AALTO UNIVERSITY

School of Science and Technology

Faculty of Electronics, Communications and Automation

Department of Communications and Networking

Petri Ylikoski

Applying DTN to Mobile Internet Access: a Case Study

Thesis submitted in partial fulfillment of the requirement for the degree of Master of Science in Technology

Espoo, Finland, 10.5.2010.

Supervisor	Prof. Jörg Ott
Instructor	Prof. Jörg Ott

AALTO-YLIOPISTON TEKNILLINEN KORKEAKOULU | **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Petri Ylikoski		
Työn nimi:	Applying DTN to Mobile Internet Access: a Case Study		
Päivämäärä:	10.5.2010	**Sivumäärä:**	60
Tiedekunta:	Elektroniikan, tietoliikenteen ja automaation tiedekunta		
Laitos:	T4070 Tietoliikenne- ja tietoverkkotekniikan laitos		
Työn valvoja:	Prof. Jörg Ott		
Työn ohjaaja:	Prof. Jörg Ott		

Internetin mobiilikäyttö on yleistynyt voimakkaasti. Internet-protokollat on kuitenkin kehitetty kiinteän verkon viestintää varten ja niiden suorituskyky, erityisesti TCP:n, kärsii olosuhteissa, joissa kiinteää yhteyttä verkkoon ei ole saatavilla. EU-tutkimusprojekti nimeltä CHIANTI perustettiin tutkimaan mahdollisuutta paremman suorituskyvyn tarjoamiseksi mobiilikäyttäjille. Sen pyrkimyksenä on kehittää tietoliikenneratkaisu, jossa välityspalvelimet suojaavat käyttäjiä verkkoyhteyden katkoksilta.

DTN on tietoliikennearkkitehtuuri joka on kehitetty viestinvälitykseen vaativissa olosuhteissa, esim. avaruusviestinnässä, ja mahdollistaa viestien välityksen pitkien viiveiden ja katkonaisten verkkoyhteyksien yli.

Diplomityöni tarkoitus oli selvittää, voitaisiinko CHIANTI-projektin mukaiset välityspalvelimet toteuttaa hyödyntäen DTN-tutkimusryhmän kehittämää DTN-sovellusta. Työtä varten olen kehittänyt ja toteuttanut yksinkertaisen protokollan, jolla voidaan välittää HTTP-pääteyhteyksiä kahden DTN-solmun kautta. Protokollatoteutuksen avulla voidaan mitata DTN-toteutuksen suorityskykyä ja sitä kautta arvioida sen soveltuvuutta CHIANTI-projektin kannalta. Tätä varten mitattiin DTN-toteutuksen tiedonsiirtokapasiteettia sekä sen aiheuttamaa lisäviivettä HTTP-tiedostonsiirtoihin.

Mittaustulokset osoittivat, että DTN-toteutus pystyy vain rajalliseen tiedonsiirtoon, suurin mitattu siirtonopeus oli vain noin 1,5 megatavua sekunnissa ja kaikissa tapauksissa DTN:n käyttö lisäsi yhteysviivettä yli 100 millisekunnilla.

Tulosten valossa työssä todetaan, että tarkasteltu DTN-toteutus on hieman rajallinen suorituskyvyltään mutta silti käyttökelpoinen ja omaa potentiaalia jatkokehitykseen.

Avainsanat: CHIANTI, disconnection tolerance, DTN, HTTP

AALTO UNIVERSITY **ABSTRACT**
SCHOOL OF SCIENCE AND TECHNOLOGY

Author:	Petri Ylikoski
Name of the Thesis:	Applying DTN to Mobile Internet Access: a Case Study
Date:	10.5.2010 **Number of pages:** 60
Faculty:	Faculty of Electronics, Communications and Automation
Department:	Department of Communications and Networking
Supervisor:	Prof. Jörg Ott
Instructor:	Prof. Jörg Ott

Mobile use of Internet is increasing rapidly. Internet-protocols, in particular TCP, have been designed for operation with fixed connections and perform poorly in conditions of intermittent connectivity. CHIANTI is an EU-funded research project established to offer better performance for mobile Internet users.

DTN is a communications architecture that has been developed to enable communications over long delays and intermittent connectivity, such as in space communications.

The purpose of this work is to investigate applicability of the reference DTN implementation developed by the DTN Reseach Group to the needs and aims of CHIANTI. For this purpose I have developed a simple protocol to relay endpoint HTTP connections over a DTN link in order to be able to measure DTN performance and assess its usefulness for CHIANTI purposes. To this end, throughput capacity and delay caused by DTN are measured.

Results of measurements indicate limited throughput performance of around 1.5 megabytes per second and over 100 millisecond additional delay to endpoint communications even in best cases.

In light of attained results this work concludes that the DTN implementation used in this work has limited performance but could still prove useful, and has potential for further development.

Keywords: CHIANTI, disconnection tolerance, DTN, HTTP

Foreword

As far as I am able to tell, the actual process behind this master’s thesis was not particularly long or arduous as such. However, for me the process gradually leading to the point where I was finally able to begin working with it was much more difficult. Having finally reached the culmination of my studies and facing imminent graduation, I feel with all the more reason that a sentiment of appreciation is not entirely inappropriate here.

I owe a debt of gratitude to Professor Jörg Ott, the supervisor and instructor of this thesis, for finding time to personally guide me through the entire process of creation. While it could be argued that this is something he is supposed to do as a professor, it is an altogether different matter to recognize that he has done so with inexhaustible patience and with a positive, encouraging and constructive manner despite pressing work load. For guiding me through several courses, a special assignment and finally this thesis, always with a friendly smile, he has earned my gratitude and respect.

By far the greatest and most important pillar of support for me has, however, been my dearly beloved wife, who for the last ten years has always given me unwavering and loving support where needed, and a proverbial but swift kick to the posterior when necessary. Today, I would not be here without her by my side.

Espoo 10.5.2010

Petri Ylikoski

Table of Contents

Illustrations

List of Tables

Abbreviations and Acronyms

2.5G	Packet switching technology added to 2nd generation mobile networks like GSM. Also known as General Packet Radio Service, GPRS
3G	3rd Generation Mobile Telecommunications, also IMT-2000
ACK	Acknowledgement
API	Application Programming Interface
CHIANTI	Challenged Internet Architecture Network Technology Infrastructure
DHARMA	Distributed Home Agent for Robust Mobile Access
DNS	Domain Name System
DTN	Delay-Tolerant Networking
DTNRG	→DTN Research Group
EU	European Union
GNU	GNU’s Not Unix, a software project to develop free software, also operating system
GSM	Global System for Mobile Telecommunications
HIP	Host Identity Protocol
HTTP	HyperText Transfer Protocol
I/O	Input / Output
IBR	Institute of Operating Systems and Computer Networks at the Braunschweig Technical University
ICT	Information and Communication Technologies
IEEE	Institute of Electrical and Electronics Engineers
ISC	Internet Systems Consortium
IP, IPv4, IPv6	Internet Protocol, version 4 or 6
Kbps	Kilobits per second
KB, KB/s	Kilobyte, kilobytes per second. A kilobyte = 1000 bytes
KiB	Kibibyte, 1024 bytes
LAN	Local Area Network
Mbps	Megabits per second

MB, MB/s	Megabyte, megabytes per second. A megabyte = 1000 →kilobytes
MiB	Mebibyte, 1024 kibibytes
ms	Millisecond
NAT	Network Address Translation
OCMP	Opportunistic Connection Management Protocol
OS	Operating System
PCMP	Persistent Connection Management Protocol
RFC	Request For Comments
SOCKSv5	SOCKS version 5. SOCKS (SOCKetS) is a network protocol developed to assist communication through firewalls
SYN	SYNchronize sequence numbers, a →TCP Profocol flag
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless →LAN
x86	A processor architecture originally developed by Intel
XML	Extensible Markup Language

1. Introduction

The end of the second millennium saw the introduction of groundbreaking new ways of communication: the Internet and mobile telephony.

The Internet, with its powerful, evolving infrastructure and proliferation of personal computing resources and innovative applications and protocols, has provided us a way of sharing vast amounts of information as well as become a platform for new, previously unimaginable services and possibilities. The first ever Millennium Technology Prize was awarded in 2004 to Tim Berners-Lee, the founding father of the World Wide Web, as a reflection and recognition of the profound effect it has had on the society.

Mobile telephony, with the introduction of the GSM communication standard and proliferation of inexpensive, hand-held mobile telephones made possible by the advances in microelecronics and computing, now allows us to communicate with each other with flexibility and convenience unseen ever before.

A natural idea for further development is combination of Internet and mobility, and indeed it has been the subject of fervent research, as service providers have been rushing to provide mobile broadband to customers and mobile multimedia has been at the center of many a research conference. As anyone with experience in using a laptop while on the move can tell, there is still a long way to go before Internet services can be offered to mobile users with a degree of service comparable to Internet use through fixed cable networks. Moving out of range of a WLAN hotspot will interrupt connections and force users to restart application sessions – even if network connectivity with some other access technology existed. Overcoming such connectivity intermittence is one of the key challenges in mobile communications and correspondingly has spawned countless research projects focusing on challenged Internet access.

Moving outside of the Internet environment, space exploration, satellite communications and other more exotic and demanding network environments and communication scenarios have also given rise to different research fields.

Data transfer over extremely challenged and/or heterogeneous networks with little common technological ground have produced proposals such as Delay-Tolerant Networking. DTN has been developed for transmitting messages in a robust manner over difficult conditions, such as over links with extreme latency and intermittent connectivity, conditions where traditional Internet protocols fail or fare poorly. A key idea behind this work is that DTN technologies might have the potential to alleviate or even solve problems inherent in mobile Internet access as well.

A favourite Internet application today is the World Wide Web, widely used for business and pleasure alike with a user base in the hundreds of millions and thus the initial starting point for this work. This document investigates the possibility to utilize an existing reference implementation of Delay-Tolerant Networking software in order to provide at least a basis for disconnection-resilient communication environment for mobile Internet users. This document also describes the design and implementation of a simple communication protocol for relaying web session data over an unreliable communication link masked by the DTN, and assesses the performance of the DTN software to gain some insight on its suitability for the task.

For the purpose of assessing DTN performance, a set of simple measurements will be made to compare throughput and latency of HTTP traffic over a DTN link with corresponding measurements without the DTN software. Measurements aim to find out the data throughput capability of the DTN software, magnitude of the effect it has on latency for file transfers, and effects of different DTN-related parameters on both throughput and latency.

The rest of this document has been organized as detailed below.

Section 2 provides background information relevant for this work, especially explaining the DTN concept in more detail.

Section 3 concentrates on the design and implementation of an adaptation protocol to facilitate relaying HTTP traffic over a DTN link. It gives overview of the protocol design choices and functionality, as well as describing the actual protocol implementation.

Section 4 documents the test setup and procedures and measurements performed to test the performance and functionality of the protocol and DTN implementations.

Section 5 presents the results of testing and measurements and discusses their implications to evaluating the performance of the DTN technology.

Section 6 concludes the document, summarizing the findings of this work and gives recommendation for future work for improving on the concept.

2. Background

Heterogeneous wireless environments and the resulting intermittent connectivity experienced by mobile users is a widely recognized and copiously researched subject around the world. Section 2.1 is a brief introduction to some notable work on the subject. One culmination of this earlier research is the DTN concept, which forms a crucial part of this work and is explained in more detail in section 2.2. The CHIANTI project, more thoroughly discussed in chapter 2.3, is a recent, more practical development for improving resilience for connection disruptions. The work described in this thesis originated as an aside during the CHIANTI project as a possible alternative for implementing CHIANTI functionality using existing technologies, more specifically, using the implementation created by the DTN Research Group to provide improved disconnection tolerance to normal HTTP traffic.

2.1. Mobility and disconnectivity

The Internet and its core communication protocols, the TCP/IP suite, were designed for robust communication in a fixed network with static nodes. Internet protocol and application design was based on the end-to-end principle, formulated in [1]. In brief, the end-to-end principle states that many necessary functions in communication over an unknown, heterogeneous network can only be performed by the endpoints actually engaged in the dialogue. This is because the design of networks was at the time directed towards simple, minimal network which would efficiently provide the bare minimum of services – mostly routing and packet forwarding – and avoid replicating more extensive functionality both in the network and at higher protocol levels. The TCP protocol operation is a good example of the end-to-end principle in action; endpoint hosts running TCP/IP protocol stack employ TCP to take care of such necessary functions as flow control, out-of-order caching, acknowledgements, retransmitting missing packets and timing out connection. Following the end-to-end principle usually involves interaction between endpoints, which is not a problem, if they have a fixed, low-latency network connection available.

On the other hand, the GSM revolution of the early 1990’s triggered an increasing interest for mobile communications, exacerbated by the creation of other wireless technologies such as the IEEE 802.11 technology family (WLAN). By now, proliferation of different wireless access technologies has led to existence of wide variety of different computing and communication devices supplied with several different access technologies, ranging from copper-based Ethernet to V.90 and 3G modems.

Increasing demand for mobility combined with the development of Internet-based services, particularly of those related to multimedia, is increasingly bringing Internet and associated applications into mobile devices. It has also created demand for wireless broadband communications. In general, radio signals suffer from attenuation and poor signal to noise ratio, especially so at lower levels of transmitted power. Also, higher data rates demand higher carrier frequencies for more transmitted information per time unit, while higher frequencies demand more power to transmit and tend to attenuate faster than low frequencies, thus having shorter range than lower frequencies. Furthermore, mobile devices are often severely constrained in terms of size and thus available power, making economical use of energy an important design criterion. As a consequence, cell sizes used in different communication technologies tend to become smaller as data rate increases, and so, for users of high data rate mobile communication services connection disruptions are commonplace. For example, laptop user in a WLAN hotspot will eventually have to leave the coverage area of the WLAN base station, the radius of which is typically around ten to a hundred meters, and from there on will have to use other access technologies (e.g. 2.5G, 3G, WiMAX) for wireless communication.

As noted earlier, modern communication devices tend to be capable of communicating via more than one access technology. However, Internet applications and protocols quite often rely on TCP for establishing and maintaining endpoint connections. Implementations of TCP (and UDP) however rely on sockets as endpoint identities; sockets are bound to IP address – having one is a mandatory requirement for any entity wishing to communicate in the

Internet – which is very likely to change if user roams between different access networks. As a result communication context is lost, at least from the perspective of the TCP protocol. Even in the case of moving between different WLAN networks, coverage may be intermittent, and TCP (or the application in use) will quickly time out if it does not reach acknowledgements from the other endpoint, again losing all context and forcing the user to re-establish session.

Of course, several improvements have been suggested to either improve TCP's mobile performance or to circumvent it altogether with higher-layer approaches. Much work has also been done in trying to mitigate the effect of changing IP addresses. For instance, Mobile IP, as specified in RFC 3344 [2] and RFC3775 [3] for IPv4 and IPv6 respectively, uses home and foreign agents to keep track of mobile endpoints and enable continuous routing of packets to mobile endpoint via aforementioned agents even as IP addresses change, maintaining application and transport-layer (TCP) connection. However, this approach will not protect the user from disconnections.

A higher-layer protocol dubbed HIP [4] was proposed to separate IP's endpoint identifier and locator functionalities from each other, thus creating an identification technology better suited for mobility. To provide protection for connection outages, a proposal of combining HIP with some custom TCP enhancements has been made [5]. The TCP options, called User Timeout Option and Retransmission Trigger, would prevent TCP from timing out during outages and resume transmission as soon as connection becomes available, and by binding the TCP to HIP addresses instead of IP addresses, immunity from IP address changes is obtained. A similar solution is TCP Migrate [6], which modifies the TCP SYN packets and adds a new state into the protocol to protect it from disconnections and uses Dynamic DNS for protection from changing IP addresses.

These kinds of endpoint-oriented approaches have their own problems. For instance, modifying the TCP protocol in one endpoint generally has the effect of rendering it incompatible with other, unmodified endpoints. With the staggering growth of Internet in the recent decade the estimated amount of endpoints in the Internet today is vast; ISC Internet Domain Survey estimates the Internet host count to have been over 680 million in July 2009 compared with 43 million in January 1999 [7]. Large-scale efforts of introducing new endpoint functionality and support for it in global scope are thus usually considered quite infeasible.

In contrast to endpoint-oriented approaches, several suggestions use an overlay approach by introducing a small number of proxies with enhanced functionality at key points in the network – for instance, as gateways between the Internet and a mobile access network – to relay traffic between each other and endpoint hosts. Even a single proxy in the network can be considered to form an overlay. Usually endpoints have to be provided with extra functionality or information to communicate with proxies, but their capability to communicate with other hosts using standard applications and protocols is not hampered in any way.

Such overlay approaches include the Euonym architecture [8], which places intermediate hosts with shim layer software in isolated networks and operates on custom “name stacks” to achieve IP address independence and disconnection tolerance. The Distributed Home Agent for Robust Mobile Access, or DHARMA [9], which uses Dynamic DNS for IP independence and home and mobility agents for additional functionality and improved disconnection tolerance. It also provides the possibility to operate in end-to-end fashion by placing the agents at endpoint hosts - a technique applicable to overlay solutions in general.

Other notable suggestions include the Persistent Connection Management Protocol (PCMP) [10] and the Opportunistic Connection Management Protocol (OCMP) [11]. The PCMP is a session management protocol which uses custom peer names and can be deployed in proxies to provide persistent connections over disconnections and changing IP addresses. OCMP is a further development of the same idea and provides better support for applications and protocols besides the TCP.

Most of these solutions, be they end-to-end or overlay based, generally focus on the Internet, that is, their design focuses on the assumption that Internet is the principal carried network and endpoints operate using current Internet protocols – usually TCP. This choice carries with it certain implicit assumptions about the conditions the communication takes place in. They of course try to do away with the assumption that endpoints are always connected, but, disruptions notwithstanding, use of TCP usually assumes that connections tend to be fairly reliable and have a relatively small latency and round-trip time – usually of the order of under a second, probably much less, and in worst cases not more than several seconds. Likewise, use of TCP presumes a given degree of interaction between endpoints; three-way handshake is a requirement before data can be transmitted, and acknowledgements are continuously needed. In the largely favourable conditions of the terrestrial Internet these preconditions usually hold. But for more constrained communication environments different approaches are needed.

2.2. DTN

Delay-Tolerant Networking, or DTN, was developed partly as one possible answer to some of the more important shortcomings of the TCP/IP suite in communication over long-distance, high-delay, low-bandwidth, disruptive links. Interplanetary communication in our own solar system is often used as an example target application. DTN is not a single protocol or mechanism but a generalized architecture for communication over different networks – “regions” – with might use completely different addressing and routing mechanisms and protocols. The DTN architecture is described in RFC 4838 [12] and a good source of more information is the Delay-Tolerant Networking Research Group web site [13].

In deep-space communications, such as between Earth and space probes in Saturn orbit, distances become so great that message propagation at the speed of light will take well over an hour, given the speed of light of approximately 3 x 10^8 m/s and minimum distance between Saturn and Earth of about 1.2 x 10^9 km as stated in [14]. The propagation delay in this case would work out to about an hour and seven minutes (4000 seconds). For TCP three-way handshake, it would then take about 3h 20min before the transmission of the first actual data byte. To make matters worse, other celestial bodies of the ecliptic plane, including the sun, might interject between the probe and Earth listening stations or satellites, meaning that no direct communications could be established at all, for relatively long periods of time. Using relays positioned elsewhere in the solar system, such as in Mars orbit, could provide a communication path, but it would lead to even greater delays in communication. Clearly, TCP and Internet protocol suite are not appropriate for communication of this scale.

Besides the poor performance of TCP outside Internet conditions, another motivation for the DTN architecture is the existence of other communication networks besides the Internet and the desire to be able to relay messages through heterogeneous networks with incompatible addressing and protocols via a unified mechanism. For instance, it might be desirable to relay data from a remote underwater acoustic network otherwise disconnected from the Internet via a satellite link to a research station. In this case, data messages would have to traverse first through the acoustic network, then through the satellite link, and only at the final stages of communication through some part of the Internet before reaching the other endpoint.

Of course, networks outside the Internet could always be easily incorporated into the Internet by simply applying the IP protocol to all component networks. However, just as with the TCP, IP is not necessarily always a feasible solution for all conditions and networks. For instance, in some constrained environments where memory, processing and bandwidth are scarce, overhead incurred by having to transmit the 40-byte IP header plus higher-layer protocol headers and associated header processing might prove prohibitive. Or in very scarce networks with very few nodes and/or fixed links there might not be need for routing functionality provided by the IP. In such cases it could well be more sensible to apply other networking technologies in place of IP and then connect to the Internet using a higher-layer mechanism – such as the DTN.

To solve the challenge of delay-tolerance, DTN specifies a message storing-and-forwarding mechanism with persistent storage, and a higher-layer end-to-end message protocol, called Bundle Protocol and specified in RFC 5050 [15]. To make spanning of heterogeneous communication networks possible, DTN uses a URI-based (see RFC 3986 [16]) addressing, the aforementioned Bundle Protocol, and a mechanism called convergence layer. Application data is packaged into bundles, which are routed through the DTN using transport protocols applicable to component networks along the way. Bundle data is passed to the transport protocol through an appropriate convergence layer, which essentially provides a protocol to transmit bundle data to another DTN entity using transport-layer protocols pertaining to the network being traversed, and then relays it to the next bundle router. For instance, in the Internet, DTN nodes might have TCP and UDP convergence layers for relaying bundles.

Figure 1 below is a simplified example depiction of DTN communication between two different networks which use a satellite relay to opportunistically forward bundles when a communication satellite passes over.

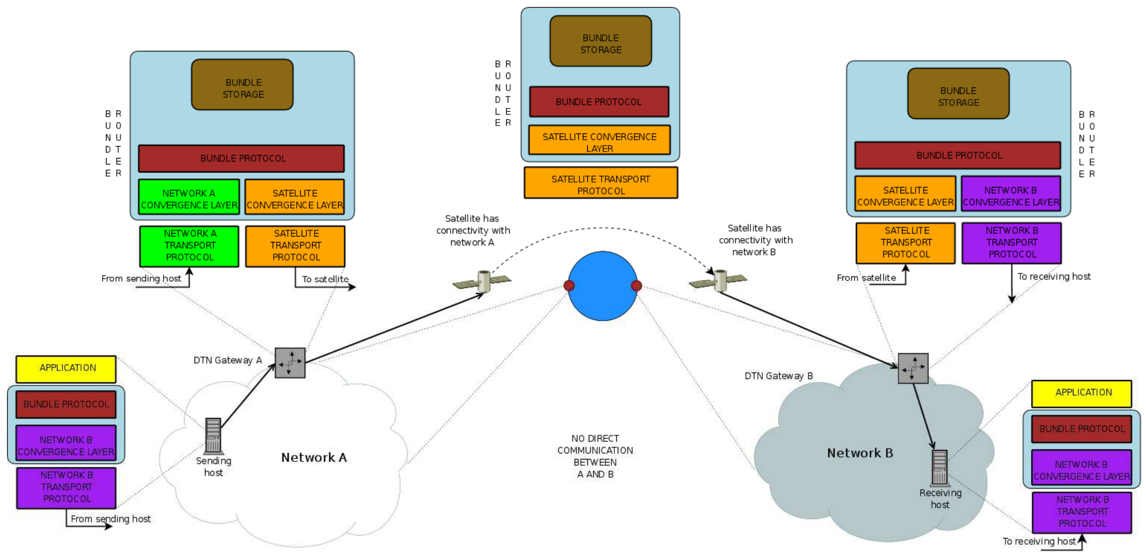
BUNDLE ROUTER
BUNDLE STORAGE
BUNDLE PROTOCOL
NETWORK A CONVERGENCE LAYER
SATELLITE CONVERGENCE LAYER
NETWORK A TRANSPORT PROTOCOL
SATELLITE TRANSPORT PROTOCOL
From sending host
To satellite
BUNDLE ROUTER
BUNDLE STORAGE
BUNDLE PROTOCOL
SATELLITE CONVERGENCE LAYER
SATELLITE TRANSPORT PROTOCOL
Satellite has connectivity with network A
Satellite has connectivity with network B
BUNDLE STORAGE
BUNDLE ROUTER
BUNDLE PROTOCOL
SATELLITE CONVERGENCE LAYER
NETWORK B CONVERGENCE LAYER
SATELLITE TRANSPORT PROTOCOL
NETWORK B TRANSPORT PROTOCOL
From satellite
To receiving host
APPLICATION
BUNDLE PROTOCOL
NETWORK B CONVERGENCE LAYER
NETWORK B TRANSPORT PROTOCOL
From sending host
DTN Gateway A
Sending host
Network A
NO DIRECT COMMUNICATION BETWEEN A AND B
DTN Gateway B
Network B
Receiving host
APPLICATION
BUNDLE PROTOCOL
NETWORK B CONVERGENCE LAYER
NETWORK B TRANSPORT PROTOCOL
To receiving host

Figure 1: DTN communication architecture – example scenario

Routing between different networks and DTN nodes is something of an open question, not having been explicitly defined in the specification. Routing mechanisms and protocols vary from network to network, and routing might be implemented using static routes or existing routing protocols adapted to DTN.

The Bundle Protocol is in a sense at the heart of the DTN and has many features and capabilities. To begin with, bundles are the basic unit of storage and transmission in the DTN architecture. The bundles are forwarded to the next hop towards the target endpoint, and if they cannot be immediately forwarded, e.g. if no connection is available for the next hop, they are stored until communication is possible. Bundle protocol also has mechanisms for authentication, bundle fragmentation and notification of successful delivery. It also defines and provides custody transfers, meaning essentially that nodes further along the communication path can accept the responsibility for storing and retransmitting bundles, providing more efficient retransmission behaviour in long and challenged paths.

The DTN architecture and related protocols are quite complex; bundling and bundle processing as well as convergence layer operations induce an extra overhead in transmissions and processing. DTN Research Group has developed and published a reference implementation of the DTN suite, labeled DTN2. The currently available version, 2.6.0, is hosted by SourceForge [17] and is a central component in this work. The reference implementation supports most DTN features and has TCP and UDP convergence layer functionality, but it requires extensive support libraries and has an overall memory footprint of around 40MB [18]. This is quite much considering mobile devices, wireless access points and similar hardware today. However, it has a well-defined API, reasonably useful documentation and provides an accessible starting point for concept testing and development and was thus chosen to serve as the foundation of this work.

2.3. CHIANTI

Challenged Internet Access Network Technology Infrastructure [19], or CHIANTI, is a two-year research project within the ICT initiative of the Seventh EU Framework Programme [20], scheduled to end in February 2010. It is a multilateral effort between participants from the public and private sectors, participants being European universities and enterprises. In a nutshell, CHIANTI aims to improve mobile user experience by providing existing user applications enhanced disconnection and disruption tolerance, making use of the existing Internet infrastructure and by adding deployable service-support infrastructure to key locations, as expressed in project deliverable D1.1 [21].

The CHIANTI project defines spheres of control, defined in D1.1, based on the main functioning entity within the sphere, and specifies CHIANTI components and most important use cases in the form of two main scenarios, the Nomadic User and Vehicle Support scenarios. Deliverable D1.2 [22] provides more information about project requirements and goals, while deliverable D3.1 [23] describes the architecture in more detail as well as elaborates on the scenarios. CHIANTI protocols, in turn, are better explained in D2.4 [24].

The CHIANTI project has the ultimate goal of creating an improved service with commercial potential. As a consequence, CHIANTI system architecture has several practical and technical requirements regarding its deployment and functionality. Key requirements are ability to work with existing Internet infrastructure and with existing user applications and devices, while providing enhanced service to users with CHIANTI-aware equipment. As to the definition of enhanced service, D3.1 lists among other things requirements such as 30% increase in throughput in intermittent conditions and tolerance of disconnections longer than five minutes. In preparation, extensive traffic analysis has also been done in authentic environment, findings include clear prevalence of TCP in client traffic and consequently CHIANTI stresses optimization of TCP for disruption tolerance and increased throughput.

The functional core of the project is formed by a CHIANTI client-proxy pair, located at the opposite sides of a disconnection point. Client functionality could reside in a mobile device or in a moving vehicle and in the latter case could also protect vehicle occupants from disconnections – the key idea behind the Vehicle Support scenario. The proxy, on the other hand, usually resides somewhere in the “fixed” Internet and can serve several clients. The architecture also takes into account the possibility of nested client-proxy configurations possible in a system with several service providers.

Figure 2 below presents the CHIANTI architecture as it appears in the CHIANTI Deliverable D3.1. Different spheres are highlighted, clarifying their role in the overall architecture. CHIANTI proxies can reside in different spheres, depending on the role of their service provider. CHIANTI clients are not shown, but they always reside within the Mobile sphere.

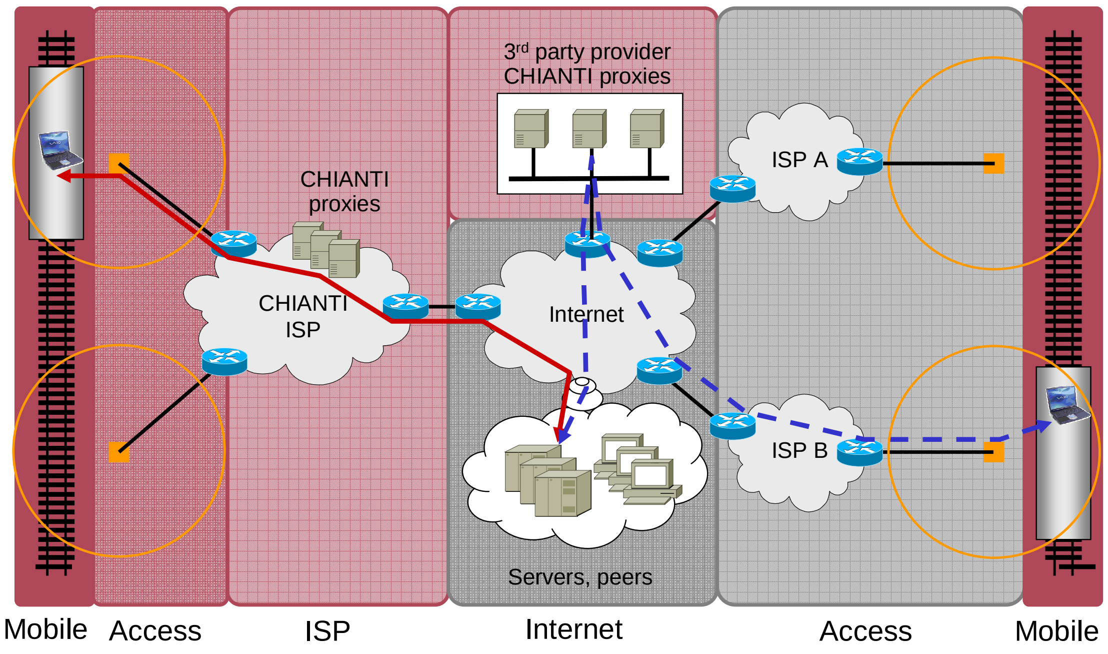

Figure 2: Overview of CHIANTI architecture, spheres of control; from [23]

Both client and proxy are equipped with the CHIANTI protocol stack which is essentially a core (called "Flex Proxy"), a chain of modules to provide application support for users, and a tunneling protocol for robust, disconnection-resistant communication between client and proxy. Interface for intermodular communication exists; CHIANTI modules communicate via the SOCKSv5 protocol, defined in RFC 1928 [25]. Figure 3 below depicts CHIANTI Flex Proxy topology with external chain modules as it appears in Deliverable D2.4. The DTN module would be one such external chain module.

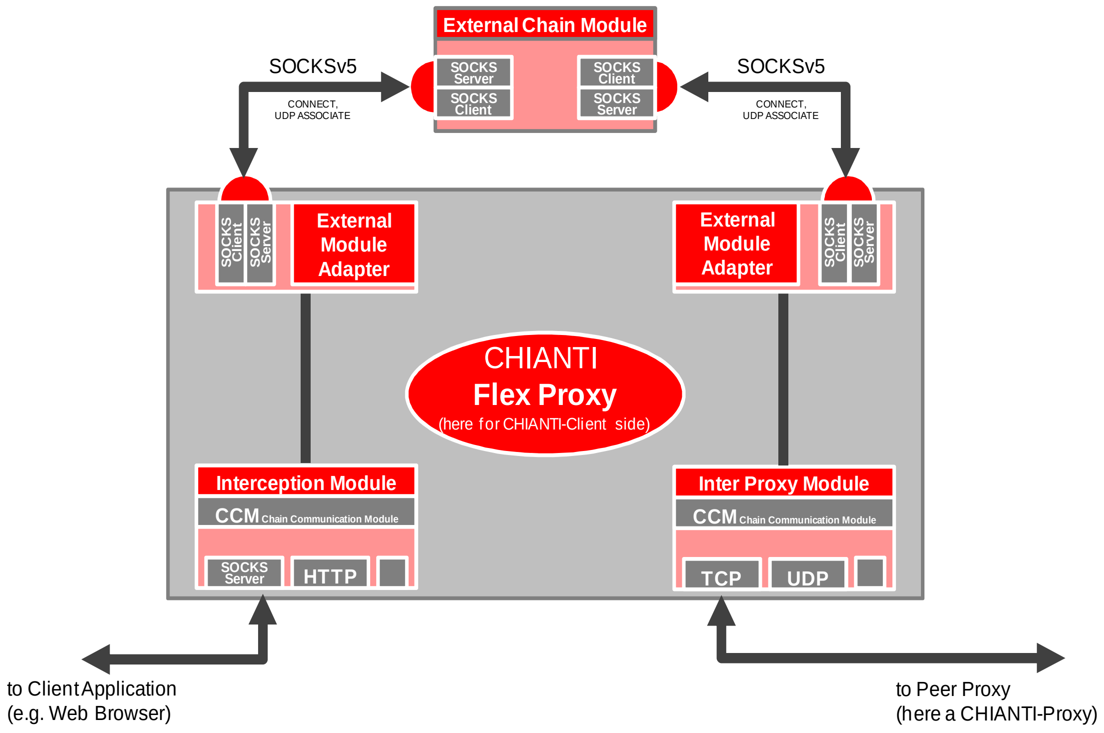

Figure 3: CHIANTI Flex Proxy topology, external chain modules; from [24]

As already mentioned earlier, this work arose from the sidelines of the CHIANTI project as a case study to explore applicability of DTN to provide users protection against disconnections. The modular structure of the CHIANTI protocol architecture should make it possible to implement a module which provides SOCKSv5 server functionality for incoming connections and directs incoming traffic to a DTN entity which is then used to relay data over an intermittently connected link to another DTN module, which would then feed it on, socksified, to awaiting module chain to pass on to the target endpoint.

The operational scenarios considered by CHIANTI are strictly limited to Internet environments, and the decision to use DTN, designed for much more challenging conditions, might seem almost inappropriate; especially as DTN is earlier criticized as an overly complex and resource-intensive for small-scale use in mobile environments. Even so, DTN should work fairly well in a benign environment. Testing the DTN provides an opportunity to investigate its impact and overhead on traffic, and, should the results seem promising, there is always the possibility of using a scaled down, more efficient implementation.

3. Design and Implementation

This section documents the design process and the reasoning behind the implementation. It also explores its details; the communication protocol, packet formats and such.

Subsection 3.1 details the design goals for the implementation.

Subsection 3.2 describes in more detail the problems and needs arising from the specified goals and the rationale behind the details of the protocol developed for the implementation; a review of design choices.

Subsection 3.3 describes the actual protocol in more detail. It contains description of protocol states and exchanges taking place between protocol entities and explains packet types and their header formats in detail.

Finally, subsection 3.4 describes the actual software implementation as well as the software environment of the implementation in more detail, also briefly commenting on some implementation-specific issues.

3.1. Goals of the design

Being from the outset affiliated with the CHIANTI project described in the previous section CHIANTI system architecture is implicitly reflected in the design of the software implementation, ultimately considering possible system integration within CHIANTI architecture. Implementation therefore has to keep in mind some of the key restrictions and considerations of the CHIANTI project itself, for instance the support of existing applications and protocols, and importance of TCP. The scope of this work, however, is much more modest than that of the CHIANTI, and concentrates on a particular TCP application, namely, the World Wide Web service using HTTP.

From a more practical perspective, purpose of the programming task here is to implement a computer program – from hereon referred to as daemon – which will accept user HTTP traffic and relay it through a DTN link to another similar daemon which in turn will relay incoming HTTP traffic onwards. For standalone-testing, the daemon should be able to relay HTTP traffic directly to the originally specified endpoint – i.e. the target web server – while, in consideration of the CHIANTI scenario and modular architecture, it should also provide interfaces to communicate with CHIANTI devices and modules. As CHIANTI defines SOCKSv5 protocol as its primary interface, implementation also has to provide at least a limited degree of SOCKSv5 functionality to be able to forward traffic to the next CHIANTI module in a possible module chain.

For accepting HTTP requests, and also in keeping in mind integration with CHIANTI architecture, the daemon should provide a (minimal) SOCKSv5 server. For the DTN link, DTNRG implementation will provide the API and functionality for bundle sending and reception; what remains to be done for the daemon is to multiplex several HTTP client connections into over a single DTN link, to convert incoming HTTP requests into bundles and relay necessary information pertaining to the HTTP connections – such as target address and identifiers – to the other communicating daemon, to make sure all bundles come across and that all data is relayed in correct order, and to keep track of endpoint connections at either end.

Initially, to demonstrate basic functionality, the daemon should listen for incoming SOCKSv5 connections and respond to “TCP Stream” requests in IPv4 protocol. Once these basic concepts have been implemented and tested, support for UDP and IPv6 along with other desired features can be added as deemed necessary.

The CHIANTI architecture describes a client agent residing in the mobile sphere, essentially at the mobile user’s side of the anticipated disconnection point, and a proxy agent residing at the other side of the disconnection point, with a fixed, reliable connection to the Internet. In a vehicle support scenario where one provider might e.g. have several CHIANTI clients in a single train and operate with several trains it is clearly impractical to have a separate proxy entity serving each client. This means that proxies must have the capability to distinguish between and communicate with several client entities.

Finally, in the interests of flexibility and simplicity – and of conformity with CHIANTI specifications – it was decided that the daemon should work in a symmetric fashion, i.e. each daemon instance should be able to provide both client and proxy functionality as needed.

3.2. Design

The main problems to be solved for the design of the daemon implementation can be summarized as follows:

- Opening a new connection context between daemons
- How to convert data from a TCP stream into bundles
- Multiplexing several client TCP connections over one DTN link
- Replicating TCP-style reliability between daemons
- Connection context termination

Each of these problems is expanded and discussed and corresponding design solutions presented in following subsections. Furthermore, there are some additional issues related to specification of the required implementation functionality which are less concerns of protocol design than they are details of the implementation itself, mainly:

- Symmetric operation of the daemon
- Connectivity detection

These issues are commented more later on in the subsection concerning the implementation itself

3.2.1. Opening a New Connection Context between Daemons

When a mobile client opens up a new HTTP connection, it will first have to perform a brief SOCKSv5 negotiation with the daemon. This initial exchange provides the daemon with knowledge of the IP address and port of the target host the mobile client wants to communicate with. Daemon at the client end has to explicitly relay this information to the other daemon so it will be able to establish connection to the target host; after the initial SOCKSv5 negotiation, client endpoint will start receiving TCP stream data from the mobile client which is then packaged into a bundle and sent to the other daemon.

At the very least, the first bundle to be transmitted needs to have target IP address and port information included in a bundle header. Also needed is an identifier which creates the context between the two daemons of a unique HTTP exchange – there might well be several requests directed at the same web server, so an address/port pair is insufficient for context identification.

After the initial bundle is received by the other daemon, it can immediately establish a TCP connection to the target host and deliver received data; as it receives reply data from the end host it includes the context identifier in the bundle header so the client end daemon can direct received data back to the appropriate mobile client. There is no need to include address/port information bundle headers after the first bundle, but there could be slight additional benefits for supplying the address information with each bundle header. Mainly, implementation will be slightly easier with identical bundle header structure for all data bundles. Also, if the first bundle is delayed or gets lost, connection to the end host can be opened upon the reception of the second bundle, although this is hardly an advantage as bundle data cannot be transmitted out of order anyway.

3.2.2. TCP Stream Conversion into Bundles

Conversion of continuous TCP data stream into bundles presents some interesting problems. First, there is no predefined size limit for a bundle; they can be almost arbitrarily small or large. Different bundling sizes pose different benefits and drawbacks: smaller bundle sizes means smaller delays in bundle transmission but on the other hand induce a greater overhead from bundle headers and thus decrease overall effectiveness of transmission; larger bundles mean extra delay and increased likelihood of a bundle being discarded in case of packet loss but increased overall efficiency.

Second, the issue of multiplexing described in the next subsection also has an effect on bundling; data from several different connections could be stored in a single bundle. This issue is more closely examined in the next subsection; the design choice made here is to keep each bundle associated with only one client connection.

Third, HTTP protocol session usually consists of an exchange of relatively small messages, client first sending a request for a resource and server replying with a status message or with the desired resource. In these cases, waiting for a bundle to be filled with more data is impossible and a timer mechanism is needed to trigger bundle transmission in case of inactivity so that messages will be delivered in a timely fashion and the user will not have to experience excessive extra delay. This, in turn, leads to the problem of choosing an appropriate bundling timeout, especially at the server side which is connected to the Internet and will probably behave in a much less predictable way than the client side due to larger variations in available bandwidth, latency and server load.

Too long or too short timeout will nullify any benefits gained from optimal bundle size; too short timeout will make connection more responsive but will also incur larger overheads, and too large timeouts will increase unresponsiveness and negate benefits from smaller bundle sizes. Having said all this, the subject of choosing optimal combination of timeouts and bindle sizes is a complex mathematical exercise and ultimately beyond the scope of this work, where quick tentative qualitative concept testing takes precedence to excess quantitative optimization. Once initial testing is complete, further refinements may be implemented in the form of e.g. user-selectable parameters and/or adaptive bundling algorithms. For the initial version, maximum bundle size limit is therefore set at 48 KiB, and value of bundling timeout is left to user's discretion with the possible range of 1 to 1000 ms. This will also enable tentative tests for finding out good approximations for practical use.

3.2.3. Multiplexing Client TCP Streams into the DTN Link

TCP connection multiplexing over DTN link is in itself fairly straightforward. The basic solution for multiplexing several connections into one link is to supply each different connection with an identifier and to label each bundle with this identifier. Inherent here is the assumption that a single bundle will only contain data from a single connection.

While it is perfectly possible to define a more elaborate and flexible framing format for packing a bundle with data from several TCP connections, as mentioned in the previous subsection, it would add considerable complexity to the implementation and have an unpredictable and probably detrimental impact on perceived client connection quality. For instance, if a new connection is established and a short "HTTP GET"-message is sent through it, bundling timeout must be applied to trigger sending a bundle containing the request. Just waiting for more data to arrive is obviously not a solution as the connection might be the only client connection present in the daemon and no further data will arrive before the requesting client receives a reply from the server. Now, then, if there is more traffic present, it is possible that more data will arrive at the daemon from some other client before the timeout is triggered. In this case, the bundle will be filled with arriving data and a new timeout set. This cycle is repeated until timeout triggers or the bundle is full. In the worst case scenario, new connections with short initial packets could arrive at the daemon just before the timeout, in which case the original first connection would experience considerable extra latency, possibly several times larger than the value of bundling timeout itself.

Another drawback in multiplexing TCP connections within a bundle is the fact that an out-of-order arrival or actual loss of a bundle will then hamper all the client connections having had data in the disrupted bundle instead of just one connection. Particularly loss events are troublesome as the lost bundle will have to be first detected as being lost and then retransmitted, possibly leading to much greater delay suffered by client connections than if the bundle had been simply delayed just enough to arrive out of order.

After deciding on labeling, the size and format of the connection identifier label has to be decided on. Given the TCP/IP architecture with 16-bit port number identification, the maximum amount of TCP connections per one IP address remains at 2^{16} and is typically much less. However, traditional TCP/IP implementations in different operating systems use sockets for binding into TCP (and UDP) communication endpoints, which are typically identified with a 4-byte integer value with a range greatly exceeding 2^{16}. Therefore the most straightforward solution here is to use the 4-byte socket identifier as provided by the target platform as such.

The idea of using identifiers is to provide uniqueness to each connection. Using socket identifiers provides a degree of uniqueness in the sense that no two sockets may coexist with the same identifier at a single host. However, it is entirely possible if not indeed probable that two successive client connections be assigned the same socket identifier. A degree of protection against such temporal collisions should also be included in the protocol to prevent daemons from getting confused by late-arriving bundles having belonged to a previous client connection. This is to some degree an implementation issue, as the identifiers are already bound to sockets and as such are affected by the rules of the socket API. By way of an example, this provision of uniqueness in time as well as in numerical space can be done with a bit of extra accounting of recently used connection/socket identifiers or by attaching extra delay to calls for closing sockets after client connection teardown.

3.2.4. Providing Sufficient TCP-style Reliability between Daemons

Traditionally in the Internet TCP has provided end-to-end reliability to higher-level protocols. In this case, TCP will not be able to operate end-to-end because of the interjecting daemons. Instead, endpoints are in TCP communication with a daemon entity, and daemons communicate via a custom protocol which uses Bundle protocol as a transport, which in turns uses convergence layer agents for point-to-point transmission.

Bundles are generally subject to similar problems in transmission as IP datagrams. Given that the DTN link between daemons is unreliable and prone to interruptions, bundles may get lost in transit. Even if they do not, it is entirely possible for them to arrive at their destination out of order. For all these reasons, daemon entities must have mechanisms for keeping track of sent and received bundles and their correct sequence, for acknowledging or requesting retransmitting bundles and storing bundles in buffer for possible retransmission until they are acknowledged.

The traditional method to counter out-of-order arrival of and to facilitate keeping track of bundles is to apply sequence numbers to them. Both daemons must keep separate sequence numbering; client connections have their own flow, and responses received from the remote endpoint have their own, applying a common sequence numbering to apply to both directions is difficult, especially considering the assumed intermittent nature of the DTN link and subsequent possible delays in packet arrival.

For acknowledgements and retransmissions, information about requested and acknowledged bundles has to be included in protocol messages. And, although DTN and Bundle Protocol do provide persistent storage for bundles, their retransmission mechanisms and reliability are unclear and it is probably safer to keep a buffer of sent bundle payloads in memory for more flexible and effective bundle retransmission.

HTTP is a request/response protocol. Typically, an HTTP session consists of a number of exchanges between client and server; client sending a request for a resource and server responding to the request. In other words, there are going to be alternating data streams to both directions between the two. Usually, one data stream (i.e. a request) has to be received in its entirety before another data stream (i.e. response) can be formulated and sent. This assumption may not hold generally, but for the purposes of designing and implementing the necessary reliability features for relaying HTTP traffic it is assumed to be valid.

In this sense, a daemon receiving a reply from the other daemon usually means that the data stream and all bundles belonging to it were successfully delivered and serves as a sort of acknowledgement. Including a bundle sequence number corresponding to the latest received bundle in the reply helps daemons with buffer management, as they can instantly discard all bundles from the buffer which have a sequence number equal to or smaller than the number reported in the received bundle.

Large, unidirectional file transfers mean there might be plenty of bundles flowing in one direction but none in the other, so explicit acknowledgements (ACKs) are needed as the transmitting daemon will not receive enough feedback from the other daemon for effective buffer management otherwise. The details for this acknowledgement mechanism need to be defined.

Sending ACKs for each bundle seems excessive and creates lots of traffic with large overhead – bundles with no other payload than identifiers and a 4-byte sequence number. ACKing several bundles at once on the other hand means that a single ACK getting lost can have a larger impact on communication. A threshold value n could be defined, as a function of the bundle buffer size, e.g. one-quarter of the buffer size, and ACKs then sent for every n bundles, always reporting the latest consecutive bundle sequence number received. If data stream ends, a reply is most likely to follow and will again be supplied with the sequence number of the last received bundle.

Also, retransmission mechanism needs to be given due consideration to gain sufficient reliability without sacrificing too much performance – the purpose, after all, is to provide a service enhancement for mobile Internet users.

One question is which one of the daemons is responsible for retransmissions, the sending or the receiving daemon. Of course, the receiving daemon cannot know if there is data coming in, especially before establishing a new connection context, so the sending daemon must assume responsibility for retransmitting bundles if it receives no reply to the bundle it has sent. This can be done with a retransmission timeout. On the other hand, the receiving daemon has to keep track of arriving bundles, and in case of bundles arriving out of order, possibly due to a lost bundle, it will have to make a decision on when to request a retransmission of a bundle. For instance, if bundle 5 is received after bundle 3, immediate request for bundle no 4 could be premature. On the other hand, bundles are relatively large compared to e.g. Ethernet frames, and after receiving bundle 6 it is getting more and more unlikely that bundle 4 will arrive without it being retransmitted. A threshold value will have to be defined.

Another question is whether the sending daemon should proactively trigger retransmissions whenever it is sending a bundle with sequence number greater than the last received ACK number plus the aforementioned threshold value. This would reduce the impact of ACK bundles getting lost but in a case of connection disruption would lead to fruitless retransmissions. As connectivity is assumed unreliable, it is probably better for the sending daemon only to retransmit on request. Better yet, using selective acknowledgement mechanism similar to that of TCP will also allow for more efficient retransmission behaviour. For instance, in the previous example, bundles up to 3 could be acknowledged cumulatively, but on top of that explicit acknowledgements for subsequently received bundle 5 and later could be included. This would allow the sender to react faster and retransmit missing bundles.

Using timeouts always brings forth the question of finding a suitable value for them. Short timeout values generally improve responsiveness, but might cause needless extra traffic, while long timeouts make for more efficient bandwidth usage but increase the impact of bundle loss. Experimenting with different timeout values is once again needed for finding suitable values, but for initial implementation and concept testing reasonable default values will have to be defined.

Further additional problems with the Bundle Protocol specification are identified in [26]. The most notable problem mentioned is the lack of reliability due to there not being a usable checksum mechanism in place. In light of this it would doubtless be a good idea to implement a simple checksum mechanism by implementing a simple hashing algorithm to be applied to TCP data and reserving some bytes for storing the resultant hash value within the bundle. However, in can also be noted that within the Internet environment, by using the TCP convergence layer supplied by the DTN software, it is possible to gain some benefits from TCP's own reliability mechanisms: a bundle transmitted through TCP can be trusted to be uncorrupted if it arrives at its destination. So, for the purposes of this project, adding checksum mechanisms is probably not a priority issue; if desired or if deemed necessary by test results, it can be implemented later.

In summary, bundle headers will have fields for sequence numbers and for latest sequentially received sequence number – the cumulative ACK – plus for sequence numbers received after that – the selective ACKs. For simplicity, sequence number will be an unsigned 4-byte integer. This provides 2^{32} unique sequence numbers per connection and even without wrap-around mechanisms will be quite sufficient for the initial testing.

Daemon schedules two timeout events as it sends the first bundle of a data stream to another daemon. The value of the first timeout, or retransmission timeout, is initially 5 seconds and probably subject to change during testing, the second timeout is a connection timeout of 5 minutes. If the timeout triggers without daemon having received a reply or an acknowledgement it will retransmit all unacknowledged bundles in its retransmission buffer, setting another timeout with equal value. The daemon will repeat this behaviour until it receives a reply or until connection is timed out.

Daemons will keep a retransmission buffer of several bundles. Initial value is more or less arbitrarily selected as 16 bundles, different values can be used and tested during testing phase. On reception of an acknowledgement sequence number in a bundle daemon will discard from its buffer bundles with sequence number equal to or smaller than in the acknowledgement. Receiving daemon will send out acknowledgements for every retransmission buffer / 4 bundles received successfully. If retransmission buffer fills up, daemon must not receive any more data from the endpoint it is communicating with before the other daemon has acknowledged some earlier bundles and buffer space may be freed.

3.2.5. Connection Context Termination

A connection between a HTTP server and client can be closed by either end. Endpoint hosts are not in direct connection but instead converse with daemons. After either endpoint closes the connection, there is probably data left in the pipeline waiting to be relayed to the other endpoint. Daemon at the closing side in these cases must take care that all data is delivered and then notify the other daemon about the connection having been closed. Both endpoints can then, after data has been delivered, release all resources related to the connection context.

A protocol flag is reserved for connection teardown notification. The last bundle in the data stream coming from the closing client will be marked with the teardown flag by the daemon sending it. The other daemon, upon receiving a bundle with a teardown flag must then acknowledge the final bundle.

By now the most critical issues regarding the requirements of the working protocol have been elaborated on, briefly but adequately. The next subsection will focus more on details of the protocol specifics themselves and serve as the protocol specification.

3.3. Protocol

The previous subsection has described the design problems and choices; this subsection concentrates on the detailed description of the protocol arising from those choices and also defines its details: protocol states, message types and exchanges, and bundle header formats. The protocol is fairly simple and designed for quick concept testing and implementation and is no doubt suboptimal; the precedence in design has been in putting together a workable first approximation for a future protocol basis.

3.3.1. States and Exchanges

In its default state, the implementation runs a SOCKSv5 server and waits for incoming SOCKSv5 client connections. The first major state transition occurs when a new client connection is established, and daemon creates a new connection context, which then moves into the SOCKS negotiation phase, which actually has several sub-states according to the proceeding of the SOCKS negotiation. A failed SOCKS negotiation results in termination of the connection.

After the SOCKS negotiation is complete, the HTTP dialogue between endpoints begins. HTTP itself is a stateless protocol, and the daemon only relays HTTP data between itself and another daemon, so it has no need to keep track of any specific states while the connection context is established. Client daemon enters the established state when it transmits its first bundle. Upon receiving the first bundle for a new connection context, the proxy daemon, depending on whether relays traffic to a chain module or to endpoint, will either enter SOCKSv5 client negotiation phase or connection context establishment state. A failed SOCKS negotiation or endpoint connection establishment will result in teardown of the connection context.

Connection context enters the teardown phase as either daemon detects that its served endpoint has disconnected. At that point, last bundles are sent if data is in the buffer and the last bundle is flagged as disconnected. Upon receiving a disconnection-flagged bundle, other daemon sends a final acknowledgement and is then free to tear down the connection context. The other daemon will do so upon receiving the final ACK.

Besides these message exchanges, connection contexts may be torn down if the connection timeout triggers at any point during the exchange, in practice after several minutes of DTN link disconnectivity. A state diagram is presented in figure 4 below.

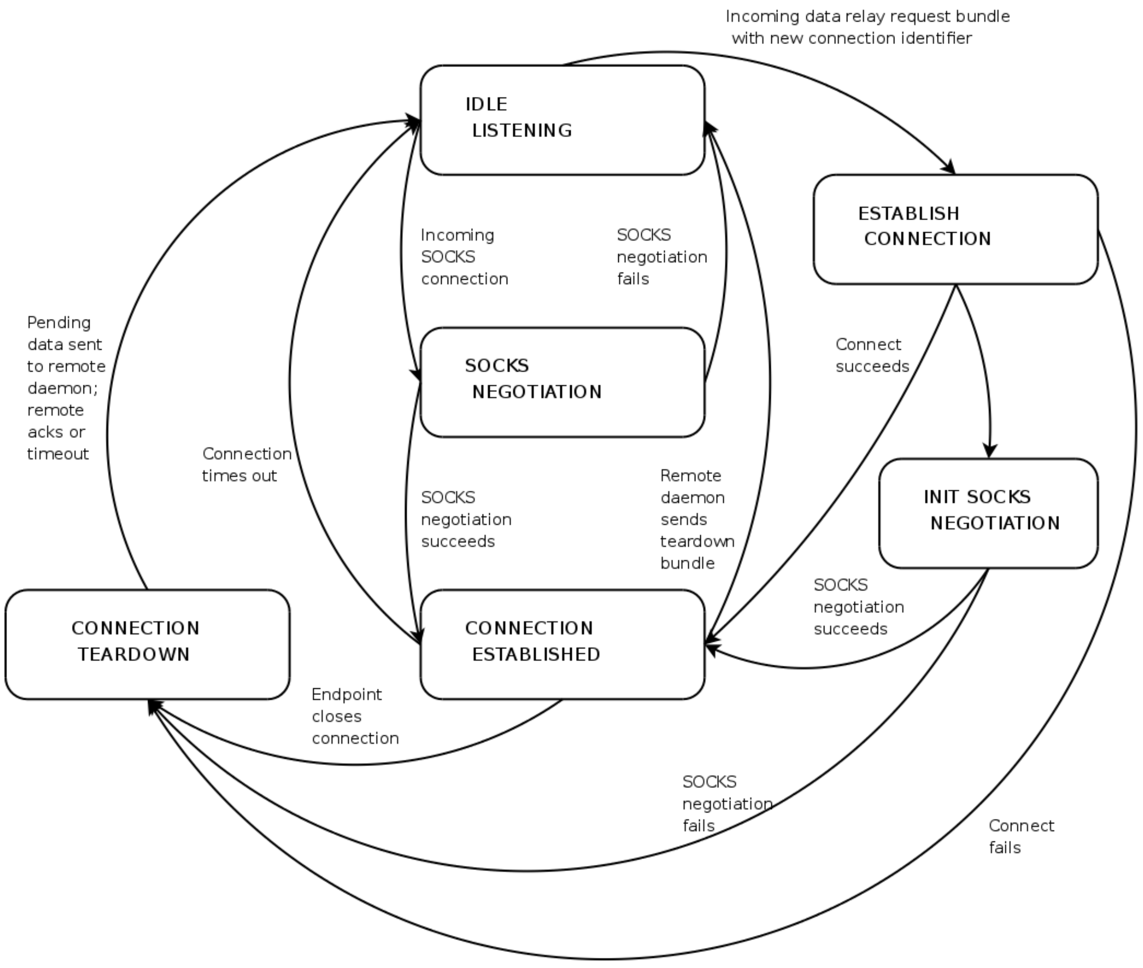
Incoming data relay request bundle with new connection identifier
IDLE LISTENING
ESTABLISH CONNECTION
Incoming SOCKS connection
SOCKS negotiation fails
Pending data sent to remote daemon; remote acks or timeout
SOCKS NEGOTIATION
Connect succeeds
Connection times out
SOCKS negotiation succeeds
Remote daemon sends teardown bundle
INIT SOCKS NEGOTIATION
SOCKS negotiation succeeds
CONNECTION TEARDOWN
CONNECTION ESTABLISHED
Endpoint closes connection
SOCKS negotiation fails
Connect fails

Figure 4: HTTP Relay protocol states

Protocol exchanges occur only in three distinct types: data bundles, acknowledgement bundles and retransmission request bundles. Data bundles come in two flavours: client-to-proxy bundles and proxy-to-client bundles, they are used for data transmission whenever other daemon has data from an endpoint to be relayed. In short exchanges, these data bundles have a double role as acknowledgements, but when larger one-way streams occur, separate bundles are used explicitly for acknowledgements. A retransmission request is sent when receiving daemon notices one or more missing bundles and has to explicitly request them from the other daemon.

All other messaging, such as relaying endpoint addresses and connection termination, are carried by these three bundle types. Exact bundle formats are the subject of the next subsection.

3.3.2. Packets and Formats

All bundles relayed by the daemon have a header, used for protocol signaling and relaying necessary connection related information between daemons. All headers have a protocol flag field, which is used to mark address and protocol types as well as protocol message types. In keeping with 32-bit field alignment, the size of the flag field is 4 bytes. First byte is used for various relay protocol flags, as follows:

Bit 1 signifies a client request - i.e. value 1 indicates that the bundle is coming from a client daemon agent as opposed to the proxy agent. This makes implementing symmetric operation easier.

Bit 2 flags bundle acknowledgement and is on if request has no data payload but is exclusively used for acknowledgement purposes.

Bit 3 flags a retransmission request, and if it is flagged, indicates that the bundle has info on what bundles are requested for retransmission, but no other data.

Bit 4 flags the use of IPv4 or IPv6 addressing, and like bit 4, is provided as a support for possible future implementation of IPv6 addressing. Value 1 means use of IPv4.

Bit 5 indicates that target address field is a DNS name instead of an IP address; SOCKS protocol allows for using DNS names instead of IP addresses.

Bits 6 and 7 have no immediate use for now but could be useful in the future if implementation is to be refined.

Bit 8 signifies endpoint shutdown and closing the connection. A daemon receiving a bundle with this bit flagged may, after acknowledging, tear down the communication context related to this bundle.

The second byte of the flag field indicates the protocol used on top of IP, e.g. UDP or TCP, and is copied directly from the Protocol/Next Header field of the IP datagram received from the client. This makes it easier to support additional protocols in future implementations.

The third byte signifies target address length in bytes. This information is needed when a DNS address is transmitted instead of an IP address with fixed length. One byte will be enough as DNS address length is limited to 255 bytes.

Fourth byte denotes header length and is counted in 4-byte words. This provides information for calculating the number of acknowledged bundles at the end of the header. Figure 5 below represents the flag field graphically.

Bit

1	2	3	4	5	6	7	8	16	24	32
REQ	**ACK**	**RTX**	**IPV**	**DNS**	**-**	**-**	**TDN**	**Protocol**	**Address length** (in bytes)	**Header length** (in 4-byte words)

Figure 5: Protocol header flag field

The Data Relay bundle has the 4-byte protocol flag field first. If the bundle originates in a client daemon, bit 1 has value 1 and bits 2 and 3 have value 0. If the bundle originates in a proxy daemon, all first three bits are 0.

For both types of Data Relay bundles, next field is the 4-byte connection identifier field, in practice containing the 4-byte socket identifier reserved for the client endpoint connection at the client daemon. Next 4-byte field contains the bundle sequence number, an unsigned integer value, 1 being the sequence number of the first bundle.

For client-originating Data Relay bundle, the next field is reserved for the target endpoint IP address, being four bytes for IPv4 addresses, 16 bytes for IPv6 addresses, and variable size for DNS addresses. In keeping with 32-bit header field alignment, DNS address field is padded with zeroes to an even multiple of

4 bytes. After the address field, next two bytes are reserved for the target port number, with the next two bytes padded with zeroes for 32-bit field alignment.

For all Data Relay bundle headers, next 4-byte fields are acknowledgement sequence number fields. The fourth byte of the flag field indicating header length provides the information needed for calculating the amount of acknowledgement numbers included. The first reported sequence number is always the cumulative acknowledgement; sending daemon reports here the sequence number of the last bundle it has received from the other daemon in a consecutive manner. If no bundles have yet been received from the other daemon, value here is 0; otherwise, daemon receiving a cumulative ACK may discard bundles from its send buffer with sequence number smaller than or equal to the number in this field. Subsequent fields, if present, acknowledge bundles received out of order in an ascending order. Figure 6 below illustrates both data relay headers.

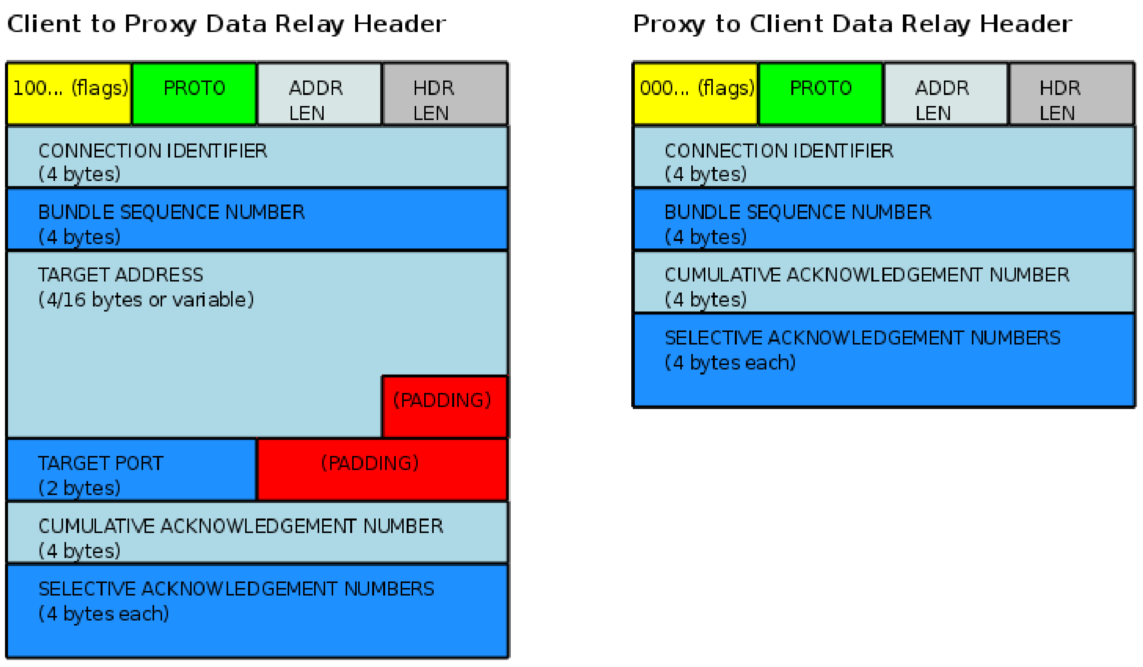
Client to Proxy Data Relay Header
100... (flags)
PROTO
ADDR LEN
HDR LEN
CONNECTION IDENTIFIER (4 bytes)
BUNDLE SEQUENCE NUMBER (4 bytes)
TARGET ADDRESS (4/16 bytes or variable)
(PADDING)
TARGET PORT (2 bytes)
(PADDING)
CUMULATIVE ACKNOWLEDGEMENT NUMBER (4 bytes)
SELECTIVE ACKNOWLEDGEMENT NUMBERS (4 bytes each)
Proxy to Client Data Relay Header
000... (flags)
PROTO
ADDR LEN
HDR LEN
CONNECTION IDENTIFIER (4 bytes)
BUNDLE SEQUENCE NUMBER (4 bytes)
CUMULATIVE ACKNOWLEDGEMENT NUMBER (4 bytes)
SELECTIVE ACKNOWLEDGEMENT NUMBERS (4 bytes each)

Figure 6: Data Relay bundle header formats illustrated

The Acknowledgement bundle has the 4-byte protocol flag field first. Bit 2 of the protocol field must be flagged, value of bit 1 is not significant, but value for bit 3 must be 0. Address length field has the value 0 as no address information is relayed. Header length field denotes normally the size of the header in 4-byte words.

Next 4-byte field is the connection identifier field, exactly as in the Data Relay bundle. The Acknowledgement bundle has no sequence number of its own, as it contains no data that has to be buffered and does not need to be kept track of. The last 4-byte fields in the ACK bundle, then, contain the cumulatively and selectively acknowledged bundle numbers, exactly as with the Data Relay bundle header. See figure 7 below for illustration.

The Retransmission bundle also has the 1-byte protocol flag field first, with bit 3 flagged. Value of the first bit is not significant but bit 2 must be 0. Address size field is 0 and header length field is as with other bundle types. The following 4-byte field contains the connection identifier number, and as with the ACK bundle, the Retransmission bundle has no sequence number of its own. Instead, next 4-byte fields contain sequence numbers of those bundles the daemon is missing – there might be several. The final 4-byte field is the cumulative acknowledgement of the sender. Providing sequence numbers for selective acknowledgements is redundant, as explicit retransmission of missing packets is already requested anyway. See figure 7 below for illustration.

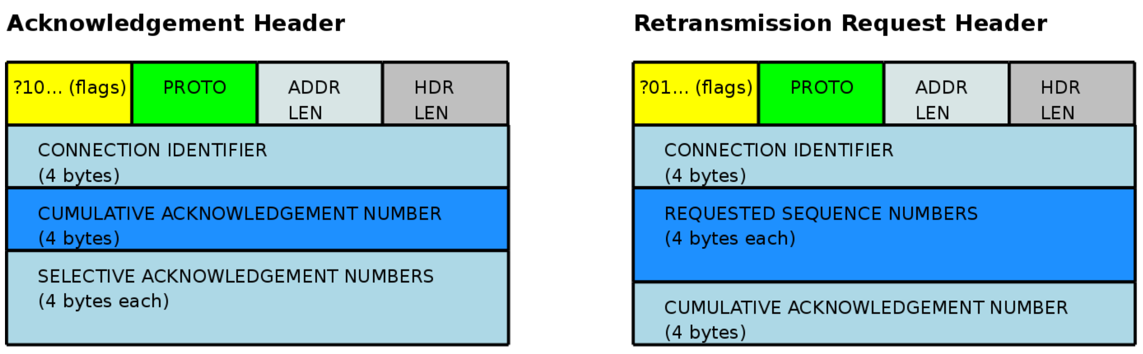
Acknowledgement Header
?10... (flags)
PROTO
ADDR LEN
HDR LEN
CONNECTION IDENTIFIER (4 bytes)
CUMULATIVE ACKNOWLEDGEMENT NUMBER (4 bytes)
SELECTIVE ACKNOWLEDGEMENT NUMBERS (4 bytes each)
Retransmission Request Header
?01... (flags)
PROTO
ADDR LEN
HDR LEN
CONNECTION IDENTIFIER (4 bytes)
REQUESTED SEQUENCE NUMBERS (4 bytes each)
CUMULATIVE ACKNOWLEDGEMENT NUMBER (4 bytes)

Figure 7: Acknowledgement and Retransmission Request bundle headers

3.4. Implementation

This subsection describes the composition and establishment of the software framework and environment for the prototype implementation. It also describes the more immediate practical design of the implementation code and what information is stored and how it is organized as well as provides commentary on some choices that have been made during implementation process.

3.4.1. Software environment

The central software component of this project, the DTN2 reference implementation, has been developed for and tested in a Linux environment. For this reason alone, but also for reasons of familiarity and relative ease of programming, Linux was chosen as the development platform for this project as well.

Linux comes in many flavors, difficult to compare in appropriateness to the task at hand without considerable experience and deeper knowledge of the properties of different distributions. Thus, mainly for reasons of familiarity and ease of maintenance a 64-bit x86 version of a Debian [27] release dubbed as Lenny [28] was chosen. The development platform uses 2.6 series kernel.

The DTN2 distribution is hosted by the Sourceforge web site. However, the hosted version is rather old, dated July, 2008 and did not compile and run on the more modern operation system used here. Sourceforge also has latest developing versions of the code hosted in a Mercurial [29] repository; this implementation uses the Mercurial versions of the DTN2 from summer 2009.

DTN2 also requires a set of support libraries called Oasys, also available on Sourceforge with and without Mercurial [30]. Configuring and compiling the DTN2 implementation also requires a selection of other dependent sotftware packages: GNU C and C++ compilers version 3.3 or newer, 3.4 was used in this project. It also requires development packages of TCL, any version between and including 8.3 and 8.5 series. Bundle storage requires a database backend. Used here was the BerkeleyDB version 4.6 development version; versions from

4.2 to 4.7 inclusive can be used. For XML processing the xerces library version 2.6 or newer is a requirement as well.

The design of the prototype implementation requires an effective handling of several incoming and outgoing network connections in an asynchronous manner. Also, protocol specification calls for timeout mechanisms for which an event scheduler has to be implemented.

For asynchronous network I/O the first solution to come to mind is the socket handling interface provided by select(). From earlier experience, however, this is considered to be a cumbersome and limited interface. Also, implementing an event handler for timers and events other than socket activity is not a trivial task, once again deriving from earlier experience with similar software programming activities. An event notification library known as libevent [31] provides a convenient, ready implementation to solve these both problems, and thus libevent library version 2.0.2-alpha was chosen to be used here. The alpha version was preferred, as the Debian package management system coupled with the chosen distribution provides only version 1.3 of libevent, which lacks several features supported by later versions.

Finally, to assist in development and testing, a virtualization software called VMWare Workstation 6.5 [32] was used. VMWare Workstation provides a convenient environment for quickly deploying a number of virtualized testbed computers and allows for saving and resuming virtual machine states for extremely convenient testing, although it is by no means necessary for the development process.

3.4.2. DTN Reference Implementation

To be able to run the daemon on a host, an instance of the DTN reference implementation needs to be running on the same host, as well as a set of libraries required by the daemon and the DTN. The compilation, installation and configuration procedures for the DTN reference implementation are sufficiently well documented elsewhere [33] and repeating such instructions here makes little sense. Likewise, installation of the libevent library is a fairly straightforward procedure and sufficient guidance is provided with the release.

DTN reference implementation needed to be configured for various options before it could be used. The implementation uses databases as a backend for bundle storage; Berkeley DB was chosen as the easy and lightweight option. Nodes were given simple addresses using the format dtn://[hostname].dtn. Routing between DTN daemons using such unusual addressing was done using static one-way routes, configured into the DTN configuration file, with IP address information coupled with the dtn address entry and TCP convergence layer links defined between DTN nodes. As for bundle transmission, the reference implementation specifies payload types of either “memory” or “file” when using the DTN API. For this work, memory-type payloads were used to avoid unnecessary file operation overheads. Maximum bundle size for memory-type payload was defined as 50000 bytes in the implementation code, so bundle size used in the relay protocol implementation was chosen to be 48 KiB.

3.4.3. Implementation Architecture

The prototype implementation performs several functions and provides data types and structures for handling all associated protocol data. A rough overview of the organization of these structures follows.

- Endpoint connections are stored in connection tables. There are two connection tables: one for storing client-to-proxy connections and another for storing proxy-to-client connections. These are fixed-size pointer tables with necessary management functions for keeping track of active endpoint connections.
- A data structure is defined for containing all relevant data pertaining to a single endpoint connection. This endpoint data structure gathers all the individual variables and other related items required for managing bundle transfer between the client-proxy pair. These items include endpoint IP addresses and port numbers, sequence numbers, socket identifiers and

buffers for storing unacknowledged or out-of-order bundle data, and event instances for event management.

- Data for individual bundles is stored in a buffer structure which contains a memory block for data and necessary variables for keeping track of data size, sequence numbers and such necessary protocol information.
- Buffers are implemented as double-linked lists of bundle buffer structs. One is needed for bundles sent over the DTN link and another for received bundles.
- Event management is handled by an event base (implemented in the external libevent library) into which events are registered.
- Events are information structures which define an event type (read, write, timeout), associated socket and callback function used for handling the events.
- Finally, the DTN-API provides a socket-like descriptor for the DTN connection and necessary functions for bundle management, reception and transmission.

Following is a rough description of the most important functional blocks of the implementation as well as their responsibilities. It also provides a general understanding of the program flow in normal operation.

- The main loop initializes the DTN-API interface, sets up the event handler base, establishes a server port for listening for incoming connectionsfrom endpoints, sets up all the necessary event structures, connection tables and such and starts the event loop.
- A callback function for events associated with the server port is run when endpoints connect to the server port. This callback accepts the incoming connection and initializes all relevant endpoint data structures described earlier.
- A callback function for reading data from a socket associated with an endpoint connection is executed when data arrives at such a socket. This callback performs SOCKSv5 negotiation at the client entity, and at both

daemon entities, accepts and stores incoming data into a bundle buffer. If the buffer fills, it dispatches the bundle for delivery by setting up a dtn write event, and if not, sets up a bundling timeout event for triggering bundle transmission.

- A callback function for writing data to the DTN handle is executed when there is a data buffer pending packaging into a bundle and transmission over the DTN link. It also manages sending of acknowledgements and retransmission requests to the other daemon when such transmissions are triggered.
- A callback function for reading data from the DTN handle is executed when a bundle arrives at a daemon. This callback function parses the information in bundle headers and initializes and establishes endpoint connections at the proxy daemon. It manages incoming bundle buffer space, monitoring for lost or out-of-order arrivals of bundles, setting up retransmissions requests when necessary, and triggers write events to endpoint sockets when it has more data to be relayed to an endpoint. It also reacts to retransmission requests as well as to acknowledgements, freeing obsolete buffers as they get acknowledged by the other daemon.
- A callback function for writing data to endpoint is executed when daemon decides there is data to be relayed to the endpoint. The callback sends data from the incoming bundle buffer to the endpoint, freeing old buffer space after successful delivery and triggering acknowledgements when necessary.
- In addition to these main building blocks, there are callbacks for different timeouts, such as the bundling timeout, which triggers data bundling if no data is received in a while and bundle still has available space.
- There are also other functions for option parsing, writing bundle headers into data buffers, for socksification, endpoint connection establishment and for printing diagnostics as well as for managing data structures. While necessary, they are not focal points of protocol operation and are not covered in deeper detail here.

The descriptions above avoid are not excessively detailed but instead provide a general outline of the implementation functionality, as the details in any case are subject to constant change and evolution as the code matures.

3.4.4. Design Issues Specific to Implementation

As mentioned before, certain aspects of implementation functionality concern less the communication protocol than the circumstances in which it operates. It makes arguably little sense for the protocol to concern itself with such external factors, and so there problems must be solved in some way or another by the implementation itself. Here such problems are considered to be the detection of connectivity and symmetric operation between daemons. Consideration of these topics follows.

3.4.4.1. Connectivity Detection

The DTN connection is inherently unreliable and may go down at any given moment. During a connection outage, sending anything through DTN is useless so sensible daemons should refrain from sending anything until connectivity is restored. However, this brings forth the problem of how to detect connection losses – and restorations.

There are two basic options for detecting outages: either using local resources or by sending periodic probes to the other daemon. In the first case, daemon might communicate with the host OS and try to detect network connection states and act accordingly. This option is not straightforward, as the host OS configuration is not uniform in different daemon instances, e.g. there might be different access network interfaces – or a different host OS altogether. Besides, this method only works if the disconnection-prone interface actually resides within the same host as the daemon. On the other hand, this approach would not generate excess traffic and would probably be quick to respond to connection state changes, of course depending on the underlying OS mechanisms.

The other option for outage detection is quite straightforward: daemons sending probe bundles to each other at regular intervals. Connection outages could then be detected with only a modest delay if a daemon stops receiving bundles from the other daemon. This approach generates some extra traffic on the network. Also, if bundles are being sent in any case to detect connectivity, those bundles might just as well be given a useful payload. Probe bundles could also get lost or delayed, possibly introducing additional complications to operation. A proxy daemon communicating with several client daemons also involves receiving several probes and generating replies to each probe, leading to possible scalability issues.

A further development of this latter approach would be to monitor incoming bundles from the other daemon and compare it to existing connection table. Not receiving any bundles from any given daemon could be indicative of a connection loss. Furthermore, suppose a client endpoint is the only communicating entity served by a client daemon, and is downloading a large file, thus sending nothing else than acknowledgements to the proxy daemon. In such an instance, the first symptom of a connection loss is a lack of acknowledgements from the client side and subsequent filling of the transmission buffer at the proxy side. Having the proxy keep track of time between arriving bundles will bring little benefit to the scenario.

In conclusion, no specific measures for detecting connectivity events will be implemented at least in the initial prototype. Actual performance of the prototype might provide clues to whether such a mechanism will be needed in possible later implementations.

3.4.4.2. Symmetric Operation

Keeping the daemon symmetric is fairly simple, with some assumptions about operating environment. Given the architecture specified in CHIANTI, there is an assumption of client-proxy functionality model and connections tend to be initialized by the client side. Another assumption is that each proxy serves several clients, and client daemons are therefore assumed to communicate with a single proxy daemon.

With these assumptions, symmetric operation for a daemon is reasonably easy to achieve: in client mode, daemon has a predetermined proxy daemon entity for relaying HTTP connections; in proxy mode, it will just have to maintain separate connection tables according to incoming DTN addresses, as each arriving bundle will have a source DTN address.

However, a true, generalized symmetry in the sense that proxy daemon should be able to relay client requests initialized in the Internet side to mobile endpoints in the access sphere, served by one of its client daemons, is much more difficult. Mobile clients and even the daemon entities might reside within NATted private networks. If both endpoints or worse yet both daemon entities reside behind NATted networks, finding endpoint (or daemon) IP addresses becomes problematic and demands MobileIP-style application of address tracking agents within the public Internet, adding considerable complexity to the protocol architecture and implementation.

Impact of NAT has been considered in CHIANTI project and documented in several deliverables (D1.2, D3.1); the CHIANTI architecture itself assumes that connections between CHIANTI components must be client-initiated and that proxies reside within publicly reachable Internet. This also implies that endpoint connections are always initiated from within NATted networks. With this specification as the reference guide, this work does not aim to provide symmetric operation in the strictest sense, but rather within the same scope as specified in the CHIANTI architecture.

4. Testing and Measurements

To assess the potential usefulness of the DTN reference implementation to the CHIANTI scenario a testing procedure needs to be defined. While this work is tentative and implementation of the software daemon immature and likely to evolve should the initial results be encouraging enough to warrant for further development. Also, the protocol designed for the HTTP-over-DTN functionality is minimal and the values chosen for the parameters regulating its performance lack rigorous analysis and testing, meaning that the protocol performance is almost assuredly suboptimal. Thus, the focus of the tentative testing is not rigorous analysis of protocol performance or accurate modeling of typical Internet experience or environment. Rather, the aim is to perform a simple series of tests under good conditions in order to assess impact of the DTN reference implementation to performance compared to plain HTTP traffic.

Subsection 4.1 describes the test setup and procedure for performance testing. Subsection 4.2 details the focus of the testing procedure and explains the measurements performed during the testing. Results are presented in the last subsection 4.3.

4.1. Test scenario and setup

For the test procedure, two Linux hosts running the software daemon and DTN reference implementation and a web server hosting files are used. The hosts reside in a LAN environment providing at least 100 Mbps transfer infrastructure between them, and the web server resides in the Aalto University network. This setup provides high transfer capacity and minimal delay and jitter environment, minimizing the impact of network conditions to the transfer performance. Unfortunately, available resources did not allow for a completely isolated test environment, as both the local LAN and especially the university server and its hosting network are subject to a degree of continuous network traffic which will be visible as minor fluctuations of performance. However, in general, network environment in the test setup can be considered close to ideal, considering the tentative nature of testing at this point.

One of the hosts is a desktop computer running the proxy daemon, the other host is a laptop running the client daemon. The laptop then runs the test scripts, relaying HTTP requests over the DTN link to the desktop host, which forwards the requests to the web server. The basic idea of the test scripts is to use wget, a simple program designed to perform file retrieval using HTTP, to download test files from a web server, first over plain HTTP, then through a DTN link using the software daemon as an HTTP proxy. Testing is done in several phases.

The first phase of the testing involves the transfer of a single large file of around 50 MiB from the web server. Downloading the file using wget with plain HTTP first gives a benchmark value for performance in ideal conditions, which acts as a reference when assessing file transfer performance of wget using DTN. Several runs are made using the DTN link, with the HTTP-DTN adaptation protocol configured with different values of protocol parameters. This is to perform a coarse tuning of the protocol parameters to gain some insight of their impact on performance and possibly to minimize the impact by selecting reasonable values.

The second phase of the testing procedure is done after achieving results from the first phase of testing and it is a simplified adaptation of the procedure used for actual CHIANTI performance testing as described in CHIANTI Deliverable D5.2 [34]. Wget is provided resource files which contain URLs pointing to test files. Test files are generated in different sizes, in an exponentially increasing sequence beginning with one kibibyte (1 KiB = 2^{10} bytes = 1024 bytes) and increasing in size by a factor of two, i.e. 2 KiB, 4 KiB, … , 1 MiB, 2 MiB. Requests are generated using exponential distribution with a mean value of 300 KiB, rounded to the nearest test file size, i.e. 300 KiB request translates to downloading the 256 KiB test file, while a 400 KiB request translates to the 512 KiB test file. Once again, testing is done first with wget using plain HTTP, then using the DTN link.

The final phase of testing involves creating artificial latency to network traffic between the proxy daemon and the web server. By introducing fixed latencies of various sizes to the connection between the proxy and the web server, observations about the latency caused by the DTN software can be made by first fetching a file of given size with wget through the delayed network without the DTN software in-between and then by fetching the same file with wget through DTN, and then comparing measured values of latency. By repeating measurements for different values of bundling timeout in the software daemon, more information about the impact of bundling timeout and TCP convergence layer mechanisms to latency can also be gained.

To evaluate the “real-life” performance of the DTN implementation and relay protocol from user perspective, some browser testing is also performed simply by browsing through some web pages with and without DTN enhancements and observing the subjective user experience in both cases.

4.2. Measurements

Testing phase one concerns itself mostly with the performance of the DTN transfer and the software daemon with different operating parameters. As such, the most important metric is the data throughput rate, i.e. the rate at which user receives files from the web servers. Throughput rate is reported by the wget at the end of each download and is used for performance assessment.

It would be illustrative to measure also the actual data transfer rate of the DTN link, but this is difficult as the DTN API does not provide access to read actual bundle or bundle header sizes and thus estimating the overhead in data transfer is difficult without painstaking analysis of all captured network traffic in the DTN link. The order of magnitude of overhead induced by the adaptation protocol can be easily estimated. In ideal conditions with no retransmissions it is very small, i.e. 24 bytes per bundle – around 0.05% for a 48 KiB bundle, and is certainly much less than the overhead from bundle headers. Acknowledgement bundles incur further overhead, but with ACK bundle sizes upwards of 12 bytes and one ACK sent for e.g. every four bundles this overhead too is next to meaningless for large bundle sizes. In any case, as the emphasis on testing is on getting qualitative results for indicators of development potential, therefore rigorous measurement of all possible overhead influence is not considered essential here.

For the first test run, the 50MiB target file is fetched with wget. Wget reports transfer rate for the file after a successful download, this value is used as a reference value for DTN test runs.

Daemon is configured to use different values of bundle buffer sizes: 8, 16, 24 and 32 bundles per connection. For each buffer size configuration, three sets of test runs are performed, each run using a different value for bundling timeout, i.e. the interval of time during which the daemon will wait for further incoming endpoint data before sending the current bundle. Timeout values of 10, 100 and 1000 milliseconds are used. For each run, the 50MiB target file is downloaded five times to be able to calculate an average value. Being a performance test, the highest throughput value for each run is also recorded as perhaps the better indicator of what the DTN implementation is capable of.

For the second stage of testing, ten different batches of exponentially distributed file sizes are downloaded with wget, first without and then with DTN software. This time, test runs are repeated with different values for bundling timeout to estimate their effect on transferring smaller files; the expectation is that in the first phase, bundling timeout should have little effect on the throughput rate of a file considerably larger than bundle size of 48 KiB used. In effect, only the last bundle should suffer from bundling timeout, as all earlier bundles should fill up with data at the same (high) rate as the plain wget is able to download the file. In the second phase, file sizes are closer to and even less than bundle size, and bundling timeout should have a clear impact on throughput rates.

Each run is repeated ten times for each bundling timeout value, timeout values are 10, 25, 50, 100, 250, 500 and 1000 milliseconds. Wget logs are examined for reported throughput rates for each file size, of these, minimum, maximum and average values are then recorded.

Last testing stage involves fetching files over a connection subject to increasing delay in order to provide further information on the effect of DTN on latency. Some of the applications used by mobile users might well be delay-sensitive, and so it is all the more desirable to keep additional latency introduced by service enhancements as low as possible. At the basic scenario, delay between the proxy daemon and the web server is about one millisecond. Single file is then fetched with wget, first 1 KiB and 32 KiB files, which fit well in a single bundle, and then 64 KiB and 1 MiB files, which involve sending two or more bundles, are fetched. A test script will record timestamp with sufficient accuracy before running wget and another one immediately after wget completes. Same files are then downloaded through DTN software with TCP convergence layer, using different bundling timeout values of 10, 100 and 1000 milliseconds. Each download is repeated five times, and the procedure is repeated for increased delay values of 10, 100 and 1000 milliseconds. It is assumed that delays incurred by factors independent of delay such as script processing remain approximately constant between different runs. The differences in timestamp values recorded can then be compared to find the extra latency due to DTN implementation for each test case.

5. Results

Downloading the 50 MiB test file with plain wget (no DTN), wget reported average throughput rate of 10.63 MB/s, with maximum reported throughput value being 11.02 MB/s. Considering underlying 100 Mbps switched Ethernet LAN, this is fairly close to full utilization of the network; 11.02MB/s = 88.16 Mbps, some bandwidth is wasted on the protocol overheads from Ethernet framing, TCP/IP headers and HTTP messages.

Table 1 below shows throughput rates for the 50 MiB file downloaded through DTN software, as reported by wget.

Table 1: Wget throughput downloading 50 MiB file using DTN

	Bundling timeout (ms)		
Buffer size	10	100	1000
8 – avg	644.91 kB/s	620.42 kB/s	577.41 kB/s
8 - max	*744.75 kB/s*	*818.80 kB/s*	*591.35 kB/s*
16 – avg	1.06 MB/s	1.17 MB/s	1.18 MB/s
16 – max	*1.18 MB/s*	*1.24 MB/s*	*1.22 MB/s*
24 – avg	1.29 MB/s	1.31 MB/s	1.24 MB/s
24 – max	*1.53 MB/s*	*1.51 MB/s*	*1.36 MB/s*
32 – avg	1.20 MB/s	1.21 MB/s	1.18 MB/s
32 – max	*1.34 MB/s*	*1.41 MB/s*	*1.33 MB/s*

The resulting throughput values do not compare favourably to the plain wget case; with DTN in place, throughput rate in the best case – 1.53 MB/s for 24 bundle buffer, 10 ms bundling delay – corresponds to about 14% of the best case without DTN. Furthermore, best results are achieved when 24 bundle buffer size is used – given 48 KiB maximum bundle size this would correspond to 1152 KiB of memory needed for buffering – per client connection.

To better assess the effects of bundling timeout to throughput in the second phase of testing, test results for files larger spanning more than one bundle are examined. As mentioned before, bundle size used was always 48 KiB.

For file sizes of 64 KiB and above, reported reference throughput results for plain wget are presented in the table 2 below. Values of throughput are given in MB/s as reported by wget. Minimum and maximum values were originally included to give some idea of performance fluctuations in the network. Corresponding reported throughput rates for same file sizes, downloaded through DTN, are presented in tables 3 through 9 below for bundling timeout values of 10, 25, 50, 100, 250, 500 and 1000 ms respectively.

Table 2: Reference values for throughput, plain wget

Throughput	File size					
(in MB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**9.52**	**10.02**	**10.26**	**10.43**	**10.69**	**10.31**
-min	8.67	7.41	8.96	7.79	9.27	8.83
-max	9.97	10.75	10.85	11.02	10.97	10.92

Table 3: Throughput for wget through DTN, bundling delay 10 ms

Throughput	File size					
(in KB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**620.43**	**565.64**	**430.58**	**545.73**	**637.92**	**713.89**
-min	424.68	436.37	382.40	408.26	459.07	621.24
-max	1100.00	828.54	523.63	658.36	812.34	841.20

Table 4: Throughput for wget through DTN, bundling delay 25 ms

Throughput	File size					
(in KB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**636.73**	**582.90**	**439.68**	**553.62**	**661.63**	**701.69**
-min	380.39	437.03	351.90	410.84	430.50	604.80
-max	1190.00	891.56	541.35	740.41	833.46	759.42

Table 5: Throughput for wget through DTN, bundling delay 50 ms

Throughput	File size					
(in KB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**543.04**	**552.43**	**430.57**	**569.85**	**675.48**	**765.81**
-min	408.64	446.36	377.13	433.12	504.74	665.93
-max	1040.00	711.06	501.88	723.91	919.30	833.24

Table 6: Throughput for wget through DTN, bundling delay 100 ms

Throughput	File size					
(in KB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**405.58**	**466.02**	**416.46**	**565.23**	**705.78**	**810.42**
-min	302.99	379.31	354.08	406.04	460.33	647.23
-max	756.34	576.88	484.05	705.97	901.48	919.38

Table 7: Throughput for wget through DTN, bundling delay 250 ms

Throughput	File size					
(in KB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**196.78**	**284.21**	**362.72**	**456.14**	**547.64**	**646.12**
-min	164.30	236.02	322.73	372.64	448.32	576.35
-max	246.92	328.22	412.41	542.30	664.53	709.09

Table 8: Throughput for wget through DTN, bundling delay 500 ms

Throughput	File size					
(in KB/s)	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**110.84**	**185.83**	**270.02**	**386.80**	**536.71**	**569.57**
-min	101.08	169.78	231.86	318.18	378.58	401.46
-max	123.01	207.01	300.73	446.77	621.08	678.62

Table 9: Throughput for wget through DTN, bundling delay 1000 ms

Throughput (in KB/s)	File size					
	64 KiB	128 KiB	256 KiB	512 KiB	1 MiB	2 MiB
-average	**59.24**	**106.76**	**176.67**	**278.71**	**420.70**	**564.19**
-min	55.46	100.75	156.05	244.75	343.07	510.39
-max	62.29	121.16	193.35	305.82	460.39	595.10

The results of measurements are summarized in the graph presented in figure 8 below.

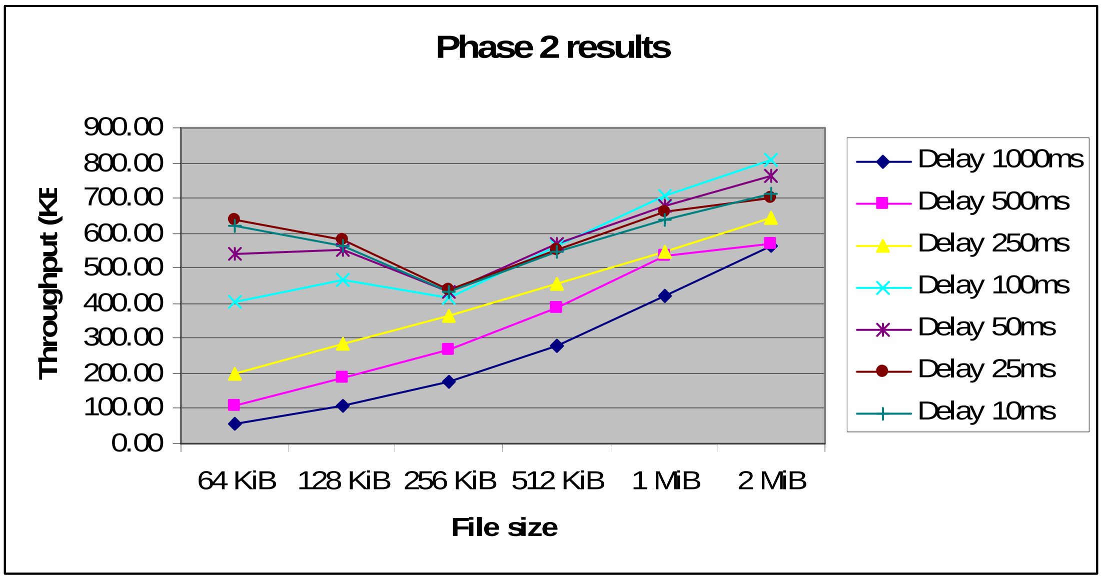

Figure 8: Graph summary of measurement results for phase 2.

From the results some insight to the effects of DTN implementation overhead may be gleaned. As expected, the reference throughput rates for plain wget downloads are high and comparable to plain wget performance in the first phase of testing. The effect of bundling delay to throughput is obvious, as smaller files are concerned – with 64 KiB files, increasing bundling delay from 10 to 1000 ms drops throughput rate to 10%. However, even with smaller values for bundling delay and larger files throughput remains somewhat disappointingly low, being less than 1 MBps even in the best case. Furthermore, throughput performance is best for larger files, as is expected. However, file objects in the Internet on the average tend to be smaller, of the order of tens or hundreds of kilobytes rather than of megabytes.

A closer examination of the reported throughput values reveals that, somewhat unexpectedly, 64 KiB files have experienced a better throughput rate than some of the larger files. This seems counterintuitive at first, as smaller files comprising of fewer bundles should be impacted more by the bundling delay, as it should only affect transmission of the last bundle and the proportionate effect of the bundling delay should be greater than with larger files. However, another observation is that 64 KiB files have only experienced better throughput rates for the lowest values of bundling timeout, and fluctuation between minimum and maximum throughput rates is large. Most likely this is a combination of effects of web server load and too aggressive bundling timeout causing transmission of extra bundles and thus extra latency; for larger files and longer bundling timeouts throughput fluctuations decrease, with bundling timeout of 100 ms providing best throughput measurements for larger files and only a slight decrease of throughput performance for smaller files.

The final set of measurements was designed to provide further information on the effect of DTN implementation and TCP convergence layer on latency. Tables 10 to 13 below list for each file size used latency in milliseconds for different combinations of (simulated) transmission delay and bundling delay. The bottom three rows of each table also list reference values for series of plain wget downloads. Different file sizes were chosen as 1 KiB, 32 KiB, 64 KiB and 1MiB. Smaller file sizes were chosen because they should fit into a single bundle, allowing for a better estimation of the effect of bundling delay to latency, while larger file sizes were chosen to provide better estimation of effects of DTN implementation and TCP convergence layer operations on latency.

Table 10: Latency for downloading a single file, size 1 KiB

Bundling	Delay (ms)			
Timeout	1	10	100	1000
10 – avg	**173.3**	**190.4**	**352.7**	**2163.2**
10 – min	157.8	162.1	345.8	2134.4
10 – max	192.2	231.4	375.4	2208.8
100 – avg	**331.0**	**348.6**	**531.9**	**2670.5**
100 – min	326.0	344.3	516.3	2319.0
100 – max	343.5	361.1	559.2	3672.5
1000 – avg	**2131.0**	**2173.0**	**2332.3**	**4154.4**
1000 – min	2120.8	2152.7	2318.1	4127.5
1000 – max	2145.1	2199.0	2364.2	4212.7
Ref – avg	***15.2***	***36.7***	***216.6***	***2018.3***
Ref – min	*8.2*	*28.5*	*214.6*	*2016.7*
Ref – max	*33.3*	*49.7*	*217.5*	*2024.1*

Table 11: Latency for downloading a single file, size 32 KiB

Bundling	Delay (ms)			
Timeout	1	10	100	1000
10 – avg	**179.6**	**219.6**	**659.4**	**5152.4**
10 – min	157.8	203.1	641.3	5142.4
10 – max	208.2	240.7	679.6	5169.4
100 – avg	**330.3**	**392.9**	**833.5**	**5329.9**
100 – min	328.8	376.1	831.0	5320.3
100 – max	331.3	417.0	836.2	5352.0
1000 – avg	**2143.7**	**2195.9**	**2635.5**	**7150.9**
1000 – min	2127.4	2182.4	2629.3	7136.3
1000 – max	2179.0	2227.7	2649.4	7169.2
Ref – avg	***23.3***	***68.3***	***526.7***	***5026.7***
Ref – min	*14.7*	*67.3*	*525.4*	*5025.9*
Ref – max	*39.7*	*69.5*	*528.5*	*5027.4*

Table 12: Latency for downloading a single file, size 64 KiB

Bundling Timeout	Delay (ms) 1	10	100	1000
10 – avg	**259.6**	**278.0**	**763.6**	**6150.0**
10 – min	241.0	265.9	752.5	6143.0
10 – max	284.6	320.3	797.1	6158.1
100 – avg	**396.3**	**444.9**	**929.9**	**6339.0**
100 – min	393.1	416.7	918.4	6322.9
100 – max	399.0	504.6	941.7	6379.3
1000 – avg	**2217.5**	**2234.8**	**2729.6**	**8129.8**
1000 – min	2199.9	2223.5	2725.7	8123.0
1000 – max	2237.3	2267.6	2733.8	8133.0
Ref – avg	***23.2***	***79.7***	***627.1***	***6027.2***
Ref – min	*15.4*	*76.5*	*621.8*	*6021.7*
Ref – max	*32.3*	*88.0*	*636.4*	*6036.7*

Table 13: Latency for downloading a single file, size 1 MiB

Bundling Timeout	Delay (ms) 1	10	100	1000
10 – avg	**734.0**	**873.4**	**2534.9**	**22187.5**
10 – min	621.7	795.5	2502.3	21175.3
10 – max	849.6	994.1	2608.2	26179.3
100 – avg	**957.0**	**1055.2**	**2628.1**	**21355.9**
100 – min	853.7	918.7	2597.5	21344.3
100 – max	1118.3	1193.6	2656.6	21362.2
1000 – avg	**2687.1**	**2895.7**	**4418.3**	**23241.7**
1000 – min	2651.1	2776.6	4385.0	23200.8
1000 – max	2732.9	3052.2	4448.6	23269.9
Ref – avg	***118.2***	***263.7***	***2153.0***	***21052.8***
Ref – min	*107.6*	*256.0*	*2147.7*	*21046.0*
Ref – max	*144.7*	*274.8*	*2160.3*	*21058.2*

Observing data gathered while downloading the 1 KiB file provides a good starting point to estimate the inherent delay inflicted by the DTN software, as it should involve sending only one bundle after bundling timeout has triggered. By subtracting from the measured latency twice the value of bundling delay (which takes place both at the client daemon as the initial endpoint request arrives and at the proxy daemon which receives the file from the other endpoint) and the reference value a rough estimate for DTN-induced latency can be obtained. From this, the increase in latency is around 130 ms for one bundle. Repeating the calculation for all values of 1, 10, 100 and 1000 ms of transmission delay yields latency values of 138.1, 133.7, 116.1 and 124.9 ms respectively. Further repetition of the same calculation for increasing bundling delays yields similar values for most cases. Earlier measurements show reduced throughput rates for traffic carried by DTN, at this point it is difficult to determine the degree of extra latency caused by decreased throughput and that caused by bundling overhead itself.

Results from the 1 MiB file transfer provide other possibilities for performance assessment. TCP performance typically begins to deteriorate as latency increases; comparing plain transfers of 32 KiB and 64 KiB files over a delay of 1000 ms, file transfer operation takes about 1000 ms longer to complete – yielding throughput rate of about 240 kbps. Same comparison for 1 KiB and 1 MiB files shows that transferring about 1 MiB of data over a 1000 ms delay takes about 19 seconds longer, which means throughput of approximately 440 kbps – significantly less than throughput DTN has earlier proved capable of.

With the long-delay transfers DTN is no longer a bottleneck; now, a closer examination of their measured latencies is in order. In fact, comparing every latency value for DTN transfers with the respective reference value reveals that in nearly all cases, the difference in latency is just over 100 ms – very close to the latencies around 130 ms calculated for 1 KiB file transfer. This would suggest that DTN implementation and its TCP convergence layer mechanism combined with the relay protocol implementation have a characteristic latency of around 100 ms. Sources of this latency include TCP connection establishment of the TCP convergence layer and bundle handling and management. The DTN reference implementation uses a database backend for bundle storage, quite likely a major source of latency. However, more accurate breakdown and analysis of component effect on latency requires more extensive and carefully designed measurements.

The induced extra latency is quite acceptable for bulk traffic transfers. For the more delay-critical real-time applications addition of another 100+ milliseconds of latency is potentially much more disruptive. Of course, on top of this, delay due to bundling timeout has to be added for every message which fails to fill a bundle, further increasing the negative impact.

Final browser testing supplies no additional quantitative results here, nor was it meant to do so. The most important result of browser testing was that the relay protocol actually managed to relay real web traffic. Subjective comparison of user experience between normal browsing and browsing through DTN was that browsing through DTN was perceptibly more sluggish than normal browsing, especially so when browsing web pages over high-capacity, low-delay network connection. In light of the more quantitative results gained earlier this is not surprising. However, while browsing through DTN was slower than plain browsing, degradation of service was fairly light even in worst cases and at no point could be considered unacceptable for normal use.

6. Conclusions

This work set out to test the potential usefulness and applicability of the DTN reference implementation for the purposes and goals set in the CHIANTI project, designing and implementing a simple protocol for multiplexing endpoint HTTP connections over a DTN link provided by said reference implementation. Simple measurements of key performance values of DTN communication have been performed in order to form an initial assessment of its usefulness to the project.

DTN is a communication architecture designed for robust communication over communication environment difficult to the extreme. As such, high throughput and low delay performance are not critical in a store-and-forward architecture, which also reflects on performance of the DTN reference implementation.

The performance measurement results for the DTN reference implementation using TCP convergence layer mechanisms coupled with the simple HTTP relay protocol implementation developed for this work compared with performance measurement without the DTN software have provided some insights to its performance in different conditions with respect to increased latency and decreased throughput.

Results show increased latency of at least 100 ms plus bundling delay and maximum achieved throughput of around 12 Mbps. Best performance values are achieved for relatively large files and for bundling delay value of 100ms. Qualitative browser testing has proved the concept workable in practice as well, extra latency not being too disruptive for casual use. The limited throughput is enough for serving a limited amount of users in a vehicle, and will saturate a 3G or a 11 Mbps 802.11b wireless uplink, although not a 54 Mbps 802.11b/g or a WiMAX link. All in all, the DTN reference implementation is useful, if not optimal.

Now that a prototype CHIANTI-compliant HTTP-over-DTN module has been developed and tested, trial integration with CHIANTI architecture remains to be done. With a DTN module place in a CHIANTI FlexProxy, final evaluation of its capabilities and usefulness could be made, possibly along with performance comparison against CHIANTI core tunneling modules. Before such comparison, it is worth investigating how much performance of the DTN implementation can be improved.

Performance of the DTN reference implementation suggests potential for future improvements, especially with regard to throughput. Investigating latency and throughput performance of UDP convergence layer is another possible option, as well as exploring effect of different database backends on performance. Furthermore, there are other lightweight, scaled-down DTN implementations in existence, such as the IBR-DTN; they might well perform better than the reference implementation and comparing their performance with the results gained here would be interesting.

Effects of factors such as bundling timeout, delay, buffer and file sizes to overall performance having now been briefly investigated, refinement of the HTTP relay protocol and its implementation also hold promise for improving performance. The prototype protocol implementation is rather crude, with emphasis on quick testing rather than optimal performance. Future versions of the protocol could experiment with adaptive bundling delay and bundle size depending on latency, duration, and possibly even jitter of an endpoint connection.

This work has been a case study of applying the DTN reference implementation to mobile Internet. A protocol for relaying HTTP traffic in bundles has been specified and implemented and its performance measured and results reported, with some suggestions for future work and improvements, for which this thesis should provide a useful basis.

References

[1] Saltzer, J, Reed, D, Clark, D. D; End-to-End Arguments in System Design. Second International Conference on Distributed Computing Systems, pages 509-512, April 1981

[2] Perkins, C; RFC 3344: IP Mobility Support for Ipv4, 2002; http://www.ietf.org/rfc/rfc3344.txt (24.4.2010)

[3] Johnson, D, Perkins, C, Arkko, J; RFC 3775: Mobility Support in Ipv6, 2004; http://www.ietf.org/rfc/rfc3775.txt (24.4.2010)

[4] Moskowitz, R, Nikander, P, Jokela, P, Henderson, T; RFC 5201: Host Identity Protocol, 2008; http://www.ietf.org/rfc/rfc5201.txt (24.4.2010)

[5] Schütz, S, Eggert, L, Schmid, S, Brunner, M; Protocol Enhancements for Intermittently Connected Hosts; ACM SIGCOMM Computer Communication Review; 2005

[6] Snoeren, A. C, Balakrishnan, H; An End-to-End Approach to Host Mobility; Mobicom ’00; 2000

[7] ISC Internet Domain Survey, Jan 2009; http://ftp.isc.org/www/survey/reports/2009/01 (24.4.2010)

[8] Kempe, G, Hutchinson, N. C; Networks without Borders: Communication despite Disconnection; IEEE AICT/ICIW 2006; 2006

[9] Mao, Y, Knutsson, B, Lu, H, Smith, J. M; DHARMA: Distributed Home Agent for Robust Mobile Access; IEEE INFOCOM 2005; 2005

[10] Ott, J, Kutscher, D; A Disconnection-Tolerant Transport for Drive-thru Internet Environments; IEEE INFOCOM 2005; 2005

[11] A. Seth, S. Bhattacharyya, S. Keshav; Application Support for Opportunistic Communication on Multiple Wireless Networks; 2005 http://blizzard.cs.uwaterloo.ca/keshav/home/Papers/data/05/ocmp.pdf (24.4.2010)

[12] Cerf, V, Burleigh, S, Hooke, A, Torgerson, L, Durst, R, Scott, K, Fall, K. Weiss, H; RFC 4838: Delay-Tolerant Networking Architecture, 2007; http://www.ietf.org/rfc/rfc4838.txt (24.4.2010)

[13] Delay Tolerant Networking Research Group website; http://www.dtnrg.org/wiki (24.4.2010)

[14] Saturn Observation Campaign India website: Saturn – Earth facts; http://soc06.tripod.com/id11.html (24.4.2010)

[15] Scott, K, Burleigh, S; RFC 5050: Bundle Protocol Specification, 2007; http://www.ietf.org/rfc/rfc5050.txt (24.4.2010)

[16] Berners-Lee, T, Fielding, R, Masinter, L; RFC 3986: Uniform Resource Identifier (URI): Generic Syntax, 2005; http://www.ietf.org/rfc/rfc3986.txt (24.4.2010)

[17] SourceForge website; http://sourceforge.net (24.4.2010)

[18] Doering, M, Lahde, S, Morgenroth, J, Wolf, L; IBR-DTN: An Efficient Implementation for Embedded Systems, 2008; http://chants.cs.ucsb.edu/2008/papers/p-4.pdf (24.4.2010)

[19] CHIANTI project website; http://www.chianti-ict.org/ (24.4.2010)

[20] European Commission CORDIS Seventh Framework Programme website; http://cordis.europa.eu/fp7/home_en.html (24.4.2010)

[21] Seifert, N; Description of Use Cases and Scenarios; CHIANTI project Deliverable D1.1, 2008; http://www-rn.informatik.uni-bremen.de/chianti/public/chianti-D1.1.pdf (24.4.2010)

[22] Seifert, N; Operational and User Requirements; CHIANTI project Deliverable D1.2, 2008; http://www-rn.informatik.uni-bremen.de/chianti/public/chianti-D1.2.pdf (24.4.2010)

[23] Ylikoski, P, Ott, J; System Architecture; CHIANTI project Deliverable D3.1, 2008; http://www-rn.informatik.uni-

bremen.de/chianti/public/chianti-D3.1.pdf (24.4.2010)

[24] Bergmann, O; Protocol Specification; CHIANTI project Deliverable D2.4, 2009; http://www-rn.informatik.uni-bremen.de/chianti/public/chianti-D2.4.pdf (24.4.2010)

[25] Leech, M, Ganis, M, Lee, Y, Kuris, R, Koblas, D, Jones, L; RFC 1928: SOCKS Protocol Version 5, 1994; http://www.ietf.org/rfc/rfc1928.txt (24.4.2010)

[26] Wood, L, Eddy, W. M, Holliday, P; A Bundle of Problems, 2009; http://personal.ee.surrey.ac.uk/Personal/L.Wood/publications/wood-ieee-aerospace-2009-bundle-problems.pdf (24.4.2010)

[27] Debian operating system website http://www.debian.org/ (24.4.2010)

[28] Debian ”Lenny” release information webpage http://www.debian.org/releases/lenny/ (24.4.2010)

[29] Mercurial source control management tool website http://mercurial.selenic.com (24.4.2010)

[30] OASYS Project pages hosted at SourceForge website http://sourceforge.net/projects/oasys/ (24.4.2010)

[31] Libevent – en event notification library website http://www.monkey.org/~provos/libevent/ (24.4.2010)

[32] VMware Workstation product page at VMware website http://www.vmware.com/products/workstation/ (24.4.2010)

[33] DTN2 Manual Table of Contents webpage hosted at SourceForge http://dtn.sourceforge.net/DTN2/doc/manual/index.html (24.4.2010)

[34] Bergmann, O, Gerdes, S; Final Trial Report; CHIANTI project Deliverable D5.2, 2010; http://www-rn.informatik.uni-bremen.de/chianti/public/chianti-D5.2.pdf (24.4.2010)