0.0329	7	4
12	0.0329	7	5
13	0.0329	7	6
14	0.0329	7	7
15	0.0329	7	8
16	0.0329	7	9
17	0.0329	7	10
18	0.0329	7	11
19	0.0329	7	12
20	0.0329	7	13

The bundles listed in the Table A.3 have been created and forwarded accordingly before the other bundles (from iteration 6) were created according to the Table A.2.

Similar as the FIFO policy, in the Table A.2, the MOFO policy starts deleting the bundles when the storage consumption reaches the quota limit. However, the order of deletion is different. The Table A.3 shows how many times each bundle has been forwarded. The MOFO policy deletes the bundles by descending order of "forwarded times". Hence, the first bundle to be deleted is the bundle 1, then 3, 2, 5 and 5. From this point, it deletes the bundles exactly like the FIFO policy because all the bundles have been forwarded 0 times.

Table A.2. Storage consumption with the MOFO queuing policy

Iteration	Storage Consumption (MB)	Number of bundles	Deleted bundle's id
...	
5	0.0252	5	-
6	0.0291	6	-
7	0.0329	7	-
8	0.0329	7	1
9	0.0329	7	3
10	0.0329	7	2
11	0.0329	7	5
12	0.0329	7	4
13	0.0329	7	6
14	0.0329	7	7
15	0.0329	7	8
16	0.0329	7	9
17	0.0329	7	10
18	0.0329	7	11
19	0.0329	7	12
20	0.0329	7	13

Table A.3. Number of forwarded times for each bundle

Bundle ID	Forwarded Times
1	3
2	2
3	3
4	0
5	1

Bibliography

- [1] Postfix. <http://www.postfix.org/> accessed June 10th, 2011.
- [2] A. Doria A. Lindgren and O. Scheln. Probabilistic routing in intermittently connected networks. In *Proceedings of the Fourth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2003)*, 2003.
- [3] E. Davies S. Grasic A. Lindgren, A. Doria. Probabilistic routing protocol for intermittently connected networks. Internet-Draft, October 2010. <http://tools.ietf.org/html/draft-irtf-dtnrg-prophet-08> accessed June 10th, 2011.
- [4] A. Hasson A. Pentland, R. Fletcher. Daknet: Rethinking connectivity in developing nations, 2004. http://www.firstmilesolutions.com/documents/DakNet_IEEE_Computer.pdf accessed June 10th, 2011.
- [5] MySQL AB. Mysql. <http://www.mysql.com/> accessed June 10th, 2011.
- [6] Abdullah Azfar. Installation of postfix and configuring village and city dns servers. Technical report, October 2009. http://www.tslab.ssvl.kth.se/csd/projects/092106/sites/default/files/Postfix_and_DNS.pdf accessed June 10th, 2011.
- [7] Abdullah Azfar. Integration of postfixwith dtn2. Technical report, November 2009. http://www.tslab.ssvl.kth.se/csd/projects/092106/sites/default/files/Postfix_DTN2_Integration.pdf accessed June 10th, 2011.
- [8] et al. Berners-Lee. Uniform resource identifier (uri): Generic syntax. RFC 3986, January 2005. <http://tools.ietf.org/html/rfc3986> accessed June 1st, 2011.
- [9] Ellard & Brown. Dtn ip neighbor discovery (ipnd). Internet-Draft, March 2010. <http://tools.ietf.org/html/draft-irtf-dtnrg-ipnd-01> accessed June 1st, 2011.
- [10] Scott & Burleigh. Bundle protocol specification. RFC 5050, November 2007. <http://tools.ietf.org/html/rfc5050> accessed June 10th, 2011.
- [11] KTH Bytewalla 3. Network management tool. Technical report, October 2010. <http://www.tslab.ssvl.kth.se/csd/projects/1031352/content/network-management-tool> accessed June 10th, 2011.

- [12] KTH Bytewalla 3. Sentinel surveillance application. Technical report, October 2010. <http://www.tslab.ssvl.kth.se/csd/projects/1031352/content/sentinal-surveillance-application> accessed June 10th, 2011.
- [13] KTH Bytewalla I, TSLab. Bytewalla: Delay tolerant network on android phones. <http://www.tslab.ssvl.kth.se/csd/projects/092106/> accessed June 1st, 2011.
- [14] B. Carpenter. Architectural principles of the internet. RFC 4838, June 1996. <http://tools.ietf.org/html/rfc4838> accessed June 1st, 2011.
- [15] J. Case. A simple network management protocol (snmp). RFC 1157, May 1990. <http://www.ietf.org/rfc/rfc1157.txt> accessed June 10th, 2011.
- [16] et al. Cerf. Delay-tolerant network architecture: The evolving interplanetary internet. RFC 4838, August 2002. <http://www.ipnsig.org/reports/draft-irtf-ipnrg-arch-01.txt> accessed June 1st, 2011.
- [17] Mozilla Corporation. Thunderbird. <http://www.mozillamessaging.com/fr/thunderbird/> accessed June 10th, 2011.
- [18] Sebastian Domancic. Security in Delay Tolerant Networks for the Android Platform. Master's thesis, Royal Institute of Technology (KTH) & Aalto University - School of Science and Technology (TKK), Stockholm, 2010.
- [19] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages pp. 27–34, NY, USA, 2003. ACM New York.
- [20] et al. Farrell. Delay-tolerant networking security overview. Internet-Draft, March 2009. <http://tools.ietf.org/html/draft-irtf-dtnrg-sec-overview-06> accessed June 1st, 2011.
- [21] Consultative Committee for Space Data Systems. About ccscs. <http://public.ccsds.org/about/default.aspx> accessed June 1st, 2011.
- [22] Python Software Foundation. Python. <http://www.python.org/> accessed June 10th, 2011.
- [23] N. Freed. Multipurpose internet mail extensions (mime) part five: Conformance criteria and examples. RFC 2049, November 1996. <http://www.ietf.org/rfc/rfc2049.txt> accessed June 10th, 2011.
- [24] N. Freed. Post office protocol - version 3. RFC 1939, May 1996. <http://tools.ietf.org/html/rfc1939> accessed June 10th, 2011.
- [25] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. RFC 2045, November 1996. <http://www.ietf.org/rfc/rfc2045.txt> accessed June 10th, 2011.

- [26] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. RFC 2046, November 1996. <http://www.ietf.org/rfc/rfc2046.txt> accessed June 10th, 2011.
- [27] Google. Android service. <http://developer.android.com/reference/android/app/Service.html> accessed June 10th, 2011.
- [28] Delay Tolerant Networking Research Group. About - delay tolerant networking research group. <http://www.dtnrg.org/wiki/About> accessed June 1st, 2011.
- [29] Delay Tolerant Networking Research Group. Code. <http://www.dtnrg.org/wiki/Code> accessed June 1st, 2011.
- [30] J. Schnitzer I. McFarland, J. Stevens. Subetha smtp is an easy-to-use server-side smtp library for java. <http://code.google.com/p/subethasmtplib/> accessed June 10th, 2011.
- [31] A. Lindgren and K. Phanse. Evaluation of queueing policies and forwarding strategies for routing in intermittently connected networks. In *Proceedings of COMSWARE 2006*, January 2006.
- [32] Ming-te Lu. Digital divide in developing countries. *Journal of Global Information Technology Management*, 4:3:1–4, 2001.
- [33] K. Moore. Mime (multipurpose internet mail extensions) part three: Message header extensions for non-ascii text. RFC 2047, November 1996. <http://www.ietf.org/rfc/rfc2047.txt> accessed June 10th, 2011.
- [34] J. Klensin N. Freed and J. Postel. Multipurpose internet mail extensions (mime) part four: Registration procedures. RFC 2048, November 1996. <http://www.ietf.org/rfc/rfc2048.txt> accessed June 10th, 2011.
- [35] N4C. N4c homepage. <http://www.n4c.eu/Home.php> accessed June 1st, 2011.
- [36] Kruse & Ostermann. Udp convergence layers for the dtn bundle and ltp protocols. Internet-Draft, November 2008. <http://tools.ietf.org/search/draft-irtf-dtnrg-udp-clayer-00> accessed June 1st, 2011.
- [37] Demmer & Ott. Delay tolerant networking tcp convergence layer protocol. Internet-Draft, February 2008. <http://tools.ietf.org/html/rfc5050> accessed June 1st, 2011.
- [38] Jörg Ott. Application protocol design considerations for a mobile internet. In *1st ACM MobiArch Workshop, San Francisco*, December 2006.
- [39] R. Bush R. Elz. Clarifications to the dns specification. RFC 2181, July 1997. <http://tools.ietf.org/html/rfc2181> accessed June 10th, 2011.
- [40] Construx Software. Evolutionary prototyping, May 2002. <http://www.construx.com/File.ashx?cid=814> accessed June 10th, 2011.

- [41] Internet World Stat. World internet users and population stats. <http://www.internetworldstats.com/stats.htm> accessed June 1st, 2011.
- [42] et al. Symington. Bundle security protocol specification. Internet-Draft, November 2009. <http://tools.ietf.org/html/draft-irtf-dtnrg-bundle-security-10> accessed June 1st, 2011.
- [43] Amin Vahdat and David Becker. Epidemic routing for partially connected ad hoc networks. Technical report cs-2000-06, Department of Computer Science, Duke University, April 2000. <http://tools.ietf.org/html/rfc5050> accessed June 1st, 2011.