HELSINKI UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF COMMUNICATIONS AND NETWORKING
NETWORKING LABORATORY

Lauri Peltola

Enabling DTN-based Web Access: the Server Side

April 1, 2008

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

Supervisor: Prof. Jörg Ott

Author: Lauri Peltola
Title of Thesis: Enabling DTN-based Web Access: the Server Side
Date: April 1, 2008 **Pages:** 72

Department: Department of Communications and Networking
Professorship: S-38 Networking Technology

Supervisor: Prof. Jörg Ott

The networking landscape in which modern protocols must operate is no longer just the static, homogeneous Internet. As the demand for ubiquitous connectivity grows, the Internet stretches out to increasingly diverse environments, such as mobile ad-hoc networks. In these environments, certain assumptions that current Internet protocols rely on may not hold, thus making these protocols inefficient or even useless. Delay-tolerant Networking (DTN) is one approach to solving the problems that arise in such settings. In this thesis, our first objective is to conceptualize the mechanisms needed to enable web access in a DTN environment. More specifically, the goal is to run the Hypertext Transfer Protocol (HTTP) on top of the DTN transport protocol (i.e., the bundle protocol).

In a DTN environment, where connectivity may be intermittent and transmission delays long, it is important to avoid unnecessary round-trips between the communicating nodes. Consequently, HTTP is not directly applicable to DTN due to its conversational style of operation in which the resources of a web page are fetched one at a time. We adapt HTTP to the DTN environment by introducing the concept of resource bundling, which means that web resources are grouped together into larger aggregates in order to minimize the number of round-trips required to retrieve a web page.

The second objective of the thesis is to implement the resource bundling concept in a web server application. The server builds on two major open source software components that handle the low-level bundle protocol operations and form the basis of the HTTP server logic. We integrate these pieces and extend them with the high-level resource bundling logic to produce a native DTN web server. We also perform measurements on the server, verifying its adeptness for real-world deployment and proving that the resource bundling concept truly has a positive impact on the web browsing experience in challenged network environments.

Keywords: DTN, HTTP, bundle protocol, resource bundling, web server

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Lauri Peltola
Työn nimi:	Enabling DTN-based Web Access: the Server Side
Päivämäärä:	1.4.2008 **Sivumäärä:** 72
Laitos:	Tietoliikenne- ja tietoverkkotekniikan laitos
Professuuri:	S-38 Tietoverkkotekniikka
Työn valvoja:	Prof. Jörg Ott

Verkkoympäristö, jossa modernit protokollat joutuvat toimimaan ei ole enää vain staattinen, homogeeninen Internet. Verkkopalvelujen kysynnän kasvaessa Internet levittäytyy entistä monimuotoisempiin ympäristöihin, kuten mobiileihin ad-hoc verkkoihin. Näissä ympäristöissä toimivat verkot eivät välttämättä täytä tiettyjä ehtoja, jotka ovat edellytyksenä nykyisten Internet-protokollien käytölle. Tällöin näiden protokollien käyttö voi olla vaikeaa tai jopa mahdotonta. Delay-tolerant Networking (DTN) on eräs lähestymistapa, jolla voidaan ratkaista kyseisiä ongelmia. Tämän diplomityön ensimmäinen tavoite on mahdollistaa WWW:n käyttö DTN-ympäristöissä. Käytännössä tämä tarkoittaa HTTP-protokollan sovittamista DTN:n kuljetuskerrosprotokollan (”bundle protocol”) päälle.

DTN-ympäristössä yhteydet voivat olla katkonaisia ja tiedonsiirtoviiveet pitkiä, minkä vuoksi on tärkeää välttää turhaa edestakaista viestiliikennettä kommunikoivien noodien välillä. Normaalisti HTTP toimii siten, että se hakee www-sivuun liittyvät resurssit yksitellen. Tämä nimenomaan aiheuttaa turhaa liikennettä, joten HTTP ei suoraan sovellu DTN-ympäristöön. Työssä määritellään käsite ”resource bundling”, jonka avulla HTTP voidaan sovittaa paremmin DTN-yhteensopivaksi. Perusidea on koostaa www-sivun resurssit yhteen pakettiin, jolloin sivun noutamiseen tarvittavien edestakaisten protokollaviestien määrä saadaan minimoitua.

Työn toinen tavoite on toteuttaa www-palvelinohjelma, joka tukee työssä määriteltyä ”resource bundling”-konseptia. Palvelin pohjautuu kahteen vapaan lähdekoodin ohjelmakomponenttiin, jotka ovat vastuussa alemman tason protokollaoperaatioista sekä HTTP-palvelimen perustoiminnoista. Integroimalla nämä komponentit ja kehittämällä resurssien käsittelyyn liittyvän korkeamman tason logiikan, työssä toteutetaan natiivi DTN-pohjainen www-palvelin. Työssä myös suoritetaan mittauksia, joilla varmistetaan palvelimen soveltuvuus todelliseen käyttöympäristöön ja lisäksi todetaan, että suunniteltu konsepti todella parantaa WWW:n käyttömahdollisuuksia haastavissa verkko-olosuhteissa.

Avainsanat: DTN, HTTP, bundle protocol, resource bundling, web server

Table of Contents

Abbreviations

AA	Application Agent
AJAX	Asynchronous Javascript and XML
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BPA	Bundle Protocol Agent
CLA	Convergence Layer Adapter
CRLF	Carriage Return, Line Feed
CSS	Cascading Style Sheets
DOM	Document Object Model
DoS	Denial-of-Service
DTN	Delay-Tolerant Network(ing)
DTNRG	Delay-Tolerant Networking Research Group
EID	Endpoint Identifier
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IP	Internet Protocol
IPN	Interplanetary Internet
MHTML	MIME encapsulation of aggregate HTML documents
MIME	Multipurpose Internet Mail Extensions
S/MIME	Secure/Multipurpose Internet Mail Extensions
SDNV	Self-Delimiting Numeric Value
SMTP	Simple Mail Transfer Protocol
SSL	Secure Sockets Layer
SSP	Scheme-Specific Part
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WLAN	Wireless Local Area Network
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

1 Introduction

Internet access has become ubiquitous in the past decade and this growth can only be expected to continue. We are no longer required to sit in front of a personal computer with a wired network connection to use the Internet. Instead we have mobile devices with the capability to access the Internet wirelessly and conveniently almost anywhere.

From a networking perspective, unconstrained mobile Internet access is problematic. The protocols that the Internet relies on were developed for a network that is more or less well-connected. These protocols assume that there is a continuous end-to-end path between the communicating nodes, and that the delays and error rates are relatively low. In a mobile network environment these assumptions often may not hold and conventional protocols may perform poorly or fail completely.

Delay-tolerant networking (DTN) is an emerging research area that focuses on enabling communications in difficult and unpredictable network environments. The Delay-Tolerant Networking Research Group (DTNRG) has developed a network architecture that is designed to work in situations where traditional protocols would fail: in networks with intermittent connectivity, long delays, high error rates and asymmetric data rates. A network conforming to the DTN architecture, often simply called a DTN, is an overlay network that can comprise multiple regional networks, each of which may use different lower-layer technologies. Thus, the DTN overlay provides a common ground for applications that might not otherwise be able to communicate.

As Internet access becomes more and more common in increasingly diverse settings, the underlying networking landscape changes. It transforms from a relatively static, homogeneous environment to a dynamic, heterogeneous one. Such an environment calls for new architectural concepts and protocols, as the ones we currently have cannot be efficiently used. The DTN architecture is one possible solution to the problems arising from this evolution, and therefore, a possible future platform for many Internet applications – those existing today and those yet to be invented.

1.1 Problem Statement

Web browsing is arguably the killer application of the Internet today, and in the foreseeable future. However, the Hypertext Transfer Protocol (HTTP), which is the protocol used for transporting web content on the Internet, has properties that make its deployment problematic in challenged network environments. These properties are related to both the underlying transport protocol – which most often is the Transmission Control Protocol (TCP) – and the way in which HTTP itself operates.

TCP cannot work unless there is an uninterrupted end-to-end path between the communicating nodes during the entire session. In mobile networks, for instance, topology changes cause frequent disruptions, thus breaking end-to-end connectivity. Setting up, maintaining and tearing down a TCP connection also involves sending numerous signaling messages. In an error prone, high delay environment this kind of conversational protocol behavior is impractical, if not impossible.

HTTP is also a conversational protocol in the sense that retrieving a single web page often requires several round-trips from the client to the server. In networks with very long delays, this is obviously detrimental to the user experience. This behavior, however, is not inherent to the protocol but rather a consequence of how web pages are constructed: a web page typically consists of multiple resources and instead of retrieving all of these at once, HTTP fetches them one-by-one.

In DTN, the transport protocol used in the overlay network to carry application data is called the *bundle protocol* and the protocol messages are called *bundles*.[1] A bundle – one of the most essential concepts in the DTN architecture – is an independent data unit of arbitrary size. Bundles are transported through the network by the bundle protocol using a set of mechanisms that can overcome possible delays and disruptions.

In this thesis, our aim is to enable DTN-based web browsing, i.e., make HTTP work on top of the DTN architecture. This problem needs to be approached from two perspectives [1]:

(1) Transport perspective, i.e., how to replace TCP with the DTN bundle protocol, while retaining necessary protocol features.

(2) Application perspective, i.e., how to make use of HTTP in a way that deals gracefully with disruptions and delays that inevitably occur in a DTN environment.

1.2 Objectives and Scope

The objective of this thesis is twofold. The first objective is to tackle the issues (1) and (2) on a conceptual level; the second one is to implement them in a web server application.

[1] We use the term "DTN" rather liberally in this thesis. We may use it to refer to either the general concept of delay-tolerant networking or an implementation of the DTNRG architecture. We assume that, given the context, the reader will be able to distinguish between the two.

Issue (1) is rather straightforward. The bundle protocol provides us with means of transmitting HTTP messages reliably through a network. The main issue to consider is addressing, i.e., how are HTTP addresses, which are found in the requests that a web browser generates, mapped to bundle layer addresses in order to route the messages to the correct destination. Carrying HTTP messages in bundles can be done trivially by just placing the message in the bundle payload. However, we will find out that this is not enough: in order to change the HTTP resource retrieval behavior from "one-by-one" to "all-at-once", we must define a way to aggregate multiple resources into a single bundle. We call this *resource bundling*. It is a fundamental concept when tackling issue (2).

Resource bundling means that instead of transporting web resources separately, i.e., one per application layer protocol message, we find relations between resources and bundle those resources that are related into a single message. This is the task of the web server: when it receives a request, it deduces the relations of the requested resource and creates the response bundle. This concept makes web browsing more DTN-friendly: bundling resources together reduces the amount of messages sent through the network and it also implies that disruptions in connectivity can be better tolerated. This is due to the fact that because the client receives more data with each incoming message, it is more likely that the resource the client would want to request during a disruption has already been received.

Although the idea is simple, the problems that arise are manifold. The questions that we must consider include:

(1) How do we find out which resources are related in a meaningful way, i.e., which resources should go into a single bundle?

(2) If the each resource is not carried in a separate HTTP message, how do we retain the resource-specific metadata that the HTTP headers usually contain?

(3) Do we create mechanisms to facilitate caching of content in intermediaries, which would be highly beneficial in a DTN?

(4) How do we deal with resources that are not well suited for bundling, e.g. dynamic web pages that do not reside on the server's disk but are created on the fly.

(5) How do we address security? Resource bundling implies that more burden will be put on the server which creates opportunities for, e.g., denial-of-service attacks.

We limit the scope of this thesis by mostly omitting issues (4) and (5). These subjects are complex from both conceptual and practical perspectives. We discuss some of the problems involved with them but do not necessarily provide any solutions. Also, our focus is primarily on the server side – we do not address details concerning the client (i.e., a web browser).

Finally, we implement a web server that has support for the designed HTTP bundling scheme. This serves as a validation of the ideas presented in the thesis and, hopefully, produces a useful piece of software. The idea is not to build a new web server from scratch – instead we extend an existing open source server. This way we get a tested, stable web server for free, and are able to fully concentrate on the bundle operation logic.

There are no client applications that could be used with the server and implementing one is far beyond the scope of this thesis. However, we do implement a proxy application that translates between HTTP and the bundle protocol, and supports the devised bundling scheme. With the proxy, a regular web browser can be used for testing our server. We also carry out a set of measurements to evaluate the server performance.

1.3 Related Work

The concepts presented in this thesis are largely based on the work by Ott and Kutscher in [1, 2]. They introduce the ideas of sending aggregates of HTTP resources in bundles and describing resource dependencies with metadata files on the server. A similar approach to enabling communications under intermittent connectivity, based on proactive data retrieval (i.e., prefetching) and bundling, has been discussed in [3]. Web prefetching has also been a research subject outside the context of DTNs, e.g., [4, 5]. Prefetching schemes like this have been implemented, for example, in Mozilla-based web browsers [6].

Wood and Holliday have taken a different approach to enabling DTN-based web access [7]. They suggest using HTTP directly as a hop-by-hop store-and-forward protocol, without the bundle protocol layer. The draft also includes a mechanism for grouping resources into larger aggregates.

Caching in DTNs has been studied in [8, 9], with a focus on leveraging the DTN infrastructure to create a massive distributed storage.

1.4 Outline

In this chapter, we have given a brief introduction to the topic and objectives of this thesis. In the following chapter, we discuss the concept of delay-tolerant networking and the DTNRG architecture in more detail. This discussion, along with the introduction of HTTP in chapter 3, equips us with the necessary theoretical background on which the contributions of the thesis work can be built. Further in chapter 3, we present how HTTP-over-DTN works from the standpoint of a web server. We also briefly discuss some related issues that are not within our primary focus but still relevant to the topic. In chapter 4, we introduce our implementation of the DTN-enabled web server. The concepts presented in earlier chapters are put into practice in the implementation. In chapter 5, we present the results of measurements performed on the server and discuss its viability for real-world deployment. Finally, the last chapter concludes the thesis by reviewing what was done and pointing out potential directions for future development.

2 Delay-tolerant Networking

A delay-tolerant network is an overlay on top of a number of diverse regional networks, including the Internet. Within a DTN, the regional networks may have varying connectivity, delay and loss characteristics, and may employ different lower-layer technologies. The DTN overlay accommodates these different network characteristics and provides a service that works regardless of the underlying network environment.

This chapter describes the DTN architecture and the relevant concepts related to it. We motivate the reasons for DTN, present the operation of the primary DTN protocol – the bundle protocol – in detail, and briefly introduce the DTN2 reference implementation.

2.1 Motivation

The need for a delay-tolerant networking architecture stems from the fact that there are situations in which the existing Internet protocols do not work well, or may even fail completely. Protocols, such as the ubiquitous TCP, make certain assumptions about the characteristics of the underlying network: that an end-to-end path exists between the communicating nodes, that the maximum round-trip time between nodes is not excessive and that the data loss rate is low. When these assumptions hold true, the protocols perform well. When they do not, however, protocol performance can be severely degraded. Networks, that violate one or more of these assumptions, are called *challenged networks*. [10]

In today's Internet, network access is usually based on high-bandwidth wireline links (with perhaps a single wireless hop from an end-node to a wireless access point), and the aforementioned assumptions are usually met. But if we consider, for example, interplanetary communication it is clear that the network characteristics are completely different.

The transmission delays of data signals are ultimately limited by the speed of light. On Earth, where data is sent over fairly short distances, propagation delays are just fractions of a second, whereas in space, the delays may be several minutes or hours. At the speed of light, the round-trip propagation delay between Earth and Mars ranges from 8 minutes to over 40 minutes [11]. Clearly, in such an environment using a conversational protocol like TCP would be highly impractical.

Interplanetary communication is definitely not the only application where using the DTN architecture would be beneficial; many examples of these situations can also be found on Earth. Mobile users will often experience disruptions in connectivity as they move through areas that fall within the range of an access point and other areas that do not. Protocols that rely on an end-to-end path being available for the duration of the communication session will fail in such circumstances.

Another benefit of the DTN architecture is that it creates a common layer on top of different regional networks. This allows applications to communicate across multiple, possibly heterogeneous regions by using the common overlay.

2.2 DTN Architecture

The DTN architecture is envisioned to solve the problems that prevent using existing Internet protocols in challenged networks. The basic premise of DTN is to introduce a new protocol layer, based on an abstraction of message switching, on top of the underlying networks' transport layers. Cerf et al. provide a set of design principles for the Interplanetary Internet in [11]. These principles are realized in the DTN architecture and discussed briefly below.

Bundles

In DTN, application data is carried in variable-length messages, called bundles. Bundles are sent through the network using store-and-forward operations: nodes along the communication path hold bundles in persistent storage until the next hop becomes available. This means that an end-to-end path need not exist when the bundle is initially sent; instead, the bundle is opportunistically moved closer to the destination (according to some metric), hop-by-hop.

Actually, IP networks are also based on store-and-forward operation. However, there is an assumption that the packets will only be stored briefly in the transmission queue. In DTN, nodes are expected to hold bundles in storage for longer periods of time while waiting for the opportunity to forward them. Typically, the stored bundles will survive system crashes and restarts. The DTN bundle protocol is discussed further in detail in section 2.3.

Endpoints Identifiers and Registrations

All nodes in the DTN are identified by a unique *endpoint identifier* (EID), which conforms to the Uniform Resource Identifier (URI) syntax as specified in [12]. Typically, the mapping between DTN nodes and EIDs is one-to-one, but in the case of multicast

or anycast, an EID may refer to multiple DTN nodes. In other words, a single EID may point to one or more nodes, and a single node may have more than one EID. Every node must, however, be a member of at least one "singleton" endpoint, i.e., have a unique (singleton) EID. The following is an example of an EID:

```
dtn://host.dtn/path
```

The first part of the EID before the colon is the scheme name, followed by the scheme-specific part (SSP). Currently, it is not clearly specified how exactly the SSP should be used in conjunction with DTN [13].

A *registration* binds an application to an EID. When an application establishes a registration with a DTN node, it signals that it wishes to receive bundles sent to a particular EID. When the node receives bundles destined to the registered EID, they are handed over to the application that performed the registration.

DTN also incorporates the concept of *late binding*, which means that the mapping between a destination EID and its lower-layer address is not necessarily performed at the source node. This is in contrast to the name resolution process of the Internet, in which the destination IP address is determined at the source before data is sent. In DTN, the mapping may occur at the source, during transit or possibly at the destination. This is advantageous because the transit time of a bundle may exceed the validity time of a binding. Use of name-based routing with late binding may also reduce the amount administrative traffic in the network. [14]

Priority Classes

The DTN architecture provides three different priority classes for bundles: bulk, normal and expedited. The priorities are only relative to each other – bulk bundles are of lower priority than normal or expedited bundles – with no concrete guarantees in the quality of service. The different classes only provide means to prioritize certain bundles over others, e.g., in a queue at a sending node.

Priorities are not required to (although they may) be enforced over different sources, i.e., an expedited priority bundle from one source may not be delivered with better service than a normal priority bundle from another source.

Custody Transfers

In DTN, end-to-end reliability cannot be implemented at the transport layer because there might not be a single transport protocol operating end-to-end (or an end-to-end path might not exist at all). Instead, reliability must be implemented at the bundle layer.

DTN supports node-to-node retransmission by means of *custody transfers*, effectively creating hop-by-hop reliability. A custody transfer is initiated by the source application. When it sends a bundle, it requests a custody transfer and starts a time-to-acknowledge retransmission timer. If the next hop node accepts custody, it returns an acknowledgment to the sender. If no acknowledgment is returned before the retransmission time expires, the sender retransmits the bundle. A custodian node must store

the bundle until another node accepts custody or the bundle's time-to-live expires. The custody of the bundle traverses the network until the final destination is reached or the bundle is discarded. [15]

The custody transfer mechanism by itself does not guarantee end-to-end reliability. If the bundle gets lost along the way (e.g., due to time-to-live expiry), the source node will not be aware of it. To be sure that the bundle went through, the source must also request *return receipt*, so that a confirmation is sent when the bundle reaches its intended recipient.

Fragmentation

DTN supports fragmentation and reassembly of bundles in order to improve the efficiency of transfers. Fragmentation allows the bundle protocol to fully utilize the available link capacity and to avoid retransmissions of partially sent bundles. [14]

There are two forms of DTN fragmentation: *proactive* and *reactive*. Proactive fragmentation is performed when a DTN node knows in advance (or predicts) that sending multiple fragments, rather than a single large bundle, is more likely to succeed. For example, a node might know that a regularly occurring downstream contact is always of such a short duration that the entire bundle cannot be sent entirely. In this case the node would proactively fragment the bundle and transmit the fragments during consecutive contacts.

Reactive fragmentation occurs when a lower layer indicates that some of the transmitted bytes were successfully transferred, but the entire bundle was not. The previous-hop node may then retransmit the missing portion. With both fragmentation types, the fragments are only reassembled at the final destination. Bundle fragments may also be further fragmented along the way, either proactively or reactively. [16]

If an application wants to prevent fragmentation it can set a "do not fragment" flag in the bundle. This may be useful, for example, when the integrity of a bundle is protected by a digital signature.

Routing

Currently, no routing protocol has been defined to be used in conjunction with the DTN bundle protocol. However, a number of protocols have been proposed and also implemented. Different routing schemes have their respective strengths and weaknesses in particular application areas. The fact that there are currently no large-scale deployments of DTN makes the evaluation of routing protocols difficult. Simulations can easily lead to false conclusions as they can never entirely capture everything that is involved in a real-life deployment. [16]

There are two simple forms of routing that do not require a routing protocol of any kind: *static routing* and *flooding*. In static routing, the routes are configured manually in each node before system startup. All nodes simply have a static list of other nodes to which they have a route and transmit data accordingly. Such a routing scheme is clearly not suitable for a real DTN environment where connections between nodes are

unstable. Static routing is mainly useful for experimentation and application development in a controlled environment.

In flooding, a node transmits incoming data to all links except the one from which the data arrived. Thus, the data is flooded throughout the network and eventually reaches the destination. There must be some kind of a mechanism that detects loops, i.e., prevents nodes from sending the same data over and over again. The obvious downside of flooding is that it generates loads of unnecessary traffic that consumes resources that may be scarce in a DTN (e.g., bandwidth, power, storage capacity). Routing information does not propagate through the network in either static routing or flooding, so no routing protocol is involved.

An example of an actual routing protocol is PRoPHET (Probabilistic Routing Protocol using History of Encounters and Transitivity). PRoPHET is based on epidemic routing which essentially is flooding with some variations to reduce the overhead [17]. However, instead of assuming random movement of nodes in the (mobile ad hoc) network, PRoPHET exploits observations made on the non-randomness of human mobility. Using these mobility patterns as the basis for routing decisions yields a probabilistic routing scheme that, at least theoretically, performs better than random epidemic routing. [18]

Security

The DTN Security Overview document [19] presents a number of threats that the DTN security mechanisms are designed to counter. These threats include malicious resource consumption, denial-of-service attacks, and bundle confidentiality and integrity. Because scarcity of resources is typical for a DTN, attacks that are devised to exhaust resources are a serious concern. Only those DTN nodes authorized to send bundles should be able to do so. Unauthorized bundles should be caught early in order to prevent consumption of valuable resources. Confidentiality and integrity of bundles must be ensured so that bundle contents cannot be read or changed while in transit. This requirement implies a need for key management, which is one of the open issues in DTN security [20].

DTN security is accomplished via the use of three independent security-specific bundle blocks, which may be used together to provide multiple bundle security services or independently of one another, depending on perceived security threats, mandated security requirements, and security policies that must be enforced [21].

The bundle authentication block (BAB) is used to assure the authenticity and integrity of a bundle on a hop-by-hop basis. This allows nodes to discard bundles from unauthorized sources. The payload security block (PSB) provides the same security service on an end-to-end basis (from a “security-source” to a “security destination”, which may or may not be the actual bundle source and destination nodes). The confidentiality block (CB) can be used to encrypt (parts of) a bundle. [20]

As with many other things in DTN, security is an on-going research topic with a number open issues.

2.3 Bundle Protocol

The primary protocol used in DTN is called the bundle protocol. As we have mentioned earlier, a bundle is the basic, self-contained data unit in DTN – variable in size and contains all the data and signaling required to traverse the network. Bundles are transmitted through the DTN in a store-and-forward fashion where intermediate nodes may store bundles even for long periods of time.

The DTN protocol stack is shown in figure 2.1. The bundle protocol operates on top of the transport layer which may be different in the network regions that the DTN overlay spans. A *bundle router* is a node that forwards bundles to other nodes either within a single DTN region or between multiple regions. In the latter case the node is sometimes called a *bundle gateway* [15]. In the figure, the left hand side depicts the Internet which uses the TCP/IP protocol suite. The right hand side is an arbitrary network that could use any lower layer protocols (provided that the transport layer is supported by the bundle protocol, i.e. a convergence layer adapter exists). The bundle overlay binds these different lower layers together. The applications do not need to be aware of the transport layer; they only communicate with the bundle layer.

A bundle node is an entity that can send and receive bundles – typically, a process running on a general-purpose computer. Each bundle node has three conceptual components (figure 2.2): a *bundle protocol agent*, zero or more *convergence layer adapters* and an *application agent*. [21]

The bundle protocol agent (BPA) is the component that implements the bundle protocol. It has an interface through which the application agents (AA) can use the services of the bundle protocol. The BPA handles the low-level functions of processing bundles while the AA may concentrate on the application logic. A convergence layer adapter (CLA) sends and receives bundles on behalf of the BPA. It provides an interface for the BPA to a certain transport protocol, making the BPA independent of underlying layers. A bundle node may have several CLAs and thus it may utilize multiple transport protocols.

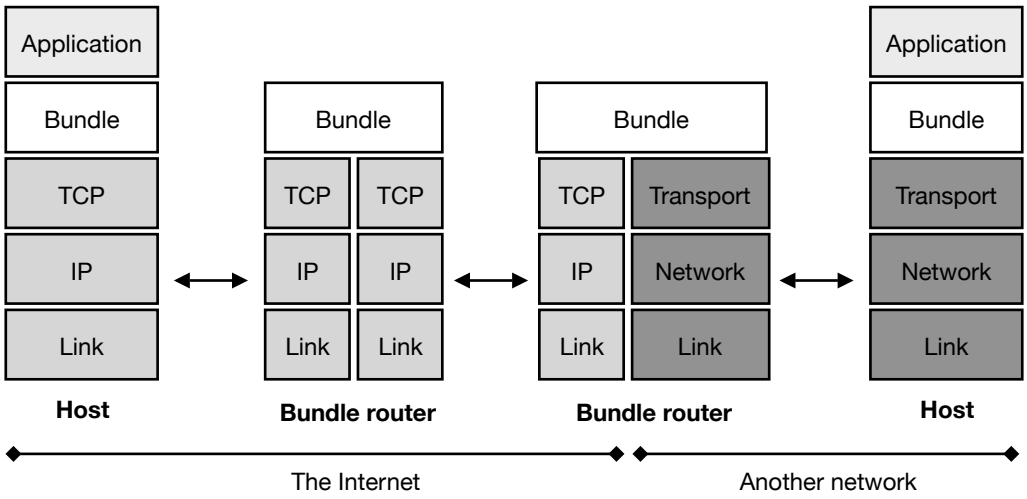


Figure 2.1: DTN protocol stack

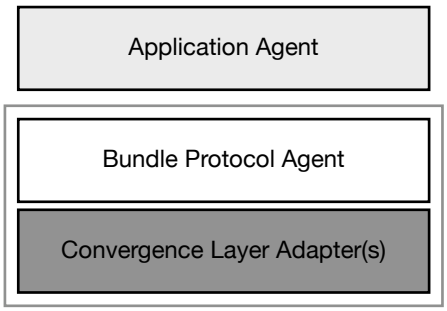


Figure 2.2: The conceptual structure of a bundle node

Bundle Structure

Bundles consist of *blocks*, which are roughly equivalent to what other protocols usually call headers. Each bundle has a primary block, a number of extension blocks and a payload block. The primary block contains the information needed to deliver the bundle to its destination, such as destination and custodian EIDs, creation timestamp and bundle lifetime. The application data (i.e., the payload) carried by the bundle is contained in the payload block, which is always the last block in the bundle. A special case of payload data are administrative records which provide some features of the bundle protocol, namely, bundle status reports and custody signals [21].

All blocks other than the primary or payload blocks are called extension blocks. The bundle protocol itself does not specify any extension blocks – instead, as the name suggests, they provide a mechanism extend the bundle protocol to suit various needs. Consequently, all DTN nodes will not be able to process all extension blocks. The default behavior with unknown blocks is to just skip them, but the source node can specify additional rules for such cases by setting certain processing flags in the block.

Many fields in the bundle blocks use self-delimiting numeric values (SDNV). The SDNV encoding scheme turns non-negative numeric values into octets with 7 value bits. The most significant bit of each octet is used to determine whether that octet is the last one (bit value 0) or not (bit value 1). The following is an example of encoding the hexadecimal value 1234 into SDNV.

```
1234 = 0001 0010 0011 0100   →   10100100 00110100
```

This scheme makes it possible to make the block fields arbitrary in size and to encode short values efficiently. The field values are commonly in the 0–32 bit range, so using SDNVs is justified. For values longer than 56 bits, a simple "one length octet, then value octets" scheme is actually more efficient than SDNV.

Convergence Layer Protocols

The bundle protocol can run on top of a variety of transport protocols, such as TCP. The mapping of the bundle protocol to lower layer protocols is performed by a convergence layer adapter (which implements a convergence layer protocol). Thus, enabling DTN on top of a given transport protocol is a matter of developing a suitable CLA.

The bundle protocol specification [21] summarizes the services that a CLA must provide for the bundle protocol agent:

(1) Sending a bundle to all bundle nodes identified by a given EID that are reachable via the convergence layer protocol.

(2) Delivering a bundle sent by a remote bundle node via the convergence layer protocol to the bundle protocol agent.

For reliable transport protocols (e.g., TCP), the convergence layer protocol may be fairly simple and just provide means for connection management and message delimiting. For unreliable protocols (e.g., UDP), the convergence layer should implement its own mechanism for reliability. Conceptually, a convergence layer protocol is an application layer protocol: it operates on top of the transport layer and utilizes transport layer mechanisms to provide a "bundle-enabled" interface for the higher-layer protocol (i.e., the bundle protocol).

A draft exists for the TCP convergence layer protocol (TCPCL) [22]. It defines the basic operations needed to enable TCP-based bundle transmission: connection setup and teardown, and bundle boundary marking. When establishing a TCPCL connection, a *connection header* is exchanged in order to set the connection-related parameters and to recognize the bundle layer identities (i.e., singleton EIDs) of both nodes. Bundle data is sent in segments, each of which have a header containing the length of the segment and the byte range of the bundle data. The first and last segments for a single bundle are marked.

Optionally, the receiving node may send acknowledgments for arriving bundle data segments. This allows the sender to perform reactive fragmentation (i.e., send only the missing parts of a bundle) in case an interruption occurs. The receiver may also send a negative acknowledgment in order to force the sender to stop transmitting the current bundle. This is useful if the receiver knows that it already has that bundle – interrupting the transmission saves capacity for other bundles. Messages for keeping an idle connection alive and for tearing down a connection are also defined in TCPCL.

TCPCL has been implemented in the DTN2 reference implementation, along with a number of other convergence layer protocols.

2.4 DTN2 Reference Implementation

DTN2 is an implementation of the DTN architecture and bundle protocol, developed by the DTNRG. The goal of DTN2 is to provide a platform for researchers to experiment with, and also to have production-grade code ready for real-world deployments [23].

DTN2 implements most aspects of the bundle protocol and evolves rapidly as things in the DTN research field progress. The current version includes support for TCP, UDP, Ethernet, Bluetooth and Sneakernet convergence layers.[1] DTN2 is written in C++ and

[1] Version 2.5.0 on April 1, 2008

includes a fairly simple API for writing applications that utilize the bundle protocol. There are also other implementations of the bundle protocol, e.g., RDTN [24], but these are generally not as mature as DTN2.

We use DTN2 as the bundle protocol agent in our implementation of a DTN-enabled web server. Our application does not depend on DTN2, though – any implementation of a BPA that provides the required services could be used.

2.5 Summary

In this chapter, we introduced the concept of delay-tolerant networking. A DTN is an overlay network that is designed to work in challenged network environments, including those with intermittent connectivity, long delays and high error rates. Traditional Internet protocols, such as TCP, cannot work efficiently, or at all, in these environments due to numerous reasons.

In DTN, application data is transmitted in variable-length messages called bundles. Bundles are carried through the network hop-by-hop so that intermediate nodes may store the bundles for long periods of time while waiting for the next-hop link to become available. An end-to-end path is not a prerequisite for sending data.

The primary protocol used in DTN is the bundle protocol which operates on top of different transport layer protocols by using different convergence layer adapters. A DTN overlay running the bundle protocol may encompass multiple heterogeneous networks. The bundle protocol abstracts the underlying technologies and provides DTN applications a common layer to communicate with.

In DTN, it is important to avoid unnecessary round-trips in the network. The bundle protocol is designed to accommodate this fact – it keeps the amount of exchanged messages at a minimum. The applications that run on top of the bundle protocol should also aim to operate in this manner. HTTP, however, is a conversational protocol. This implies that it cannot be run over the bundle protocol trivially by just wrapping HTTP messages in bundles. In the next chapter we discuss how HTTP could be adapted to a DTN environment to facilitate DTN-based web access.

3 Bundling Web Content

Web browsing, as it works in today's well-connected Internet, is not very well suited for challenged networks environments. Web pages typically contain multiple embedded resources, such as images and stylesheets. Regular HTTP-over-TCP will fetch these resources one-by-one, thus requiring several round-trips from the client to the server just to receive a single page. When round-trip delays are short, as they usually are in the Internet, everything works well. However, in a DTN environment where delays might be significantly longer, it is clear that a different approach is needed, as one-by-one retrieval of resources would make web browsing a frustrating experience.

The basic idea of avoiding these unwanted round-trips is simple: instead of returning the resources for a web page in multiple consecutive responses, the resources can be bundled together and returned in a single response. In order to achieve this, we must define how individual resources can be aggregated together into larger data structures, and how these structures are placed into bundles. We must also provide means for the server to identify resource dependencies so that in addition of knowing *how* to bundle resources, the server also knows *what* to bundle.

In this chapter we first present an overview of HTTP and briefly discuss the distinction between static and dynamic web content. We tackle the issues related to resource bundling from transport and application layer perspectives, i.e., how web resources are transported over the DTN bundle protocol and what applications can do differently in order to adapt to the DTN environment. We also discuss caching and its significance in DTN-based web browsing, and briefly introduce a few select security issues. We mostly limit our discussion to static websites. The problems that arise with dynamic web content are explored in a separate section in this chapter.

3.1 Overview of HTTP

HTTP is the protocol used in the Internet to transport web content reliably from web servers to clients. HTTP follows the request/response paradigm in which the client (typically a web browser, often referred to as the *user agent*) sends requests to the server and receives responses generated by the server. HTTP is most often used on top of TCP, although it could be deployed on top of any reliable transport protocol.

Resources

A web resource is a source of arbitrary web content. Resources can be static files on the web server's filesystem, such as HTML documents or images. Resources can also be web applications that generate content on demand based on some input parameters. [25]

Each web resource has a name – a uniform resource locator (URL) – that uniquely identifies the resource. URLs are a subset of URIs and follow the same syntactic rules. The difference between a URL and a URI is semantic: in addition to identifying a resource, a URL provides a means of locating the resource by describing its primary access mechanism (e.g., its network location) [12]. Most URLs follow a standardized format of three main parts: the scheme (*http:*), the server address (e.g., *www.netlab.tkk.fi*) and the resource location on the server (e.g., */tutkimus/projects.shtml*) [25].

In this thesis, we typically use the term resource to refer to static files. However, it should be kept in mind that the term also has a broader meaning.

Requests and Responses

HTTP transactions comprise requests sent from the client to the server, and responses sent from the server back to the client. The client is always the initiator of the transaction. To send a request, the client establishes a TCP connection to a particular port on the server host (port 80 by default). The server application listens to that port and waits for incoming requests. Upon receiving a request, the server processes it, generates an appropriate response and sends it back to the client.

Requests and responses may transfer an *entity*, which is HTTP parlance for the actual content that is being transferred, e.g., an HTML document.

Message Syntax

HTTP messages (requests and responses) contain three parts: a start line, a number of header lines, and a body. The start line and headers are lines of ASCII text, separated by CRLF end-of-line characters. The body is a chunk of text or binary data, and may also be empty. A blank line is used to separate the body from the headers. Thus, in the case of an empty body the message terminates with two CRLFs.

In request messages, the start line contains a method describing what operation the server should perform, a request URL on which to perform the method, and the

HTTP version that is used. All of these fields are separated by a single blank space. The following is an example of an HTTP request message.

```
GET /tutkimus/projects.shtml HTTP/1.1
Host: www.netlab.tkk.fi
User-Agent: Mozilla/5.0 (Macintosh;U;Intel Mac OS X) Gecko/20071025
Firefox/2.0.0.9
Accept: text/html;q=0.9,text/plain;q=0.8,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Response messages return the status of the requested operation back to the client along with any resulting data. The response start line contains the HTTP version, a numeric status code and a textual phrase corresponding to the code. The following is the response to the example request above, shown here without the message body.

```
HTTP/1.x 200 OK
Date: Fri, 07 Dec 2007 13:04:37 GMT
Server: Apache/1.3.33 Ben-SSL/1.55 (Unix) DAV/1.0.3 PHP/4.3.10
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1
```

There are four types of header fields in HTTP: general-headers, request-headers, response-headers and entity-headers. General-headers may appear in both requests and responses, but do not apply to the entity that is (possibly) being transferred. Request- and response-headers pass additional information about the respective message and its sender. Entity-headers contain metadata about the entity that is contained in the message. When transporting web resources in DTN bundles, the resources might not be inside HTTP messages, but nevertheless this metadata must be preserved. We discuss this issue further in section 3.3.

Although the HTTP specification states that a set of headers must always end in a blank line even if there is no message body, many implementations (historically) omit the final CRLF in the absence of a body. To interoperate with these noncompliant implementations, applications should accept messages that do not contain the final blank line.

Methods

There are a number of different request commands, called HTTP methods, which accompany each request and tell the server what action to perform. The most common method is GET, which asks the server to return the resource specified in the request.

HTTP methods may have the properties of being safe or idempotent. A safe method should not generate any side-effects on the server. In particular, GET and HEAD requests should only be used for retrieving resources – they should never modify data on

the server. In practice, however, GET requests are often used in violation of the safety requirement by using regular HTML links to perform operations with side-effects.

The property of idempotence means that any number of identical requests generate the same side-effects as a single request. In other words, ten idempotent requests have same effect as one request. All safe methods are inherently idempotent. Idempotent requests may be resent by the user agent without informing the user [26]. This is usually done if the server is unresponsive, and is a possible source of errors if safe methods are used for state-changing operations.

An HTTP response may be cacheable depending on the request method that was used. Responses to GET and HEAD requests can be, and typically are, cached. Responses to a POST request can only be cached if the response includes appropriate *Cache-Control* or *Expires* header fields. [26]

POST and PUT requests contain a message body because they provide data for the server to process. Requests with methods other than these do not contain a body. Table 3.1 shows the most common HTTP methods and their properties [25].

Table 3.1: Common HTTP methods

Method	Description	Safe	Idempotent	Cacheable
GET	Get a document from the server.	Yes	Yes	Yes
HEAD	Get the headers for a document from the server.	Yes	Yes	Yes
POST	Send data to the server for processing.	No	No	Maybe
PUT	Store the body of the request on the server.	No	Yes	No
DELETE	Remove a document from the server.	No	Yes	No
TRACE	Trace the request path to the server.	Yes	Yes	No

Status Codes

All HTTP responses contain a status code that tells the client what happened with the request. There are numerous possibilities what the outcome of a request may be. It may be successful, yielding a *200 OK* response, or it may be a redirection or an error.

Status codes are returned in the first line of a response and they also include a human-readable phrase describing the returned status. Only the numerical codes are used by applications; the phrases are included for convenience.

There are five different classes of status codes, as listed below in table 3.2. Only a few codes in each class are defined by the current HTTP specification. Future protocol versions are expected to define more codes. If an application receives a status code that it does not recognize, it should treat it as a general member of the class whose range it falls into. [25]

Table 3.2: HTTP status code classes

Range	Defined range	Category	Example
100–199	100–101	Informational	100 Continue
200–299	200–206	Successful	200 OK
300–399	300–305	Redirection	301 Moved permanently
400–499	400–415	Client error	404 Not found
500–599	500–505	Server error	503 Service unavailable

Protocol Versions

Currently there are two versions of HTTP that are in widespread use: HTTP/1.0 and HTTP/1.1. Despite HTTP/1.0 being extremely successful, it has numerous shortcomings that are addressed in HTTP/1.1. The key differences between the two protocol versions are discussed in detail in [27] and summarized below.

HTTP/1.0 provides a simple mechanism for caching with the *Expires* header and conditional requests. HTTP/1.1 retains this basic design, but augments it with new features and more careful specification of existing ones. The *Cache-Control* header allows for more fine-grained control over caching and mitigates problems related to clock skew which are caused by the absolute timestamps used in HTTP/1.0. *Entity tags* can be used to uniquely label resources in an unrestricted fashion which, along with new request headers, makes conditional requests more useful.

HTTP/1.1 introduces the concepts of *persistent connections* and *pipelining* of requests. In HTTP/1.0, the default operation is to make each request over a separate TCP connection, incurring the costly connection setup overhead to each request. Short-lived connections also fail to utilize the full potential of TCP, being stuck in the slow start phase most of the time. Many HTTP/1.0 implementations use the *Keep-Alive* header – which is not a part of the original standard – to create persistent connections. This unofficial, extended protocol is often referred to as HTTP/1.0+ [25]. In HTTP/1.1 persistent connections are always used by default. If an application does not want to use persistent connections, it must explicitly inform the peer that the connection will not be reused.

Pipelining means that a client need not wait for a response for one request before sending another request on the same connection. The client may send all pending requests at once, avoiding the need to wait for network round-trips.

HTTP/1.0 does not allow multiple host names to be bound to a single IP address, because the requests hold no notion of the destination host. In HTTP/1.1, all requests must contain the *Host* header which identifies the host to which the request is destined. Other improvements in HTTP/1.1 include, e.g., data fragmentation using chunked transfer-coding and refined content negotiation.

Some servers send responses with the protocol version HTTP/1.x. This simply means that the server supports both versions of the protocol and the response can be interpreted as being either one. Typically the response will contain redundant headers that convey identical (or similar) information with the syntax inherent to the different protocol versions, such as *Keep-Alive* and *Connection*.

3.2 Static and Dynamic Web Content

Web content can be roughly divided into two categories: static and dynamic. Static content can be characterized as being content that does not change often. When requesting a static resource, the response is always the same regardless of any input parameters, such as any data sent with a POST request, cookies or the time of the day. Typically static resources are files on a disk – HTML pages, Cascading Style Sheet (CSS) files, images – that only change when the website author manually modifies or replaces them. Static websites are easy to work with in the sense that web servers can serve files from the filesystem very fast with minimal processing and files can easily be cached.

In the early days of the Internet most websites were static. Today, however, the typical website is dynamic, which essentially means that the response to a given request varies based on some conditions. Often dynamic websites are database-backed and responses – and thus the content of the website as presented to a certain user – are dependent on the database state.

That said, there is an important distinction to make when defining what we consider to be a dynamic web page in the context of this thesis. Typically, a web page whose appearance or structure is modified with client-side scripting is considered dynamic. Javascript is often used to alter the document object model (DOM) or the CSS properties of a web page to create interactivity. From our point of view, however, these pages are not dynamic. The interactivity is confined within the browser and does not involve the server. The server just sends static Javascript files (or strings of Javascript code within an HTML document) to the client without any regards of how the content of the file will be used.

On the other hand, client-side scripting may employ techniques that allow the browser to communicate with the server asynchronously between page loads. These techniques that use the formerly proprietary, nowadays standard, XmlHttpRequest Javascript object are usually referred to with the term AJAX (Asynchronous Javascript and XML). Web pages with AJAX functionality may send data to the server "behind the scenes" continuously. The responses from the server may be used to alter the displayed page, creating a high level of interactivity. These pages are a special case that must be considered separately as they pose additional problems related to bundling. We return to discuss this subject further in section 3.6.

3.3 Bundling HTTP Messages

From the transport perspective, the problem that must be solved is how to replace TCP with the bundle protocol. The trivial solution is to perform one-to-one mapping of HTTP message into bundles. For request messages this works fine because sending out multiple requests at the same time is hardly ever needed.[1] For responses, however, this approach is not acceptable because it does not address the issue of unnecessary round-trips. As we have previously stated, we want to aggregate multiple resources into a single response bundle in order to reduce the chattiness of HTTP.

Figures 3.1a and 3.1b illustrate this concept. Figure 3.1a shows the interaction between a client and a server when the client requests a page with regular HTTP-style operation. The page consists of three objects: *index.html*, *styles.css* and *logo.png*. In order to retrieve all of these, the client has to perform three requests. First the client requests *index.html*. Then, after parsing the HTML and discovering that there are two more embedded objects, the client requests *styles.css* and *logo.png*.

Figure 3.1b shows the retrieval of the same page when resource bundling is used. The clients sends the first request for *index.html* like in the previous case. When the server receives the request, it collects all the other resources that depend on the requested one (the details of this are discussed in the following section), wraps them into a single bundle and sends it back. Thus, with resource bundling we are able to reduce the amount of HTTP transactions from three to just one. The number of resources found on real web pages is typically much more than three (based on our website analysis, the results which are presented in chapter 5), which means that resource bundling will have a significant impact on page retrieval time in high-delay environments.

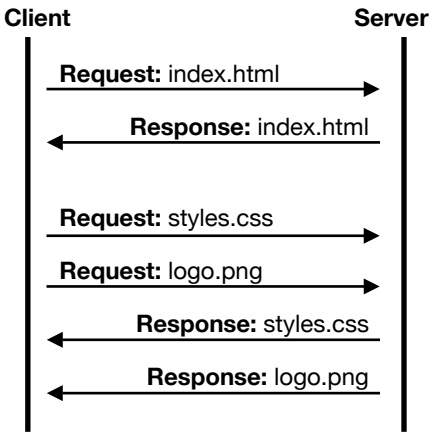


Figure 3.1a: HTTP-style page retrieval

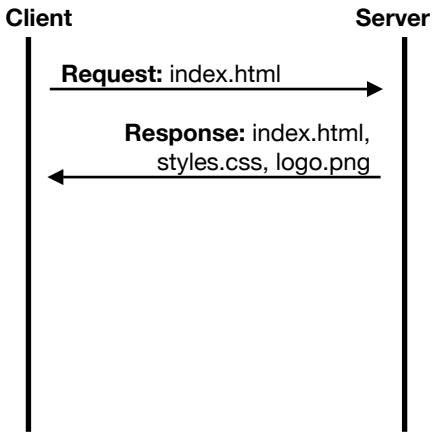


Figure 3.1b: Page retrieval with bundling

[1] Bundling multiple requests together would be beneficial in circumstances where HTTP pipelining is used, i.e., when the client has pending requests. However, the whole point of bundling *responses* is to avoid these situations: the client should not have the need to issue further requests after receiving a response bundle.

In order to carry multiple resources in a single response bundle, we must devise an aggregation format for the resources. We present a number of requirements for this format:

(1) It must provide a way to indicate the URIs of the contained resources.

(2) It should preserve the necessary HTTP metadata; most importantly the entity headers.

(3) It should be efficient, i.e., it should not introduce excessive overhead.

Requirement (1) is obvious: the client must know what resources it received in order to reconstruct the web page. (2) refers to the resource-specific HTTP headers that are sent by the server. An example is the *Expires* header which tells when the resource should be considered stale. Unless this header is preserved, a local cache might serve resources that are no longer fresh and should instead be retrieved from the origin server. Another example is the *Content-Location* header – without it the client does not know how to interpret the file and might, for example, display a text file using a wrong character encoding. Requirement (3) is also rather self-evident: the aggregation format should not negate the benefits we get from bundling by bloating the amount of bytes that need to be transmitted.

With these requirements in mind, we have come up with two alternative formats. The first one is to use the MIME/Multipart-based structure known as MIME Encapsulation of Aggregate HTML Documents (MHTML). This is the format Ott and Kutscher have proposed for this purpose in [1]. The other alternative is to simply pack a number of raw HTTP response messages inside a bundle, one after another. We implement MHTML aggregation in our web server application.

MHTML

MHTML is a MIME-based format originally designed for sending HTML documents over email. An MHTML message is a MIME/Multipart structure that contains a single root HTML document and all its embedded resources. Each part of the structure contains one resource and carries a *Content-Location* header which provides the URI of that resource, allowing the client to reconstruct the web page after unpacking the MIME structure. [28]

Having been designed for the exact purpose of carrying complete web pages in a single data structure, MHTML seems like a good fit for resource bundling. MHTML also allows us to preserve the HTTP entity headers relatively easily. Some of the headers map directly to MIME headers and the ones that do not can be included as extension headers (i.e., prefixed with "X-"). Thus, MHTML fulfills requirements (1) and (2).

When MHTML is used for its intended purpose – sending data over email – the content it carries becomes subject to the restrictions of SMTP. Specifically, the data octets must be within the US-ASCII octet range (i.e., 7-bit) and line length may not exceed 1000 octets [29]. This restriction means that all binary content must be encoded. In this case, "binary" means anything that is not 7-bit, which covers essentially all web content (HTML documents – which appear as "text" files – usually employ 8-

bit or multibyte encodings, making them binary in this sense). For files that are mostly composed of US-ASCII characters (e.g., ISO-8859-1 documents), the *quoted-printable* encoding scheme can be used. Files with arbitrary byte sequences (e.g., images) must be encoded with the *Base64* scheme. This encoding transforms the binary data into a text string that contains only US-ASCII characters, which results in about 33% increase in data volume [30]. This much overhead would severely limit the usefulness of MHTML because, as we will see in chapter 5, binary data makes up the majority of web content.

Fortunately, in our case the restrictions do not apply because we are not using SMTP transport. Binary content may be (optionally) tagged with *Content-Transfer-Encoding: binary* and included directly in the MHTML structure. Therefore, the overhead imposed by MHTML is insignificant and requirement (3) is also met.

Another point that should be taken into account is that HTTP responses carrying resources in MHTML format should have meaningful *Content-Type* header value. In our implementation, we use *message/rfc822* which is a generic content type that can be used with MIME messages [31]. It is conceivable that a more specific type, such as *message/mhtml,* could be used but from a practical point of view it makes no difference.

Raw HTTP

An alternative to MTHML is to use no structured aggregation format at all. Instead, the bundle can be filled with raw HTTP responses. The responses can be inserted into the bundle one after another so that first response is the one for the requested resource and the following ones for the embedded resources. The *Content-Length* header can be used to indicate resource boundaries so that the client can correctly extract the resources.

With this approach there is no need to map the entity headers into another data structure. The HTTP responses in the bundle obviously contain all the resource-related metadata since they are the very same responses that would be generated with a direct request. Resource URIs can be retained by using the HTTP *Content-Location* header (whose semantics differ somewhat from its MHTML namesake).[1] Since the overhead of this scheme is essentially zero, it seems that it satisfies our requirements.

However, there are both practical and conceptual problems with this solution. First, only one HTTP request hits the server but still it has to generate multiple responses. This implies that the server has to perform some kind of "internal" requests in order to create responses for the embedded resources. The details of how this could be implemented are not relevant, but nevertheless it might be wasteful in terms of using up server capacity.

Second, there is a conceptual problem: how can a single HTTP request result in multiple responses? Even if the client receiving the responses knows how to deal with

[1] The MHTML Content-Location is just a label – it does not necessarily provide a URI from which the corresponding resource can be retrieved, whereas its HTTP counterpart does just that.

them, and even if no standard is violated, the whole notion of generating an array of responses to a single request seems somehow flawed. Figures 3.2a and 3.2b below illustrate the two presented aggregation schemes.

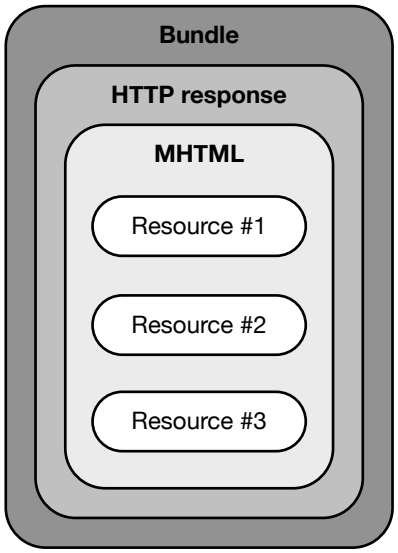


Figure 3.2a: MHTML bundling

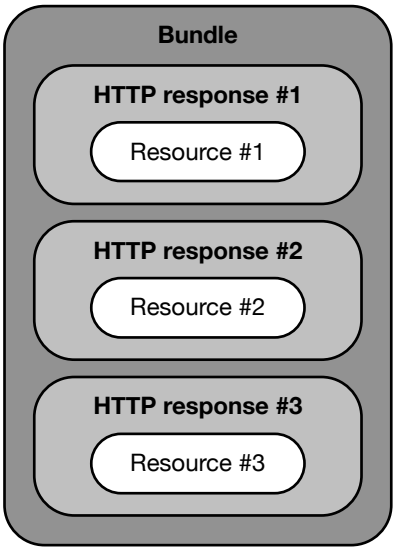


Figure 3.2b: Raw HTTP bundling

Bundle Addressing

One issue in using the bundle protocol to carry HTTP messages concerns addressing. Web resources are identified by URIs, such as *http://www.server.com/index.html*, in which the first part following the scheme name is the server address (*www.server.com*). When a client wants to retrieve this resource using regular HTTP-over-TCP, it performs a DNS lookup to find out the IP address of the server and then sends the request to port 80 (unless otherwise specified) on that address. The problem is in mapping these HTTP URIs to DTN endpoint identifiers, i.e., determining the EID to which a bundled request should be sent.

The bundle protocol does not place restrictions on the URI scheme name – any conformant scheme (e.g., *dtn:*, *http:* or *mailto:*) may be used in an EID. The scheme name merely identifies a set of rules that determine how the scheme-specific part should be interpreted [21]. The DTN2 reference implementation has adopted the convention of using the *dtn:* scheme for all EIDs (as it was proposed in earlier versions of the bundle protocol specification). Individual applications are identified by the SSP. However, it has been argued that application-specific URI schemes must be used in order to perform demultiplexing across different applications [32]. According to this perspective, applications that send and receive bundled HTTP messages should use the *http:* scheme in their EIDs.

Since our web server implementation uses DTN2, we must use the *dtn:* scheme for all EIDs. We do, however, support the idea of using application-specific URI schemes. Once the DTN community reaches consensus regarding this issue, we expect DTN2

and other bundle protocol implementations to adopt it. In any case, whatever URI scheme is used has minimal impact on our application.

Another issue concerning EIDs discussed in [32] is how much application-specific information an EID should include. There are two basic alternatives for this (consider a bundle that contains a request for *http://www.server.com/index.html*):

- The EID only identifies the application and provides no additional information about the resource. In this case, the destination EID for the bundle would be *dtn://www.server.com* or *http://www.server.com*, depending on whether application-specific scheme names are used.
- The EID includes additional information about the local resource – in the case of HTTP, the resource URI. Thus, the bundle destination EID would be *dtn://www.server.com/index.html* or *http://www.server.com/index.html*.

Our web server uses the former approach, i.e., the SSP of the EID is not used in any way when processing incoming bundles. We do not see any immediate benefits in using the EID to convey application context. What implications the different approaches might have on routing is beyond the scope of this thesis.

3.4 Identifying Resource Dependencies

In order to be able to bundle resources, the server must have a way of identifying which resources should go together. There are two basic ways of doing this – the information may be explicitly provided to the server in a *dependency file*, or the server may parse the source of an HTML document and infer the dependencies. These two approaches are not mutually exclusive and may be used to complement each other. We anticipate that parser based operation would commonly be used as a fallback mechanism when the dependency information is not available.

Explicit Dependency Information

The first way of providing dependency information to the server is to put the information into a file on the server. Whenever the server receives a request it will read this dependency file and see which other resources should be bundled along with the requested one. The information may be in a single file that describes the structure of the entire website or there might be one file per resource giving the dependencies related to that particular resource. The dependency file or files could be handwritten by the website author or might be created by some tool, e.g., a content management system. The format of the dependency file may be arbitrary but ideally it would be something that is easy for humans to edit in a text editor and easy for computers to parse.

The following is an example of a dependency file in YAML, which is a simple data serialization format [33]. It is also the format we have implemented.

```
*:
  - images/logo.png
  - css/styles.css

/index.html:
  - images/frontpage.png
  - scripts/widget.js

/information.html:
  - brochure1.pdf
  - brochure2.pdf
```

When the server gets a request for *index.html*, it checks the dependency file and finds that the files *images/frontpage.png* and *scripts/widget.js* should also be bundled. The entry with an asterisk denotes that those resources are common to all pages in the website and should always be included in the bundle. Thus, the server would return five files in the response – the one that was requested and four that were found from the dependency file.

Implicit Dependency Information

The other option for finding resource dependencies is to have the server infer them from the requested file. It is relatively easy to parse HTML documents and find the links to other resources, provided that the document is well-formed. For example, the following snippet contains a reference to an image file.

```
<img src="/images/logo.png" alt="Company Logo">
```

Upon encountering this in the HTML source, the parser would add */images/logo.png* to the list of resources to bundle. Binary files like images obviously cannot contain further references to other resources but CSS files can (e.g., background images). Thus, the CSS files linked to the requested resource must also be parsed. In fact, CSS files may refer to other CSS files using the *@import* statement, in which case the parser should work recursively through all of them. It is also possible that there are file references embedded in Javascript code but trying to find those is not feasible because the reference might exist in an arbitrary form within the code.

External Dependencies

Web pages sometimes contain references to resources that reside on a different server than the page itself. Web ads, for example, are typically located on the advertisers' servers and are just referenced from the page displaying them. Another example is high-traffic sites that use a cluster of servers to push static content in order to perform load-balancing. External dependencies pose problems to resource bundling because the server responsible for the requested web page does not possess the external resources and thus cannot include them directly into the bundle. We can think of a number of solutions for dealing with this problem. All these solutions, however, have their respective flaws.

The responsibility for retrieving the external resources may be given solely to the server receiving the initial request (server A in figure 3.3). After determining the ex-

ternal dependencies, server A contacts the remote server B and retrieves the resources. When the transaction is completed, server A creates a response bundle and sends it back to the client.

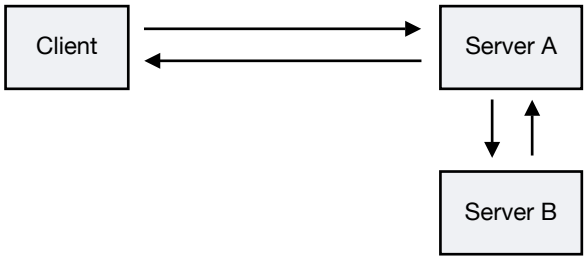


Figure 3.3: Server-based retrieval of external resources

The advantage of this solution is that it requires no extra coordination and is totally transparent to the client. The downside is that it introduces latency. If the connection between servers A and B is slow, the response bundle will be seriously delayed. It also puts more burden on server A, forcing it to maintain state during the three-way transaction. This has an adverse effect on scalability and security: a stateful server will tie up resources for longer periods of time, thus providing an easy target for DoS attacks. Moreover, the server cannot know whether the external resources are of any value to the client – they might just be ads or other irrelevant items that are not essential when displaying the web page. If the server had this information, it might make decisions on whether or not the potential delay is justified.

Another option is to share the responsibility between the servers so that A requests the resources from B, but B pushes them back to the client (figure 3.4). This solution requires specialized coordination between the nodes. The request from server A to B has to indicate that the response should not be returned to A, but should instead be sent to the client. Consequently, the client must be prepared to receive responses from multiple distinct servers. This way server A would be relieved of some of the burden as it would not have to maintain state. On the other hand, being able to command a server to send messages to any destination is a major security risk. Besides, the whole notion of asymmetrical transactions feels out of place with HTTP.

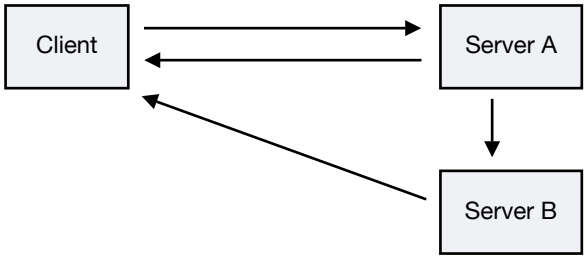


Figure 3.4: Asymmetrical retrieval of external resources

The third alternative is to do nothing and have the client retrieve the external resources. Upon receiving a request, server A bundles the resources that it itself has and omits the external ones. The client then requests the missing resources from server B after having parsed the HTML source. This is identical to regular HTTP-over-TCP operation.

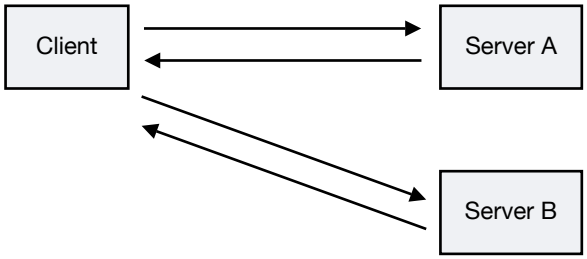


Figure 3.5: Client-based retrieval of external resources

In a way, doing nothing defeats the purpose of resource bundling since the client has to make one or more subsequent requests in order to receive all the resources of a single web page. If the external resources are not that important (e.g., advertisements), however, being able to display the part of the page that has already been received is clearly beneficial. On the other hand, if the resources are an essential part of the page, they should be delivered to the client as quickly as possible. Obviously, the quickest way to get the resources from server B to the client also depends on the proximity of the nodes (in terms of delay).

3.5 Bundling the Right Resources

In the previous sections we have implied that the server always returns (or attempts to return) one whole web page in a single response bundle. In some situations it may be beneficial to deviate from this and instead return either larger or smaller portions of a website in one bundle. For example, a moderately small website consisting of a number of linked pages could be bundled entirely. If the pages were mostly text, the resulting bundle would not be overly large. After receiving this bundle the client would have the entire website available for browsing regardless of connectivity disruptions thereafter.

Bundling Multiple Pages

Earlier we described the method of using a dependency file to allow the server decide what resources should go into a single bundle. With this approach, the website author can make arbitrary resource dependencies – there is no requirement to include just one page in a bundle. The dependency file may even be crafted so that the entire website is bundled when any one resource is requested. If parser based operation is used instead of dependency files, it is also possible to instruct the parser to follow links to a certain depth, thus creating larger bundles.

When would it be advantageous to extend the dependency hierarchy from one page to multiple pages? This is probably highly dependent on the website and the prevailing

network conditions, as the depth of the dependency hierarchy is a trade-off. Bundling more at once implies that additional server-side processing is required and that the bundle size is larger, but it also gives the client better disruption-tolerance. On the other hand, sending a single page per bundle gets the data to the client faster but prevents any offline browsing.

It is difficult to lay down any hard and fast rules on what is the correct way to craft bundles in any given circumstances. However, it seems clear that in a high-delay environment using up a few more server cycles to create a larger bundle is justified because the network delay overshadows the server processing delay. Conversely, if the network delay is low it may not be reasonable to waste any more time on the server than absolutely necessary.

Semantic Fragmentation

In some circumstances, it might be beneficial to do the opposite of what we previously described, i.e., instead of putting more resources into one bundle, it might make sense to split resources into multiple bundles. Some web pages are very large in terms of size – according to our analysis results, about 5% of popular web pages have more than 1 megabyte of content on them, typically in the form of images. Creating and delivering a single bundle from such a page might lead to a substantial delay. Assuming that the network latency is not very high, it might be a good idea to split this one large bundle into multiple smaller bundles. Furthermore, this splitting should be performed in a meaningful way so that the first bundle that is sent, which is also the bundle that the client will probably receive first, should contain the most important resources. We call this concept *semantic fragmentation*.

How exactly semantic fragmentation should be performed is beyond the scope of this thesis. However, it is obvious that the first fragment (i.e., bundle) must contain the requested HTML document. It should also contain the stylesheet files, if there are any, to allow the incomplete page render reasonably well. Beyond that it is up to the server to decide what resources it considers important.

We should also keep in mind what we are fundamentally trying to achieve: enhancing the end-user experience of web browsing. Web usability research has shown that incremental page rendering is perceived faster than rendering everything at once, even if it takes objectively longer [34]. This is a point in favor of semantic fragmentation. Even if the time between the initial request and the final response is slightly longer, the fact that the user is given constant feedback on the page rendering progress makes up for it.

There is, however, one rather serious problem with this kind of asynchronous bundle transmission where requests and responses do not match one-to-one. If the server performs semantic fragmentation and sends multiple bundles to the client, the client will not know how many bundles to expect. When it receives the first bundle and parses the HTML document, it will notice that there are resources that are missing. Normally, the right thing to do would be to send a new request for these missing resources, but in this case, the client should not do that because the resources are actu-

ally on their way. Thus, the first bundle should contain some kind of an indication that the response is split into several bundles.

Dynamically Adjusting Bundling Behavior

We have suggested that the optimal bundling strategy is dependent on the prevailing circumstances. Sometimes it might be beneficial to create very large bundles that contain whole websites, whereas other times it might be reasonable to split even a single page into smaller parts. To take this idea further, we conceive that there could be a mechanism that allowed the server to dynamically adjust the bundling strategy to achieve optimal performance in the given conditions.

For example, consider a web server that has a TCP interface for regular HTTP requests and a DTN interface for bundled requests. The server could observe usage patterns by examining the requests coming into the TCP interface (which we assume would receive much more traffic than the DTN interface). This data could then be used to predict how a user will navigate through the site. Then, when the server receives a request through DTN, it could look at the data, guess what page the user will want to see next and include that in the bundle. Other metrics, such as round-trip time measurements using bundle timestamps, could also be used to fine-tune bundling behavior. Again, the details of such mechanisms are beyond the scope of this thesis.

Avoiding Redundant Data Transfers

The server should always avoid sending resources that have already been sent in some earlier response bundle. Consider the following scenario: a user navigates to the front page of a website and receives a bundle that contains the HTML document for the page along with a number of other resources, many of which are common to all pages on the site (e.g., background images). The user clicks a link to move to another page on the site, causing the server to receive a new request for this page. As always, the server finds the dependencies for the requested document, but in this case many of them were already sent in the previous response. If the server does not take this into account, it will send the same common resources over and over again.

Normally, with regular HTTP-over-TCP, these common resources are not requested again from the server because they are held in the browser cache. When subsequent pages refer to these resources they are retrieved from the cache. In our case, the browser cache does not help because the server has to make the decision on what should be included in the bundle *before* the browser receives the response and gets to inspect its cache.

One solution to this problem would be to have the server remember which resources it has sent to each client, but as we have mentioned before, stateful web servers are a bad idea. Fortunately, there is another simple solution: the *Referer* [sic] header. This HTTP request-header allows the client to specify, for the server's benefit, the URI of the resource from which the requested URI was obtained (i.e., it tells the server where the client came from) [26]. Although it is not mandatory, most browsers seem to send this header with each request, unless including it is considered a security risk (e.g., when navigating to a non-secure page from a secure one).

Whenever the server receives a request it can check if it contains a *Referer* header that indicates that the client has already navigated on the site. The server can then compare the set of resources given by the referring URI to the ones given by the requested URI, and only send the ones that do not overlap. Since the *Referer* header is not mandatory and can easily be forged, the server cannot rely on it being available or correct. It can be used to enhance the operation of the server, whereas its absence will just cause the server to revert to blindly bundling all, possibly redundant, dependencies.

It would also be possible to use cookies to track the pages the user has visited. This would allow for more fine-grained control but would also add complexity to the server. Cookies can also be turned off from the browser, so this is not an airtight solution.

3.6 Problems with Dynamic Content

Dynamic web pages do not exist as files on the server – instead, they are created on the fly when a request hits the server. Typically, this involves a database from which data is pulled and used to create the page that goes into the response. The meaning of the term "dynamic web page" in itself is rather broad. In order to better grasp the subject, we can divide dynamic web pages into four categories:

(1) Pages that are created dynamically but not from on any volatile parameters. An example is a blog that holds the posts in a database. A request will cause data to be fetched and used to populate a template, but this process is not affected by any parameters.

(2) Pages that are created dynamically from some input parameters. An example is a search engine result page, whose content is entirely dependent on the search term supplied by the user.

(3) Pages that use Javascript in a way that has an impact on further requests issued by the browser. For example, Javascript might be used to insert a CSS link based on the result of browser detection.

(4) Pages that use Javascript to communicate with the server asynchronously between requests. An example is a search tool that shows "live" search results based on user input and refines the search as the user types in more letters.

Dynamic content that falls into category (1) is still suitable for bundling. If only a single page is bundled, there is no problem at all – the server performs the actions needed to create the response and bundles it. If more than one page is to be bundled, the server will have to perform multiple "response cycles" in order to create all the pages that will be bundled. This, however, is doable because all the information required to create the pages is available on the server.

Content in category (2) is more problematic. If we consider the example of a search engine, it is obvious that the initial bundle that contains the page with the search field cannot also contain the search result page because the server cannot predict what

search term the user will type in. Thus, dynamic pages that depend on input parameters are only suitable for single-page bundling.

Depending on the situation, web pages in category (3) may be more or less troublesome. Having file references within Javascript code is not too problematic in itself. After all, the dependency file can be used to label these files as belonging to the response bundle. However, the details of the Javascript code ultimately decide whether or not problems will occur. For example, if the purpose of the code is to insert a browser-specific CSS link, all of the possible CSS files would have to be included in the bundle, even though only one of them would be used. Furthermore, if the parser is used for finding resource dependencies, the references within Javascript code will be missed, resulting in additional requests from the browser.

Category (4) presents the worst problems. First, we should consider a bit of terminology because the content in this category can often be viewed as *web applications*. The terms *web page* and *website* – which we have already used extensively in this document – are probably well known to the layman. A web page is single document comprising multiple resources, displayed in a browser window. A website is a collection of linked web pages. The term web application refers to something more complex: an application that exposes its interface as web pages. Strictly speaking, a web application is nothing more than a regular collection of web pages. However, since the web pages that make up the web application are essentially a graphical user interface – not a means for displaying a simple piece of information – they usually offer a high level of interactivity. HTML links on a web application may not be simple links to another resource, but instead may be used to trigger actions in the application back end, often through the use of asynchronous Javascript techniques (AJAX).

It is this asynchronous communication between the web page and the server that does not work in a DTN environment. With bundling, our aim is to minimize the amount of round-trips between the client and the server. AJAX-based web applications are exactly the opposite in this regard: they rely on being able to constantly perform transactions with the server in order to create a highly interactive user interface. It is possible to perform one-to-one mapping of AJAX requests into bundles, but network delays and disruptions would probably render the application unusable.

Although some web applications may be inherently unsuitable for DTNs, some simpler websites may be made more fitting for bundling by following the principles of *graceful degradation* (or *progressive enhancement*). These terms refer to the idea that a web page should be usable even if a certain technology is not available. Thus, if a web page utilizes Javascript and a user agent does not support it, the page should degrade gracefully and still remain accessible, although possibly with slightly reduced functionality. Progressive enhancement means essentially the same thing, though the term asserts more firmly that complementary technologies should only be used to enhance the baseline user experience. In practice this means that if, for example, links are used for triggering AJAX actions, the same links should do something meaningful (with regular HTTP requests) for a user agent that has no Javascript support. By following these principles website authors can make their websites work better with bundling.

Some technologies have been developed to allow web applications to work in offline mode [35, 36]. These are, however, specialized solutions that require additional infrastructure software to work. Thus, they are not directly applicable to our purposes, and though they might contain ideas that are also relevant to what we are trying to achieve, we do not want to dig into this subject much further as it is not within the scope of this thesis.

3.7 Caching

Caching is an important concept in HTTP as it leads to a number of benefits. Caches reduce redundant data transfers and help clear out network bottlenecks. They also reduce load on origin servers and shorten distance-induced transmission delays. [25]

Caches can reside in many points along the communication path. Web servers that serve dynamic content often use caching to avoid recreating the requested page when it is not necessary. Different nodes within the network may employ their own caches. A gateway node on the perimeter of an intranet might serve cached resources to the internal network. Finally, web browsers have their own caches that can often be used to completely avoid sending a request to the network. We consider two forms of caching – in intermediary DTN nodes and on the web server – and their applicability in the context of this thesis.

Caching in Intermediaries

It has been suggested that a DTN could be used as a massive distributed storage, with each capable node acting as a cache. DTN nodes are expected to store bundles for longer periods of time and therefore must have a fairly large amount of storage capacity. This capacity could be exploited for caching by having the nodes store the bundles even longer than what is necessary for DTN delivery – for the lifetime of the bundle's application content. These caches could then respond to passing requests, provided that two conditions are met: the intermediate node is able to match the request to a cached resource and to determine whether the freshness requirements from the request are satisfied. [8]

The authors of [8] propose an additional bundle block, *Application-Hints*, to convey information that is required to satisfy the aforementioned conditions. The information would contain a resource identifier, an operation type, the lifetime of the contained resource and optional resource-specific parameters. How this idea could be specifically applied to HTTP-over-DTN is touched upon briefly in [8, 9] – here we present these ideas, augmented with our own suggestions.

HTTP resources are identified by URIs. Therefore, it is natural to use these as the identifier in the Application-Hints bundle block. Now, consider a bundle that contains a single web page of 50 resources: one HTML document and 49 other resources. An interesting question is do we label each resource separately, or do we give the bundle some kind of a common identifier. If we list the URIs of all the contained resources we end up with a bloated bundle block. However, if the resources are not labeled, the cache cannot reply to requests for these resources.

The operation type, carried in the *Application-Hints* block, should indicate the general class of operation of the message contained in the bundle. It should tell, independent of the application protocol, whether the bundle contains a request, an error message or a response with a set of resources. In HTTP, request and response messages can be distinguished by the start line of the message. Furthermore, the type of the request (the HTTP method, e.g., GET) or the response (the status code, e.g., *200 OK*) can be easily determined.

The most interesting bit of information with regard to caching is the resource lifetime. This refers to the application layer lifetime of a resource which, in the case of HTTP, is determined by the entity headers. The *Expires* and *Cache-Control* headers allow a cache to determine whether the resource can be considered fresh. The *ETag* and *Last-Modified* headers are used with conditional requests that to revalidate the freshness of a cached resource. Again, if we consider the bundle with 50 resources, we face a problem similar the one with URIs: do we include the lifetime information of all resources into the bundle extension block? If not, what do we put in there? What exactly determines the lifetime of a bundle that contains resources with differing lifetimes?

This controversy seems to boil down to this question: do we view a bundle of resources as a single entity or as a set of multiple entities? If we take the former stance, we lose some of the benefits of caching, namely, the ability the respond with resources that are contained in the bundle but not explicitly labeled. Nevertheless, we feel that the right thing to do is to expose the bundle as a single entity and use the URI and lifetime information of the HTML resource, i.e., the requested resource for which the bundle was originally created. This way the size of the bundle extension block does not grow out of hand and the nodes acting as caches do not have to work so hard – a passing request has to be matched against one URI instead of 50. That said, further research directed towards HTTP caching in DTNs is clearly needed – especially since HTTP arguably makes up for the most important traffic on the Internet.

Caching on the Server

With each incoming request, the DTN-enabled web server has to perform a great deal of work. If MHTML bundling is used, it has to extract the HTTP request from the bundle, parse it, gather all the relevant resources, wrap them into an MHTML structure and finally send them back in a response bundle. All this consumes valuable server cycles and has a negative impact on scalability. In reality, it would not be necessary to go through this whole process with every request. In fact, it is only necessary for the first request – after that the response bundle can be cached and all subsequent requests for the same resource can be satisfied with the cached bundle. As we are only working with static resources, the cached bundles will not become stale too quickly.

As discussed above, the bundles should carry the information that facilitates caching in an extension block, but clearly there are open issues as to how exactly this should be done. Therefore, we do not implement a caching mechanism based on the *Application-Hints* block, but instead take a more pragmatic approach to the issue (see section 4.5 for details).

3.8 Security Issues

We will not delve deep into the multitude of security issues that are related to our subject. The following will just briefly present a few interesting points from two different perspectives.

Application Data Security

On the Internet, HTTP security is achieved by using a security protocol, either Transport Layer Security (TLS) or its predecessor Secure Sockets Layer (SSL), above the TCP layer. TLS uses symmetric cryptography and secure hash functions to ensure privacy and data integrity [37]. The protocol operates by using a stateful end-to-end connection and therefore is unsuitable for HTTP-over-DTN. In DTN, we cannot rely on lower layer protocols for end-to-end security, and native bundle layer security mechanisms are still very much a work in progress [20]. Thus, we must use application layer mechanisms to achieve security.

Earlier we described how MHTML can be used as an aggregation format for web resources. Since MHTML is a MIME-based format, it is natural that we utilize the standard MIME security system: S/MIME. This provides us with the following cryptographic security services: authentication, message integrity and non-repudiation of origin (using digital signatures), and data confidentiality (using encryption) [38]. An S/MIME message in itself is a normal MIME-formatted structure that contains the original message (mutated by a cryptographic algorithm) along with security metadata (e.g., the digital signature). S/MIME is based on public key cryptography and so the usual complications associated with a public key infrastructure apply.

DTN Infrastructure Security

Considering the concepts presented in this thesis, we can see that a DTN-enabled web server has to do more work per-request than its plain HTTP counterpart. It has to gather multiple resources to a single response, possibly using some kind of an algorithm to determine optimal bundling behavior. It might even need to fetch resources from external servers. The fact that one simple, cheap request can trigger the server to perform a great deal of work makes it susceptible to DoS attacks.

This is related to a more general problem with DTNs. Since DTN nodes often have limited resources (bandwidth, computing power, battery capacity), all excess traffic is harmful. Therefore, a malicious party can attack a DTN by simply injecting loads of bundles designed to set off costly operations in the network nodes.

One way to counter this would be to authenticate all bundles. This, however, is in conflict with anonymous web access. Another solution would be to used trusted proxies which would authenticate the user than then send requests on their behalf, thus preserving the anonymity of the user. These proxies could operate with either HTTP-over-DTN or plain HTTP-over-TCP depending on the situation, and might also act as caches. [1]

3.9 Summary

In this chapter, we have presented the conceptual work of the thesis. The basic premise is that in DTNs, web resources should be transfered in bundles that contain aggregates of related resources instead of individual resources. This is done in order to reduce the number of round-trips between the client and the server.

Since bundles are used to carry multiple resources, there must be a suitable aggregation format for the resources. We defined a number of requirements for this format and suggested MHTML as a prime candidate for this purpose. Furthermore, we discussed how to implement mechanisms that allow the server to infer dependencies between resources, by either listing the dependencies explicitly or by having the server parse the source of an HTML document. With these concepts we have established a framework on which we can build our implementation of a DTN-enabled web server.

We also discussed the possibility of using adaptive bundling algorithms to perform semantic fragmentation and the problems related to bundling dynamic web content. Finally, we talked about caching and security, which are important concepts in relation to not only web servers but also DTNs in general.

Theoretical work, such as that presented in this chapter, has little value until it is realized in practice. In the following chapter, we take the concepts discussed here and use them as a basis for a concrete application.

4 Implementation

After laying down the conceptual groundwork in the previous chapter, we now turn our attention to the implementation of the DTN-enabled web server. Our objective is to produce a fully functional server application that can be used for small-scale deployments. The aim is not to create a commercial-grade application but rather a proof of concept verifying that the ideas presented in the previous chapter are reasonable and can be realized in practice. Therefore, we do not attempt to optimize the server for high performance and thus do not expect the server to perform very well under heavy load. Nevertheless, we do want to ensure that the performance stays on a reasonable level and does not render the server useless.

In developing the server, the main focus is on keeping the code base small and simple. The primary means to achieve this goal is to use an existing open source web server as the basis of the implementation and leverage as much as possible from the existing code. The focus on simplicity is carried out through the development process. If there are two ways to achieve a certain feature, one of them efficient but complex, the other slower but simple, we opt for the latter.

In this chapter, we first introduce technologies that form the basis of the implementation. We describe the application architecture and then discuss the major functional components of the server separately. The intention is to document the implementation with sufficient detail so that this text, the source code and its inline comments, and bit of object-oriented programming expertise will allow the reader to understand how the application works and how it could be further developed. Finally, we present the proxy application that was developed along with the server.

4.1 Background

The server uses a number of different open source technologies as its building blocks. The following introduces these, omitting the DTN2 reference implementation which we have already discussed in chapter 2.

Ruby

The web server is built in the Ruby programming language. Ruby is an interpreted, object-oriented language that has had a strong niche following since its inception in 1995, but is only now gaining more wide-spread popularity. The syntax of the language resembles Perl and the object-oriented features are similar to those in Smalltalk. There are a number of reasons for this language choice, including the following.

- Ruby is very high-level language compared to C. With Ruby we can write less code that does a lot more.
- The standard library is extensive and provides us with most of the infrastructure we need. For most everything else, third party libraries are readily available.
- Ruby is dynamic in the sense that all classes can be reopened and methods can be redefined. This makes extending existing applications convenient as we can easily overwrite just the relevant parts.
- There is a Ruby interface to the DTN2 library.

The biggest downside of using a high-level interpreted language is that it can never run as fast as compiled C code. Whether this performance penalty is acceptable or not depends on the application – for our needs Ruby is sufficiently fast. Another drawback is Ruby's threading model which is based on user-space threads (i.e., green threads). This causes problems when a multithreaded Ruby application interacts with a low-level program that uses blocking I/O (e.g., the DTN2 reference implementation).

DTN-Ruby

DTN-Ruby [39] is a set of Ruby bindings to the application library of DTN2. It provides an interface through which applications written in Ruby can utilize the functions of DTN2. Our application uses the DTN-Ruby bindings to send and receive bundles through DTN2. We have built an abstraction class on top of the direct bindings to make the interface more Ruby-like and to discourage C-style programming.

Mongrel

Mongrel [40] is an HTTP library and a web server written in Ruby. Instead of taking on the daunting task of building a web server from scratch, we use Mongrel as the basis of our implementation. Mongrel is based on a single-process, multithreaded architecture where a single main-thread accepts incoming connections and spawns new worker threads that handle the requests.

Mongrel uses a strict HTTP/1.1 parser that – unlike with many other web servers – is based on whitelisting instead of blacklisting. This means that the parser rejects everything except input that is strictly conformant to the HTTP/1.1 specification. From a security perspective this is a very smart thing to do.

Mongrel implements HTTP/1.1 with a few notable exceptions: pipelining and persistent connections. The author of Mongrel argues that these features were only potentially useful years ago when making connection requests was expensive over phone lines, and that with modern networking equipment they actually place more load on the HTTP server, unnecessarily exhausting concurrent request processors [41]. Omitting these features allows Mongrel to avoid extraneous complexity and also tackles a security issue that is related to Ruby's limited I/O capabilities (namely, the fact that Ruby can only keep 1024 open file descriptors at a time). Mongrel always sends a *Connection: close* header with each response, thus preventing persistent connections. This comes at the cost of having the extra round-trips associated with a TCP connection setup with each new request. In our case, however, this is not an issue because ideally page retrievals are handled with a single transaction and there is no need for subsequent requests.

Other External Libraries

The web server involves a few other external libraries. For generating MIME formatted messages we use the RubyMail library [42], which we have extended to provide direct support for MHTML. The server also includes a test suite developed using RSpec [43], which must be installed in order to run the tests.

4.2 Application Architecture

Our goal is to extend Mongrel so that it will be able to receive bundles, process the contained HTTP requests and send back bundled HTTP responses. The normal TCP-based operation will remain intact, and so the final server will have two interfaces: one for TCP (identified by an IP address and a port number) and one for bundles (identified by an EID).

As we said earlier, Mongrel's architecture is based on a single main-thread that listens to incoming connections and spawns new worker threads to handle requests. We extend this architecture by adding another listener-thread for bundles. Thus, the server can simultaneously receive requests from both interfaces. On the bundle interface, the HTTP request is first extracted from the bundle and then sent to the HTTP parser. Requests that come from the TCP interface yield normal HTTP responses that contain at most one resource. For requests from the bundle interface, the process involves an additional step, in which all the relevant resources are aggregated into the response. Figure 4.1 shows an overview of the server architecture.

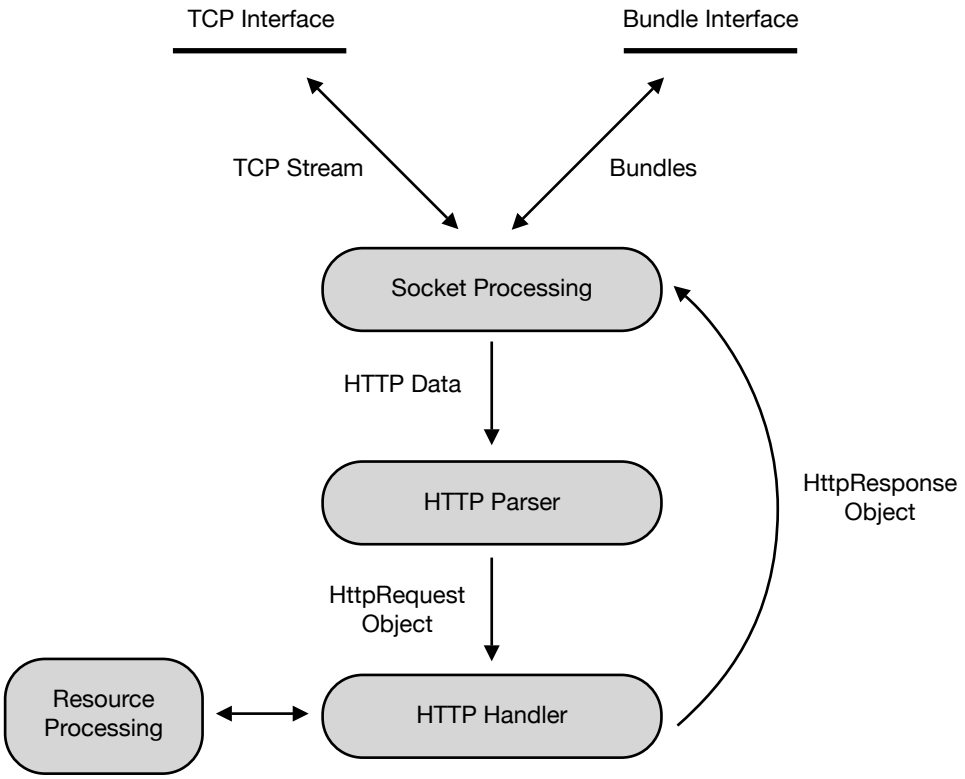


Figure 4.1: High-level overview of the server architecture

Extending Mongrel

Mongrel's architecture is composed of four main classes: *HttpServer*, *HttpRequest*, *HttpResponse* and *HttpHandler*. The *HttpServer* class contains the core functionality that processes incoming HTTP requests and coordinates the other classes to complete them. The *HttpRequest* and *HttpResponse* classes are abstractions of HTTP requests and responses, used internally in the processing. *HttpHandler* contains the logic that does the actual work in creating the response. Request processing instructions can be defined by subclassing the *HttpHandler* class (as described below). The basic Mongrel request cycle goes as follows [41]:

(1) *HttpServer* accepts a new socket connection and parses the HTTP headers. The request is rejected at this point if the headers cannot not be parsed.

(2) A new *HttpRequest* object is created based on the parsed headers. If the request contains a body, it is stored into the object.

(3) The *HttpRequest* and a newly created *HttpResponse* are passed to *HttpHandler* which does its processing and fills in the response headers and body.

(4) *HttpServer* streams the response back to the client.

Mongrel provides a very good extension mechanism for creating custom request processing instructions. Each incoming request goes through a *handler*, which is a simple function that performs some operations based on the request and creates a response.

Custom handlers can be written simply by creating a class that inherits Mongrel's *HttpHandler* and implements a single *process* method. This method receives the request as an *HttpRequest* object and writes the response into an *HttpResponse* object. Handlers can be chained so that a single request may pass through multiple handlers. Naturally, the order in which handlers are processed can be defined and handlers may also be applied only to certain request URIs.

For example, if we wanted to implement gzip HTTP compression, we could write a simple handler that performs the compression on the response body and adds the required *Content-Encoding* header. This handler could then be appended to the handler chain so that other handlers would be run first and the compression would be performed just before sending the response back to the client.

The handler mechanism allows us to implement the higher-level logic of the bundle-enabled server. The handler can take care of retrieving resources and formatting them into the response. For lower-level operations, i.e. sending and receiving bundles, this extension mechanism is not sufficient, and so we must rewrite some of Mongrel's internal methods in order to implement the bundle operations.

One of the principles we have followed during the application development is that the internals of Mongrel should be affected as little as possible. This means we do not want to end up with a new fork of Mongrel by adding the bundle functionality, but rather a bolt-on that is nearly independent of the internal implementation of Mongrel. This has a number of benefits: if future versions of Mongrel change some implementation details, our application is more likely to survive those changes. Also, since Mongrel is a tried and true piece of software, leveraging it as much as possible is more likely to produce a stable application.

DTN-Ruby Binding Abstractions

The DTN-Ruby bindings provide a Ruby interface to the DTN2 functions. Since DTN2 is written in C++, using the interface directly tends to produce C-style code. DTN-Ruby includes a Ruby class that builds an abstraction on top of the direct bindings in order to provide a Ruby-style interface. Based on this, we have taken the idea further and created set of classes that follow common Ruby conventions and expose a more native-feeling interface to DTN2.

For example, we use the standard object-oriented programming practice of raising exceptions to denote error conditions instead of relying on return values (e.g., 0 for success and -1 for error). We have taken advantage of Ruby's more sophisticated features, such as code blocks, and also tried to optimize for the common case – thus, sending and receiving bundles are very simple operations from the point of view of the application using the interface.

Class Structure

The server consists of four main modules – *Mongrel*, *Dtn*, *Mhtml* and *Blunder* [1] – each of which contains a number of classes. The modules are used to group the application into logical substructures. The modular architecture also facilitates reusability – for example, the DTN functionality could be used independently in another project because the module is not tightly coupled to the rest of the implementation. The most important classes and their main functions are presented in tables 4.1–4.4.

The *Mongrel* module contains fairly small amount of our code, only the extensions and modifications that are necessary to implement the bundle interface. The main class that runs the server, *Mongrel::BundleHttpServer*, is subclassed from the standard *Mongrel::HttpServer*.[2] It overwrites only the parts that it has to and otherwise keeps the original implementation intact, thus following our development principles.

The DTN functionality is implemented in the *Dtn* module. The classes in this module abstract the direct DTN-Ruby API to an easy-to-use, object-oriented Ruby interface. The following code snippet is a simple contrived example that shows how the interface can be used to send bundles.

```
client = Dtn::Client.new("dtn://source.dtn")          # Create a client
bundle = Dtn::Bundle.new("dtn://destination.dtn")     # Create a bundle
bundle.payload.data = "Test payload content"          # Insert payload
client.send_bundle(bundle)                            # Send the bundle
```

An important detail that helps us avoid making changes to Mongrel's internal socket processing is the *Dtn::DtnSocket* class. This class builds an abstraction on top of a *Dtn::Bundle* object so that it looks and acts like a regular socket – the exposed interface is identical but internally it just uses buffers for I/O. This allows Mongrel to process bundles in exactly the same way that it does with normal sockets.

The *Blunder* and *Mhtml* modules contain the high-level logic of the application. The methods in the *Blunder::Resource* class implement the dependency file and parser-based resource gathering operations. This class is never instantiated into objects; it only contains class methods that are used directly. The *Mhtml::Message* class extends the standard *Rmail::Message* by giving it MHTML-specific functionality and some convenience methods. Also, the binary encoding methods are implemented in the *Mhtml* module because, for some odd reason, Rmail only implements decoding.

---

[1] The code name of the application (just because it almost rhymes with "bundle").

[2] Ruby notation: *Module::Class*

Table 4.1: Classes in module Mongrel

Class	Description
BundleHttpServer	Contains low-level methods that overwrite their counterparts in plain Mongrel. Creates the bundle interface and handles starting and stopping the server.

Table 4.2: Classes in module Dtn

Class	Description
Client	An abstraction of a DTN client, i.e., an entity with an EID. Contains methods for registering the application to the DTN router, and sending and receiving bundles.
Bundle	Represents a bundle. Contains the bundle metadata (source and destination EIDs, priority, time-to-live value and other options) and the payload.
MemPayload	Represents a bundle payload that is kept in memory. Has methods for setting and retrieving the payload data. There are also two other classes, *FilePayload* and *TempfilePayload*, which are similar except that the payload data is kept in persistent or temporary files instead of memory.
DtnSocket	Gives a socket-like interface to a bundle. With this abstraction, Mongrel can process bundles just like they were regular TCP sockets. Has buffers which are used for read and write operations.

Table 4.3: Classes in module Blunder

Class	Description
BundleHandler	The HTTP handler for the server. Contains the *process* method that receives an *HttpRequest* object, performs the operations needed to fulfill the request, and returns an *HttpResponse* object.
Resource	Contains methods that deal with gathering resources on the server. Implements dependency file based resource fetching, as well as parser based operation.
Cache	Implements a simple caching mechanism for outgoing responses. Stores the responses and associated metadata on disk.

Table 4.4: Classes in module Mhtml

Class	Description
Message	Represents an MHTML message. Has methods for creating and parsing MHTML formatted structures.

Request Cycle

The following explains the steps that the server goes through when handling a request from the DTN interface.

(1) The listener-thread periodically calls the bundle reception method.[1] Depending on whether a bundle was received, it either passes execution to another thread or spawns a new one and calls *BundleHttpServer#process_bundle*.[2]

(2) A new *DtnSocket* object is instantiated from the bundle and passed to Mongrel's default *process_client* method.

(3) The *DtnSocket* object is processed like a normal socket: Mongrel reads chunks of data from it, passing them to the HTTP parser. If the request is valid, the parser creates a new *HttpRequest* object and passes it to the *process* method in the *BundleHandler* class.

(4) *BundleHandler#process* performs validation of the request URI. First, it checks that the URI is legal (i.e., not outside the document root) and that the resource exists. If the URI is illegal, the processing ends and no response is returned. If the resource does not exist, a *404 Not Found* error is sent. If the request URI is a directory, the check is performed against an index file in that directory (*index.html* and *index.htm* by default).

(5) Cache check is performed; if a cached response is found and it has not expired, processing continues at step (9).

(6) The *Resource.find* method is called with the request URI.[3] If the dependency file is present, it is used to determine the dependencies. Otherwise, the parser is used (but only if the requested resource is an HTML file). This produces a list of resources that should be included in the response bundle.

(7) The *process* method checks if the HTTP request contains a *Referer* header. If it does, *Resource.find* is called on the referring URI. This yields the resources that were sent in the previous response. These are removed from the set of resources obtained in step (6), so that no duplicates are sent.

(8) The list of resources is passed to the *Resource.create_mhtml* method, which loops through the list, sets the correct MIME headers and returns the final MHTML structure.

(9) The MHTML string is written to an *HttpResponse* object, which in turn streams the raw HTTP response to the output buffer of *DtnSocket*.

(10) Finally, *DtnSocket* wraps the HTTP response into a bundle and sends it back to the client.

---

[1] Due to Ruby's green threads, we must call the bundle reception method with a timeout value (1 ms). The underlying DTN2 C-function is blocking – if we did not use timeout and just let it block until it receives a bundle, it would halt *all* Ruby threads (including any currently running worker threads).

[2] Ruby notation: *Class#instance_method*

[3] Ruby notation: *Class.class_method*

4.3 Aggregation Format

The server implements MHTML-based resource aggregation as described in section 3.3. The MHTML module is based on the RubyMail library, which includes tools for working with MIME messages. Since MTHML is a MIME-based format, we can use RubyMail instead of writing our own MIME implementation, although we have made some extensions to the library to make handling MHTML a bit more convenient. The following shows an example HTTP response that contains an MHTML formatted web page comprising one HTML document and one CSS file. This what a response bundle leaving the web server might contain in its payload.

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 26 Feb 2008 10:41:38 GMT
Content-Type: message/rfc822
Content-Length: 2693

Mime-Version: 1.0
Content-Location: http://blunder.dtn
Content-Type: multipart/related; boundary="=-1204022498-1401-9568-2-="


--=-1204022498-1401-9568-2-=
Content-Location: index.html
Content-Type: text/html
X-Last-Modified: Tue, 26 Feb 2008 10:38:36 GMT
X-Cache-Control: max-age=2592000
X-Expires: Thu, 27 Mar 2008 10:41:38 GMT
X-ETag: "47c3ec2c-59c-28b6f0"

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">

<html lang="en">
<head>
  <title>A Simple Web Page</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <link rel="stylesheet" type="text/css" href="css/styles.css">
</head>
<body>
  <h1>Simple!</h1>
</body>
</html>
--=-1204022498-1401-9568-2-=
Content-Location: css/styles.css
Content-Type: text/css
X-Last-Modified: Tue, 23 Oct 2007 08:39:09 GMT
X-Cache-Control: max-age=2592000
X-Expires: Thu, 27 Mar 2008 10:41:38 GMT
X-ETag: "471db32d-25a-28b6f3"

body { margin: 20px; background: #333; }
h1 { font: 36px helvetica, sans-serif; color: maroon; }
--=-1204022498-1401-9568-2-=--
```

The outermost part of the multipart structure has a *Content-Location* header that specifies the base location of the web page (*http://blunder.dtn* in the example). All the other parts also have *Content-Location* headers which are relative to this base location (*index.html* and *css/styles.css*). These headers enable the client to identify the different resources and reconstruct the page.

The *Content-Type* header is identical to its HTTP counterpart, providing the Internet media type of the resource. The example also shows that each part carries a number of extension headers (prefixed with "X-"). These are the HTTP entity headers that carry additional resource metadata. *Cache-Control* and *Expires* are used by caches to determine if the resource is still fresh. *Last-Modified* and *ETag* can be used with conditional requests to fetch a resource only if it has changed. Determining meaningful values for resource expiration is up to the server – our implementation does not attempt to do anything fancy, but instead just uses fixed values for *Cache-Control* and *Expires*. The value for *Last-Modified* is obviously the point in time at which the file was last changed and *ETag* is a unique identifier formed by a combination of the file's modification time, size and the filesystem inode number.

By default, binary content is not encoded by the server; it is included in the MHTML as such (without a *Content-Transfer-Encoding: binary* header as it is not necessary). However, the implementation also offers the possibility to use *quoted-printable* and *Base64* encodings on binary resources. Encoding must be used if, for whatever reason, an SMTP compliant message is required. In this case, anything that is not 7-bit (i.e., has data octets outside the US-ASCII range) is considered binary, which covers practically everything. If encoding is activated, files with media type *text/** are encoded using *quoted-printable* and everything else using *Base64*. The media type is determined solely by the file extension. This kind of encoding scheme is very rudimentary and should be used with caution, although with the most common cases it does work correctly. The only way to reliably determine whether a file is 7-bit is to scan the entire file for out-of-range octets. We chose not to implement this because the operation might be too costly to be used in practice, and moreover, we cannot think of that many situations where content encoding would be necessary.

4.4 Finding Dependencies

In section 3.4 we presented two different ways in which the server can determine resource dependencies. The first option is to provide the dependency information explicitly in a file; the other one is to infer the dependencies directly from the requested HTML document by parsing the source.

Dependency File

Our implementation supports dependency files in a YAML-based text format. YAML is a data serialization format that is designed to be easily mappable to common data types in programming languages, e.g., arrays and hashes. Using YAML for our purposes is conceptually simple because the data representation in the text file translates directly into data structures used in the implementation. The format is also relatively

easy to understand and modify even for a less savvy website author. Furthermore, Ruby has comprehensive YAML support in the standard library. The following is an example of a dependency file.

```
*:
  - images/logo.png
  - css/styles.css

/:
  - also: /index.html

/index.html:
  - images/frontpage.png
  - scripts/widget.js

/information.html:
  - brochure1.pdf
  - brochure2.pdf
  - info/*
```

The syntax of the dependency file is rather self-evident: there is a number of request URIs (e.g., *index.html*) and for each of these a list of local resources. Thus, when a certain URI is requested, the resources listed under that URI are included in the bundle. There are two special bits of syntax: the asterisk and the *also* keyword. The asterisk can be used in two ways. When it is used in place of the request URI (as in the first entry in the example) it simply denotes that the listed resources should be included in all bundles regardless of the URI. Web pages often have some resources that are common to all pages; this syntax avoids having to repeat them for each entry. Additionally, the asterisk can be used as a wildcard character to match multiple files. The bottom row of the example means that all files within the *info* directory (and its subdirectories) should be added to the bundle.

The *also* keyword can be used to express that additional resources, from another entry in the dependency file, should also be included. In the example, a request for the root of the website (i.e., */*) would return the same resources as a request for *index.html*. This special syntax is implemented for the sake of convenience; it also avoids unnecessary repetition in the dependency file. One important detail related to the *also* syntax is that it can be used to construct circular references (i.e., two entries that have their *also* keywords pointing to each other), inadvertently or otherwise. This fact must be taken into account in the implementation – failure to break these circular references will lead to an instant server crash.

If the dependency file does not have an entry for a given request URI, the server may just return that one resource (and possibly some common resources denoted by the asterisk), or it might use the parser on the requested file, as described below.

Parser

The parser can be used in place of, or in addition to the dependency file. It's advantage is that the manual labor associated with creating the dependency file is eliminated. However, using the parser consumes more server cycles and thus is slower. Be-

sides, the parser is not perfect: it might miss some relevant resources and may also include unnecessary ones. For these reasons, the using the dependency file as the primary mechanism for finding dependencies is preferred.

The parser uses relatively simple regular expressions to perform its work, and consequently, is guaranteed to work properly only on well-formed XHTML documents. The most important constraint is that attribute values must be quoted. In HTML, quotes can be omitted for values comprising certain characters, although this is considered bad practice [44]. We reckon that a simple, strict parser is sufficient for our purposes, although in a real production environment the parser should be able to deal with malformed HTML documents which unfortunately are all but rare. Hpricot [45] is an excellent HTML parser written in Ruby that could be used for this.

The parser searches for *img*, *link* and *script* elements in the HTML source and picks up the values from the relevant attributes. Additionally, *@import* statements that are used within *style* elements to link to external CSS files are identified.

The parser does not attempt to find frame dependencies (i.e., *frame* elements in the HTML source). This would require support for recursive parsing because documents within frames may contain further frames ad infinitum. Even though frames are still sometimes used, they are being phased out from the HTML standard due to their adverse effect on usability and accessibility [46].

CSS files are also parsed for image references (e.g., *url("/logo.png")*) and *@import* statements. The fact that CSS files can refer to other CSS files necessitates recursive parsing, which in turn makes it possible to create infinite loops with circular references. We return to discuss the parser operation further in section 5.3.

Referrer Checking

In order to avoid resending resources that have already been sent in an earlier response bundle, the server performs referrer checking. The idea is to use the referring URI to find the resources that were sent in the previous response and compare these to the set of resources that the current request URI yields. The overlapping part of these two sets of resources need not be included in the response.

This functionality is achieved by using the *Referer* HTTP request header, the value of which is the referring URI. As we have stated before, this header is not mandatory and therefore might not be present in a request. In this case, referrer checking is not performed.

The obvious downside to using this method is that it puts more load on the server. It essentially means that the server has to perform the resource searching operation twice per request. However, we suspect this is a smaller price to pay than sending the same resources over and over again.

4.5 Caching

The server implements a simple file-based caching mechanism. Creating a response bundle is an expensive operation: the server has to parse the dependency file (or worse, the source of an HTML document), read the files from disk and build the MHTML structure. By caching the responses, we can avoid going through this process with each incoming request. With proper mechanisms in place to determine the freshness of cached resources, caching should have a significant positive impact on server performance.

When the server receives a request for the first time, it creates and sends the response as usual, but it also saves the created MHTML structure (i.e., the bundle payload) into a file on disk. Along with the MHTML is also saved a piece of metadata: the *Expires* and *ETag* headers of the requested resource. When a subsequent request comes in, the server determines the freshness of the cached response by comparing the header values in the metadata to the current values (i.e., the current time and a recalculated entity tag). If the expiration time has not passed and the entity tags match, the cached response can be served.

It should be noted that this caching mechanism only validates the freshness of the requested resource – i.e., the resource for which the response was created – and not the other associated resources that the MHTML structure contains. If, for example, a CSS file that is a part of a cached response is modified, this change will not invalidate the response. The *Expires* header is included in the metadata for this purpose: by setting a reasonable value for it, the cached responses will be periodically invalidated, thus allowing the server to rebuild the cache with fresh content. Furthermore, cache freshness will hardly become a problem because the server is designed to serve static websites that do not change often, and when they do, the changes are done by the website author who can also manually empty the cache (by restarting the server).

It would also be possible to cache entire bundles instead of just bundle payloads. This would make things even more efficient as it would avoid having to create a new bundle for each cached response. However, the decision to cache bundle payloads is purely pragmatic because this way we can implement the entire caching mechanism in the HTTP handler. Caching entire bundles would require implementing functionality outside the handler in the lower-level processing logic, thus creating unnecessary complexity.

The implemented caching mechanism is a simple one with limited functionality. One of its drawbacks is that bundle contents are opaque to the bundle layer. Therefore, bundles cannot be easily cached in intermediary nodes without application-level logic. Nevertheless, as we have mentioned, caching in DTNs is an active research topic. When standard conventions regarding the subject are established by the DTN community, we expect them to offer potential for more efficient caching.

4.6 Proxy

The scope of the practical part of this thesis is limited to developing the server application. However, a server alone is not very useful – a suitable client application is needed, if only for testing purposes. There are currently no web browsers with native DTN support. Instead of developing one, an easier approach is to create a proxy that wraps the outgoing HTTP traffic into bundles. This way any regular web browser can be used for testing the server.

We have implemented a simple proxy application that sends and receives bundles on behalf of the browser. The proxy has a local cache – similar to the one on the server – that stores the resources extracted from response bundles. The browser communicates with the proxy in regular HTTP-over-TCP fashion, i.e., retrieving resources one-by-one from the proxy cache. The following steps, along with the conceptual image in figure 4.2, illustrate how DTN-proxy-based web access works.

(1) The web browser sends an HTTP request and starts waiting for the response.

(2) The proxy intercepts the request and puts it into a bundle. The destination EID is obtained from the *Host* request header simply by taking the value and replacing the *http:* URI scheme with *dtn:*.

(3) The bundle is sent to the server through the DTN. The server processes the request and sends back the response.

(4) The proxy receives the response bundle, extracts the resources and stores them to the cache. The originally requested resource is streamed back to the browser.

(5) The browser parses the HTML source of the response and sends requests for the embedded resources.

(6) The embedded resources can now be served from the proxy cache.

The prerequisite for the final step – serving the embedded resources from the proxy cache – is that the server has been able to correctly identify all resources that belong to the page. If this is not the case, the proxy will have to issue further requests to the server. The proxy can be configured with a timeout period after which it returns a *504 Gateway Timeout* error if no response bundle has been received.

The HTTP interface of the proxy is also based on Mongrel. Apart from the cache, which a standalone module also used in the server, the whole processing logic of the proxy is implemented in a simple HTTP handler.

The proxy is an example of infrastructure software that will be needed if the DTN architecture becomes more widely deployed. This infrastructure will allow clients and servers, some of which are DTN-enabled and others which are not, to coexist and communicate with each other.

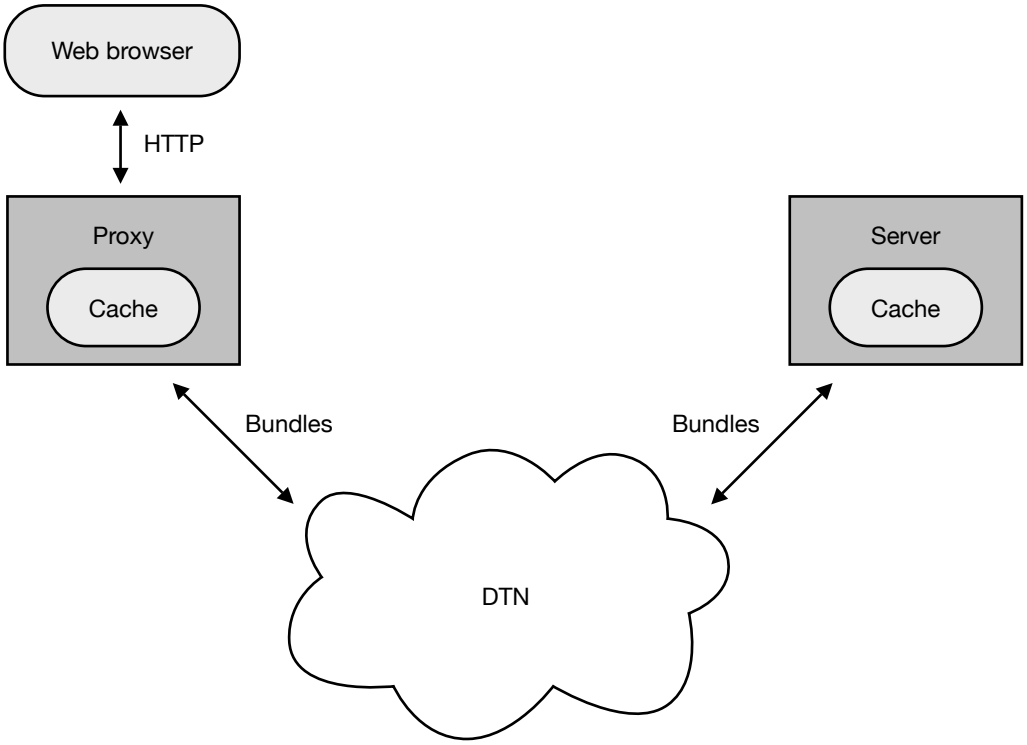


Figure 4.2: Proxy-based DTN web access

4.7 Summary

In this chapter, we presented our web server implementation which puts the concepts presented in chapter 3 to practice. Thus, the completed application serves as a verification of the ideas we have discussed in this thesis. The implementation is largely based on two supporting pieces of software: the DTN2 bundle protocol implementation and the Mongrel web server. Both of these are under development, and so we can expect that our application will also have to evolve in the future. We believe that the architecture of our implementation will allow this to happen without too much friction.

It is worth noting that currently the implementation only works on Unix based platforms. Mongrel does support Windows, but DTN2 does not – therefore, our server cannot be run on Windows. This, however, is a relatively modest shortcoming since the server side is mostly dominated by Unix variants.

An important part of implementing something is evaluation: how well it works, does it serve the purpose it was created for, and what are its potential limitations? This will be the theme of the next chapter.

5 Measurements and Analysis

After having implemented the web server, we are naturally interested in how it performs in a real-life scenario. In order to find out, we carry out a set of basic measurements in which we consider the server performance from various viewpoints. The aim of these measurements is not to produce extremely accurate and dependable data but rather to get a rough understanding of the level of performance and reveal any glaring mistakes. The HTML/CSS parser that the server uses to infer resources dependencies is a very important part of the server and therefore we devote a separate section to it.

This chapter begins with a brief digression from the main objectives of the thesis. We conduct an analysis of real-world websites to gain an understanding of the landscape in which web servers actually work. Then, we describe the details of our server measurements and present the results. Lastly, we discuss the parser operation and its implications to the server performance.

5.1 Website Analysis

In chapter 3, we discussed how web resources can be aggregated into bundles to avoid unnecessary round-trips in DTN environments. We justified the need for bundling by asserting that web pages typically consist of multiple resources. While this assumption is correct, we lack deeper knowledge of how typical websites are actually structured: how many resources usually make up a single page and what kind of resources are these? We conducted a small side study to find out answers to these questions.

Methodology

The analysis was conducted by gathering data from 500 websites, taken from a daily updated list of the most popular websites worldwide, the Global Top 500 by Alexa Internet [47]. The sites are diverse in terms of content, language and geographic loca-

tion. The data was gathered by creating a script that invoked a web browser to sequentially visit the 500 websites. The browser was configured to use a proxy (Charles [48]) that recorded all network traffic between the browser and the target servers. The raw data collected by the proxy was then parsed to a suitable format to produce the following results.

Upon processing the results we noticed that a large number (precisely, 61) of the websites on the list were different localizations of the Google search engine page. All of these pages are essentially identical, save for the language, and thus can be considered a single page. Therefore, we excluded all but one of the Google pages from the results and in doing so decreased the sample size from 500 websites to 440.

Results

Figures 5.1 and 5.2 show cumulative graphs of the web page sizes. Figure 5.1 depicts the size of the web page in terms of bytes, i.e., the total size of all resources combined. Figure 5.2 is similar except that it shows the number of files comprising the web page instead of the byte size. Notice the logarithmic x-axis in both graphs.

The shapes of the two curves are similar because, obviously, the number of files on a page and its size have a strong correlation: a large number of files is likely to result in a large total byte size. The general shape of the curve shows that extreme cases are rare: the bulk of web pages are similar in terms of the measured quantities, as indicated by the rapidly ascending middle section of the curve.

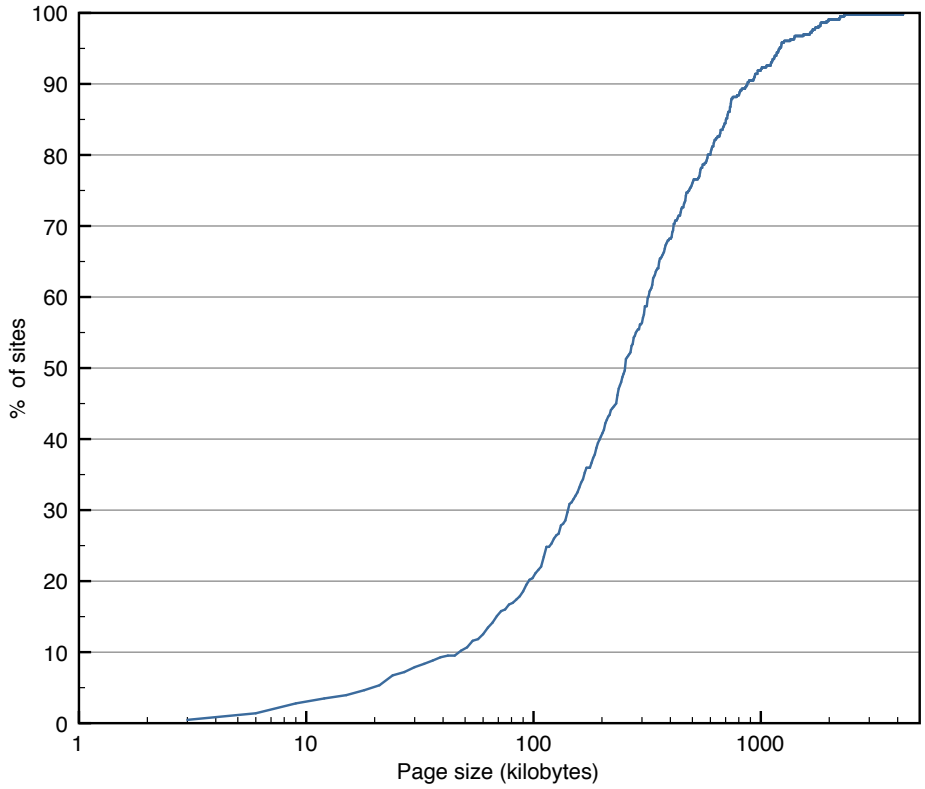
Table 5.1 below shows the percentiles from the 10th to the 100th for the graphs. For example, we can see that 50% of the websites included in the analysis are 253 kilobytes or less in size. In other words, the median size is 253 kilobytes. Correspondingly, the median of the file count is 53.

Table 5.1: 10–100th percentiles for the web page size in kilobytes the number of files

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
KB	48	96	142	196	253	319	413	585	872	4227
Files	10	21	32	42	53	64	82	104	137	399

This table also confirms the fact that extreme cases are rare. The largest web page is over 4 megabytes in 399 files, while the 90th percentiles drop to 872 kilobytes and 137 files. It is perhaps a bit surprising that there are so few small web pages – the 10th percentile of file count is as large as 10. This is doubtlessly due to the fact that the list only contains very popular, high volume sites. We suspect that personal websites and such often contain fewer resources but these are not represented in the sample.

Figures 5.3 and 5.4 show the relative abundance of different file types; the former one in terms of bytes and the latter in the number of files. The file type is determined from the *Content-Type* header sent by the server.



Figures 5.1: Cumulative page size graph in terms of bytes

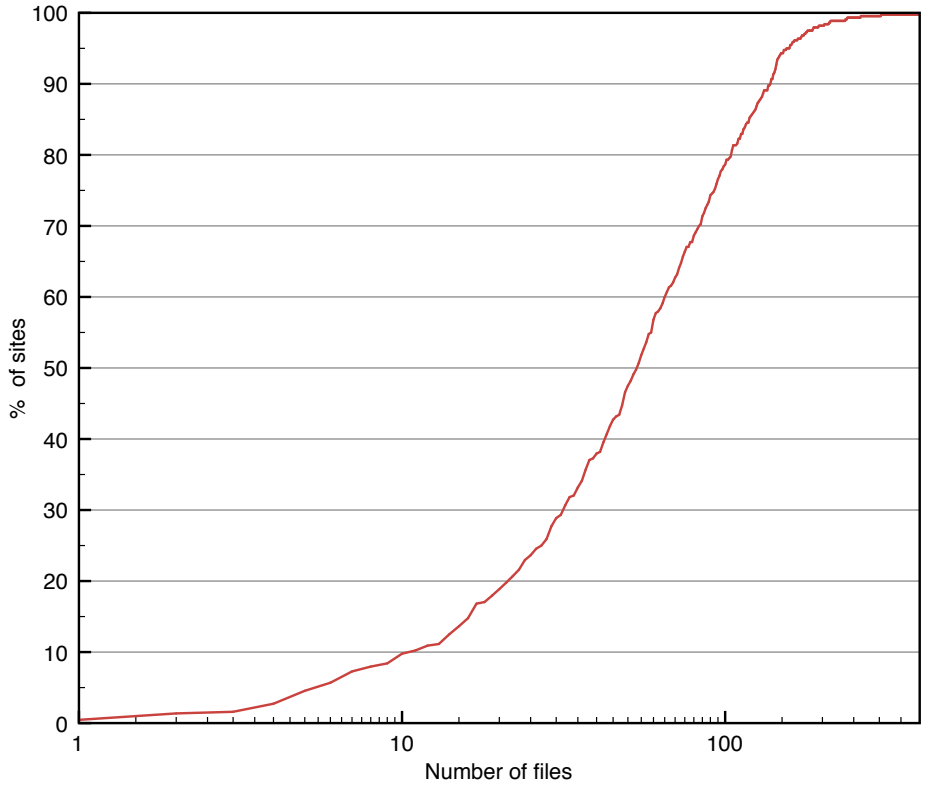


Figure 5.2: Cumulative page size in terms of the number of files

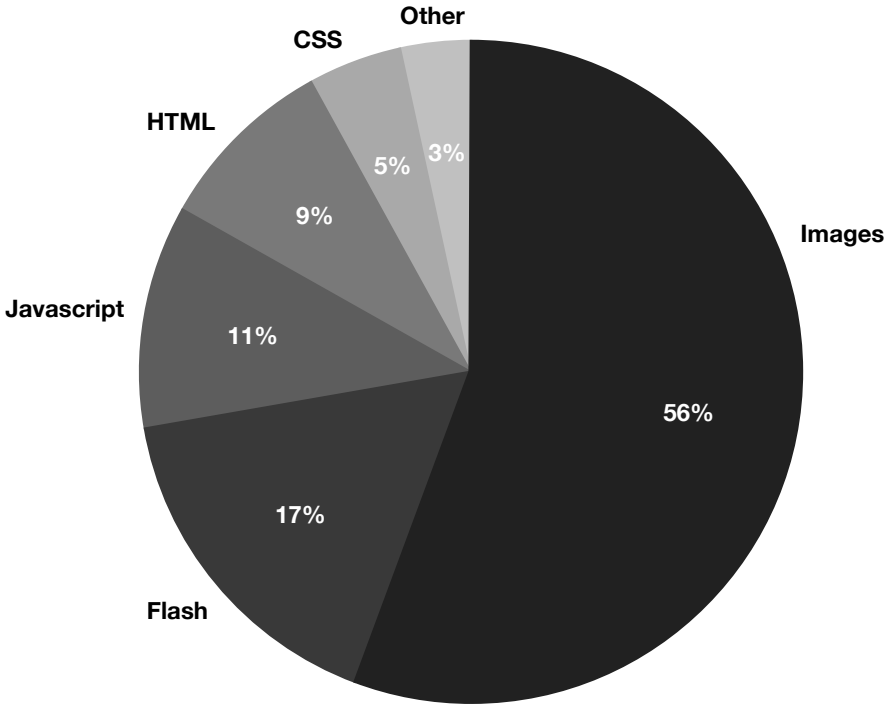


Figure 5.3: Breakdown of content types in terms of bytes

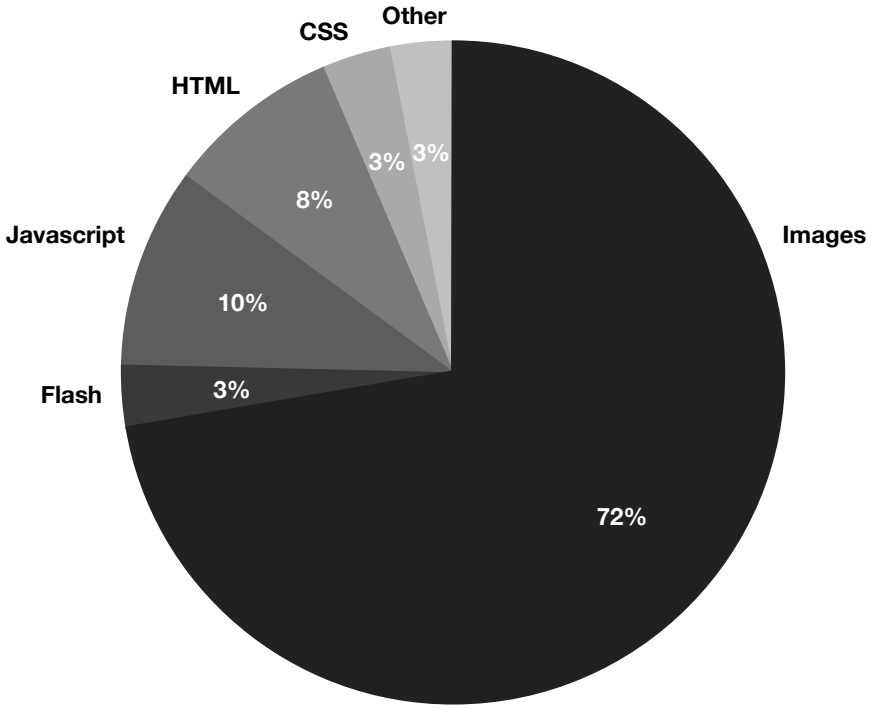


Figure 5.4: Breakdown of content types in terms of the number of files

Unsurprisingly, different types of images are the largest group, constituting 72% of all files. In terms of bytes, images make up 56% of the total amount. Flash content only accounts for 3% of files, but 17% of bytes. This can be explained by the fact that Flash files often contain animation or video that take up a lot of space. Another general observation is that the infrastructure of a web page (i.e., HTML and CSS) typically makes up a rather small part of the total – the majority is media content.

Conclusions

Based on the results obtained in the analysis, we can conclude that a typical web page consists of about 50 resources. Fetching such a page with regular HTTP-over-TCP would require 50 round trips from the client to the server, although a number of these might occur in parallel. With resource bundling we can reduce the number of round trips ideally to just one. In a high-delay environment, where the network delay dominates the additional latency introduced by the bundle layer, we can expect bundling to have a substantial positive impact on page retrieval performance.

5.2 Server Performance

When measuring the server performance, we are interested in two things: how much delay the bundle processing operations add to the server response cycle, and more importantly, how big an impact does bundling have on the page retrieval latency experienced by the user. The latter point is not so much a feature of the server as a characteristic of the entire bundling concept described in this thesis. However, decent request processing speed on the server is a prerequisite for sufficient performance of the entire system.

An important performance feature of a web server is its ability to deal with concurrent requests, i.e., serve multiple clients simultaneously. Due to Ruby's thread implementation, which is based on user-space threads, concurrency is a weak point in our implementation. However, as we have stated, our focus is not on optimizing the server performance and therefore the concurrency issues are not a primary concern.

Furthermore, we are naturally interested in how the implementation works in challenged network conditions for which the DTN architecture is originally designed. We perform measurements with artificially created delay to observe this.

Methodology

Typically, web server performance is evaluated by generating workload on the server using tools such as httperf [49] and measuring the number of requests per second the server can process. In our case, however, requests per second is not a good benchmark because we want to make a comparison between the DTN- and TCP-based modes of operation, which are inherently different. DTN-based communication stresses the server with fewer, but more laborious requests, whereas TCP-based operation causes a large amount of small requests. A logical way to compare these two modes of operation is to measure the performance experienced by the end user, which ultimately is the only thing that matters.

In order to achieve this, we used the Safari web browser [50] and its built-in web inspector tool. This tool allows us to measure the actual time between the instants when the browser sends the first request and when the page has been rendered. We used our DTN proxy to connect the browser to the DTN.

The server was run on a remote machine seven network hops away from the client. The round-trip delay between the two systems was very low and stable, the average over 100 pings being 1.47 ms with a standard deviation of 0.18 ms and zero packet loss. The client and server machines were running DTN2 bundle daemons with a direct route between them (via a TCP link) – thus, the overlay DTN was just a single hop. We configured the daemon on the server to use the filesystem for bundle storage. By default, DTN2 uses the Berkeley DB [51] database library but during development we observed that this is detrimental to the server performance – switching to filesystem storage made bundle processing orders of magnitude faster.

In addition to the low-delay environment described above, we performed measurements with artificially introduced delay. This was achieved simply by having the server hold up the response for the desired amount of time. The aim of this test was to demonstrate how network delay leads to quick deterioration of HTTP-over-TCP performance, while HTTP-over-DTN remains usable. While this simple method of creating delay is sufficient for our measurements, it does not realistically model network delay, which we will shortly point out.

The website we used for the measurements is a replication of the Mozilla Corporation's main product site *(www.mozilla.com)*. This website was chosen for testing because it is reasonably complex but does not contain excessive Javascript trickery or anything else that would cause problems. The measurements were conducted on the site's front page which consists of 29 resources: the HTML document, 4 Javascript files, 10 CSS files and 13 images. For the sake of comparison, measurements were also done on a simple self-contained HTML page.

The page retrieval measurements were done with three different server configurations. In the first case, the response bundle was created by using the parser. In the second case, the dependency file was used, and in the last one, the response was returned from the server cache.

It is important to notice that in this kind of a test setup there are numerous moving parts, all of which contribute to the final result. Ideally, what we would like to measure is just our own code. In reality, there is a good number of other factors – Mongrel, DTN-Ruby, DTN2, the operating system, network path, DTN proxy and web browser – that inevitably affect the result. Especially the proxy, being an unoptimized testing tool, produces a certain amount of latency on the client side. Therefore, we want to emphasize that the obtained numbers merely represent the results of this particular test run with this particular setup. However, this is not to say that no conclusions can be drawn from the results – the goal of the measurements is to get a rough idea of how the DTN-enabled server performs and to confirm that performance is not an obstacle to its deployment. For this purpose, this measurement scenario is sufficient.

In addition to the page retrieval measurements, we also observed the request processing times on the server at two different points of the response cycle. The first measurement excludes the bundle operations and observes only the time spent in the standard Mongrel processing cycle, including the HTTP handler in which the resources are gathered and aggregated into MHTML. The second measurement also includes the bundle operations, i.e. creating a *DtnSocket* from the request bundle, crafting the response bundle and passing it to the DTN2 daemon. These results give us an understanding on how taxing the bundling operations performed by the server are.

Results

Table 5.2 shows the results of the measurements conducted on the *www.mozilla.com* front page. The top two rows show the request processing times measured on the server, the first row excluding the time spent in bundle operations. The bottom row shows the full page retrieval time measured by Safari. The measurements were done through DTN with three different server configurations, as explained above. In each cell, the first two numbers are the minimum and maximum times and the last bolded number is the mean of ten measurements. The values are in milliseconds.

Table 5.2: Measurement results for www.mozilla.com

	DTN (Parser)	DTN (Dep. file)	DTN (Cache)	TCP
Request	43.3 / 65.8 / **53.6**	23.0 / 41.1 / **28.3**	2.5 / 2.9 / **2.7**	-
Bundle	53.7 / 201.8 / **82.4**	29.5 / 65.5 / **43.4**	8.0 / 28.2 / **16.7**	-
Page	905 / 1060 / **971**	534 / 647 / **566**	542 / 622 / **577**	359 / 479 / **435**

From the results we can see that HTTP-over-TCP yields the lowest page retrieval time. The result is not surprising because this is what Mongrel is optimized for. Server side processing times are not shown for TCP because they cannot be measured similarly to the DTN-based cases; the resources are requested one-by-one and therefore there is no single request processing time for the entire page.[1] Neither is there any bundling delay.

Also unsurprisingly, parser-based operation is the slowest of the DTN cases in terms of both server processing and page retrieval time. Whether the response bundle is created using the dependency file or served directly from cache has no impact on the final retrieval time. The server side processing of cached responses is considerably faster, although this is not reflected in the total page retrieval time because the slight advantage is lost in the other delays that occur during transmission and page rendering. However, in a situation where the server is overloaded with requests, we believe caching will be very useful in keeping the server responsive.

One of the reasons for the worse performance with parser-based operation is the fact that the resulting response bundle is considerably larger. The parser includes all re-

[1] The processing delays for these individual requests ranged from 0.5 to 50 milliseconds.

sources that are referenced in CSS files – even the ones that are not used on the bundled page. The reason for this is that the parser cannot know which of the resources are actually used unless it understands the CSS selector syntax. Implementing such a parser is far beyond the scope of this thesis, and performance-wise it might not even be reasonable. The payload of the bundle created with the parser contains 75 resources and adds up to 494 kilobytes in size. In the other two cases, there are 29 resources in a 161 kilobyte payload, so this obviously has an impact on the results.

We should also mention that the values on the second row in the table (i.e., request processing times with bundling delays) only show the time spent in the server application. The delay that occurs in the DTN2 daemon between the time instants of receiving the bundle from the application and transmitting it to the network is not included. For the test page, we observed this delay being about 150 milliseconds with little fluctuation.

The measurements were also conducted with a simple HTML page (681 bytes in size) that contained no embedded resources. In this case the operations of regular HTTP and HTTP-over-DTN are identical: a single request-response pair. Hence, we can also make comparisons of the server side request processing times. The different DTN modes (parser, dependency file, cache) would hardly make any difference in such a simple test and so only the dependency file, with no entry for the requested page, was used. The results – in the same format as earlier – are shown in table 5.3.

Table 5.3: Measurement results for the self-contained HTML page

	DTN (Dep. file)	TCP
Request	2.4 / 2.7 / **2.5**	0.7 / 0.7 / **0.7**
Bundle	7.0 / 84.7 / **19.8**	-
Page	116 / 175 / **137**	102 / 132 / **122**

These results reveal that the bundle operations have a rather modest impact on the final page retrieval time. However, the processing time on the server due to the bundling delay is considerably larger. In the scenarios for which our server is designed, i.e. light load and very modest concurrency, this is unlikely to have much consequences, but in an overload situation it might make a big difference.

The final page retrieval test was designed to show that adding network delay will not have too big an impact on the performance of DTN-based operation, whereas regular HTTP will suffer greatly from it. The delay was created artificially on the server, which – from the point of view of the client web browser connected to the proxy – is analogous to having delay in the network. Table 5.4 shows the results of the measurements. Notice that in this table, the measured times are shown in seconds.

Table 5.4: Measurements results for different delays (www.mozilla.com)

Delay	DTN (Dep. file)	TCP
50 ms	0.57 / 0.67 / **0.62**	0.81 / 0.93 / 0.**89**
100 ms	0.64 / 0.69 / **0.66**	1.36 / 1.61 / **1.45**
500 ms	1.07 / 1.22 / **1.12**	5.32 / 5.87 / **5.48**
1000 ms	1.53 / 1.93 / **1.69**	10.38 / 10.71 / **10.49**

The results are a strong indicator that the resource bundling scheme works. Expectedly, the performance of HTTP-over-TCP degrades drastically with increasing delay. This is simply due to the number of round-trips involved in the page retrieval process. Web browsers do use parallel TCP connections, somewhat mitigating the effects of delay, but nevertheless as the delay increases, each additional round-trip becomes very costly. HTTP-over-DTN, on the other, only requires one round-trip, provided that the server manages to bundle all relevant resources. Thus, the page retrieval time is essentially the sum of the round-trip delay and the server processing time. If the delay is further increased, the retrieval time of DTN increases on par with it, whereas TCP gets gradually worse – just how fast, depends on the number of resources – and finally fails completely.

However, we must point out that there is a caveat to our very simple method for creating delay: if the delay were truly in the network, it would also affect the TCP performance (e.g., the slow start mechanism). Thus, from the point of view of TCP, delay on the server is not equivalent to delay in the network. This fact does have an impact on the obtained numbers: network delay would slow down the TCP ramp-up phase, and therefore in a real scenario TCP would probably perform even worse. Thus, the outcome of the test would not change but DTN would be likely to win with an even greater margin.

We also did some concurrency testing on the server, using the httperf application for the TCP interface and a custom load generating script for DTN. As we have said before, the capabilities of the bundle-enabled server are hampered by the limitations of Ruby and its interaction with the DTN2 library. Thus, a comparison between the standard Mongrel TCP interface and the DTN interface is not really a fair one. Out of the box, Mongrel can handle 950 concurrent connections. In our quick tests, we were able to get 100 requests per second out of the standard Mongrel directory listing handler, which is not even designed for serving files directly. On the other hand, the DTN interface worked adequately up to about 10 concurrent processing threads, although with a noticeable slowdown. However, it is not clear what exactly is causing this slowdown. A significant source of delay seems to be the DTN2 bundle daemon which, for some reason, upon being overloaded with bundles slows the outgoing transmission rate to a trickle.

Conclusions

As a general conclusion, we can say that our server is *fast enough*. When browsing the test website, everything seemed snappy and the measurement results also back this up. During testing we did not run into problems that might prevent the server's real-life usage. Keeping in mind that the server is only designed for small-scale deployments, we believe it is a usable application. In this context the concurrency issue is also a non-issue.

It is also worth mentioning that Mongrel, despite having good performance for a Ruby-based implementation, is not usually run as a standalone server. Typically, Mongrel is used as a back-end application server with a separate front-end, such as Apache or Nginx, facing the public Internet and acting as a reverse proxy. For serving static files, these heavyweight web servers are still a notch faster.

5.3 Parser Accuracy

If the parser is used on the server to find resource dependencies, it is very important that it works well. If the parser does not find the dependencies with sufficient accuracy, the server will essentially revert to regular HTTP-style one-by-one fetching, which is exactly what we want to avoid. In theory, testing the parser accuracy is easy: make a local copy of a web page, make it available on the server and access it with a web browser through the DTN proxy. The number of subsequent requests sent by the browser after receiving the initial response bundle equals the number of resources the parser failed to find. If there are no further requests, the parser found all relevant resources. The test could be performed on a large number of web pages to obtain the average hit ratio.

However, it seems that in practice it is rather difficult to test the parser. The tests should be performed on real websites in order to obtain real-life data, but retrieving a complete local copy of a web page is problematic. If the local copy is not complete, the parser cannot possibly find all dependencies and therefore its accuracy cannot be judged. We have attempted retrieving local copies with two different methods, but neither of these have proven reliable enough.

Modern web browsers, such as Safari and Firefox [52], include a feature that allows the user to save an entire web page on the disk. In Firefox, the HTML page and its embedded resources are saved as separate files. However, this feature really does not work that well. First, for some reason Firefox modifies the source of the HTML page by adding extraneous elements to it. Thus, the saved page is no longer identical to the one on the remote server. Second, Firefox fails to save all the dependent resources. For example, CSS files that are linked to the page using the *@import* statement are not saved. It is baffling why this happens – if Firefox is able to render the page properly, why can it not similarly save the page? Safari seems to fare better in this regard, but it saves the web pages in a proprietary archive format instead of separate files. This prevents using these copies on the local server.

We also tried using Wget [53] to fetch web pages. Wget is a non-interactive command line tool for retrieving remote files using the most common Internet protocols. Wget can be used to save complete web pages by instructing it to follow the links on a page recursively to a certain depth. The main problem with Wget is that it essentially is just another parser, and an imperfect one at that. We cannot rely on Wget being able to find all the relevant resources and hence cannot use it for testing our own parser. A quick test revealed that also Wget fails to recognize the *@import* statement. It seems that obtaining complete local copies of web pages would require a considerable amount of manual labor.

The problem of fetching copies of web pages is not inherently difficult. After all, if a browser displays a web page, a local copy already exists in the browser cache. However, there seems to be no simple practical solutions to this problem – only tedious ones. The number of web pages on which to test the parser should be sufficiently large in order to have statistically sound results. If the manual workload associated with each test case is large, performing this kind of a test is not feasible. Therefore, we do not conduct such a test.

Conclusions

In section 4.4, we explained how the parser works and presented some of its known limitations. One limitation, which is not dependent on just the parser but the bundling scheme as a whole, is that external dependencies are excluded. Basically, this means that the parser does not recognize references that contain a full URI, such as *http://static.server.com/image.gif*, unless the domain part of the URI matches the server's own domain. The reason for this behavior is that the server obviously cannot bundle resources which it does not have direct access to, and as consequence, our server application cannot be used with websites that want to distribute their content from multiple separate hosts.

Another case when the parser will fail to find resources is when they are embedded within Javascript code. For example, a page might want to display an image randomly from a set of alternatives on each page load. This functionality may be implemented on the server side, in which case the parser would work fine, but it also might be done with Javascript on the client side. The image URIs might be placed in a Javascript array from which one would be chosen randomly and inserted to the page. The parser would not detect these images and we can even argue that they should not be detected – after all, if all the images were included in the bundle, all but one of them would remain unused.

On the test website (i.e., the local version of *www.mozilla.com*), the parser only misses one dependent resource – an image reference embedded in Javascript code. Generally, we can conclude that the parser works well enough considering that it is actually designed to be just a fallback mechanism. We assume that the dependency file will be primarily used for declaring resource dependencies.

5.4 Summary

The primary justification for the concept of resource bundling is that web pages consist of multiple resources, which in the case of HTTP-over-TCP leads to multiple round-trips. The website analysis presented in this chapter reveals the concrete numbers behind the word "multiple". This knowledge allows us to estimate the size of the bundles that will pass through the system, which in turn might be used as a basis for design decisions in other related contexts. The results of the analysis are perhaps a bit surprising: the average web page contains more than 50 resources, which are for the most part image files.

We also presented the results of the server performance measurements in this chapter. In short, these results show that the overhead of the bundle layer operations performed by the server is not very significant. Thus, even in a low-delay environment performance does not prohibit the deployment of the DTN-enabled server, although its scalability is limited by concurrency issues. In a challenged network environment with long delays, DTN-based operation easily outperforms regular HTTP-over-TCP.

This chapter concludes the content part of the thesis; in the next one, we draw conclusions on the work done and discuss some ideas for future development.

6 Conclusions

In this thesis, we have presented an approach to enabling DTN-based web access. In the first chapter, we discussed two perspectives from which this problem – running HTTP on top of the bundle protocol – should be considered: transport perspective and application perspective. We have discussed the necessary requirements for using DTN transport under HTTP, but the greater part of this thesis has explored application level issues, i.e., the way in which HTTP transfers web pages and how this can be adapted to the DTN environment.

Regular HTTP-over-TCP is inherently incompatible with the key concepts of the DTN paradigm. Therefore, running HTTP on top of DTN cannot be done trivially by mapping HTTP messages directly onto bundles. In challenged network environments, it is paramount to avoid unnecessary round-trips because they can be very costly. HTTP, on the other hand, is a conversational protocol that fetches each web resource with a separate request which means that retrieving one web page requires a number of round-trips equal to the number of resources comprising the page. The solution presented in this thesis for redeeming this mismatch is resource bundling.

Resource bundling means that the DTN-enabled web server does not send individual resources in its responses, but instead bundles of multiple resources, typically an entire web page. This reduces the number of round-trips required to retrieve a web page ideally to just one. We defined the necessary prerequisites for resource bundling: an aggregation format for the resources and a method for the server to infer resource dependencies. We opted for MHTML as the primary aggregation format on the grounds that it is a relatively established format designed for the exact purpose of transferring web pages, the MIME headers provide means for conveying resource metadata, and it does not incur excessive overhead. For finding resource dependencies on the server, we presented two methods; the first one based on explicitly listing the dependencies in a file, and the second one on parsing the HTML source of the requested document.

The practical contribution of this thesis is a web server application that implements the devised resource bundling scheme. The two largest building blocks of the server are the DTN2 reference implementation and the Mongrel HTTP server. Leveraging from these two components, we created a web server with native support for the bundle protocol and MHTML-based resource bundling.

An obvious step for future development is a DTN-enabled web browser. The proxy-based solution we have used feels more or less like a crude development and testing tool rather than a finished product. A native DTN web browser that is aware of the potentially lengthy page retrieval operations and has a user interface specifically designed for this would be very useful in furthering the DTN-based web.

As for developing the server application further, there is plenty of room for improvement. The concurrency issues stemming from Ruby's threading model should be resolved in order to produce a truly production-ready server. Ruby's upcoming releases promise support for native threads which – coupled with a Ruby-based bundle router implementation [24] – might be viable alternative for creating a truly portable web server with native DTN support. It is also conceivable that an event-based architecture [54] could be used instead of a multithreaded one. We can also think of other improvements, such as better support for different bundle payload types, more efficient caching mechanisms and clustering support.

The main issues with the resource bundling scheme are related to dynamic web content. As the technologies behind the web evolve and mature, web pages are becoming increasingly dynamic and interactive. Modern AJAX-based web applications are blurring the line between the desktop and the web. Clearly, the Internet is no longer just a medium for linked static text documents but also a platform for a myriad of rich web applications. On a technical level, the interactivity of the modern web is often implemented in a way that requires more and more frequent communication between the client and the server, thus raising the bar for DTN compatibility. Resolving this disparity is a challenge that calls for new technological solutions and standards. The HTML 5 working draft includes features, such as client-side database storage, that aim to enhance offline capabilities of web applications, hence also benefiting DTN-based web access [55].

As the Internet stretches out to ever more diverse network environments, it seems likely that delay-tolerant networking will play a role among future networking technologies. Whether the deployed DTN architecture is the current one from DTNRG, its successor or something completely different remains to be seen. In any case, research in the DTN field is advancing rapidly. One interesting question is whether DTN will remain as specialized solution for challenged networks or could delay-tolerance become a feature of the Internet in general. In other words, could the bundle protocol (or some other DTN transport protocol) replace IP as the all-encompassing baseline of the Internet. Further research is required to determine whether this is feasible, but nevertheless, the idea might be worth pursuing because universal delay-tolerance certainly holds interesting prospects for future applications.

Bibliography

[1] Jörg Ott, Dirk Kutscher, "Bundling the Web: HTTP over DTN", 2006

[2] Jörg Ott, Dirk Kutscher, "Applying DTN to Mobile Internet Access: An Experiment with HTTP", 2005

[3] Caitlin Holman, Khaled A. Harras, Kevin C. Almeroth, Anderson Lam, "A Proactive Data Bundling System for Intermittent Mobile Connections", 2006

[4] V. Padmanabhan, J. Mogul, "Using Predicive Prefetching to Improve World Wide Web Latency", 1996

[5] Zhimei Jiang, Leonard Kleinrock, "Web Prefetching in a Mobile Environment", 1998

[6] Mozilla Link Prefetching FAQ, *http://developer.mozilla.org/en/docs/Link_prefetching_FAQ*

[7] L. Wood, P. Holliday, "Using HTTP for delivery in Delay/Disruption-Tolerant Networks", Internet-Draft, 2008

[8] Jörg Ott, Mikko Pitkänen, "DTN-based Content Storage and Retrieval", 2007

[9] Mikko Pitkänen, Jörg Ott, "Redundancy and Distributed Caching in Mobile DTNs", 2007

[10] Kevin Fall, "A Delay-Tolerant Network Architecture for Challenged Internets", 2003

[11] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, E. Travis, H. Weiss, "Interplanetary Internet (IPN): Architectural Definition", 2001

[12] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", RFC 3986, 2005

[13] W. Eddy, "Internet-Draft: Architectural Considerations for the use of Endpoint Identifiers in Delay Tolerant Networking", 2006

[14] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss, "Delay-Tolerant Networking Architecture", RFC 4838, 2007

[15] Forrest Warthman, "Delay-Tolerant Netwokrs (DTNs): A Tutorial", 2003

[16] Stephen Farrell, Vinny Cahill, "Delay- and Disruption-Tolerant Networking", 2006

[17] Amin Vahdat, David Becker, "Epidemic Routing for Partially-Connected Ad Hoc Networks", 2000

[18] A. Lindgren, A. Doria, “Probabilistic Routing Protocol for Intermittently Connected Networks”, Internet-Draft, 2007

[19] Stephen Farrell, S. Symington, H. Weiss, P. Lovell, “Delay-Tolerant Networking Security Overview”, Internet-Draft, 2007

[20] S. Symington, Stephen Farrell, H. Weiss, P. Lovell, “Bundle Security Protocol Specification”, Internet-Draft, 2007

[21] K. Scott, S. Burleigh, “Bundle Protocol Specification”, RFC 5050, 2007

[22] Michael Demmer, Jörg Ott, “Delay Tolerant Networking TCP Convergence Layer Protocol”, Internet-Draft, 2007

[23] DTN2 Reference Implementation, *http://dtnrg.org/wiki/Code*

[24] RDTN, *http://dev.tzi.org/retrospectiva/projects/rdtn/wiki/rdtn*

[25] David Gourley, Brian Totty, “HTTP: The Definitive Guide”, 2002

[26] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, “Hypertext Transfer Protocol - HTTP/1.1”, RFC 2616, 1999

[27] B. Krishnamurthy, J. Mogul, D. Kristol, “Key Differences between HTTP/1.0 and HTTP/1.1”, 1998

[28] J. Palme, A. Hopmann, N. Shelness, “MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)”, RFC 2557, 1999

[29] Jonathan Postel, “Simple Mail Transfer Protocol”, RFC 821, 1982

[30] N. Freed, N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies”, RFC 2045, 1996

[31] N. Freed, N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types”, RFC 2046, 1996

[32] Jörg Ott, Teemu Kärkkäinen, Mikko Pitkänen, “Application Conventions for Bundle-based Communications”, Internet-Draft, 2007

[33] Oren Ben-Kiki, Clark Evans, Brian Ingerson, “YAML Ain't Markup Language, Version 1.1”, Final Draft, 2005, *http://www.yaml.org/spec/1.1*

[34] Nina Bhatti, Anna Bouch, Allan Kuchinsky, “Integrating User-perceived Quality into Web Server Design”, 2000

[35] Google Gears, *http://gears.google.com*

[36] Adobe AIR, *http://labs.adobe.com/technologies/air*

[37] T. Dierks, E. Rescorla, “The Transport Layer Security (TLS) Protocol, Version 1.1”, RFC 4346, 2006

[38] B. Ramsdell, “Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1, Message Specification”, RFC 3851, 2004

[39] DTN-Ruby, *https://prj.tzi.org/cgi-bin/trac.cgi/wiki/Dtn-ruby*

[40] Mongrel, *http://mongrel.rubyforge.org*

[41] Matt Pelletier, Zed Shaw, “Mongrel: Serving, Deploying, and Extending Your Ruby Applications”, 2006

[42] RubyMail, *http://www.rfc20.org/rubymail*

[43] RSpec, *http://rspec.info*

[44] Dave Ragget, Arnaud Le Hors, Ian Jacobs, eds. “HTML 4.01 Specification”, W3C Recommendation, 1999

[45] Hpricot, *http://code.whytheluckystiff.net/hpricot*

[46] Anne van Kesteren, ed. “HTML 5 differences from HTML 4”, W3C Working Draft, 2008

[47] Alexa Internet, *http://www.alexa.com*

[48] Charles Web Debugging Proxy, *http://xk72.com/charles*

[49] David Mosberger, Tai Jin, “httperf - A Tool for Measuring Web Server Performance”, 1998

[50] Safari, *http://www.apple.com/safari*

[51] Berkeley DB, *http://www.oracle.com/technology/products/berkeley-db*

[52] Firefox, *http://www.mozilla.com/en-US/firefox*

[53] Wget, *http://www.gnu.org/software/wget*

[54] Evented Mongrel, *http://swiftiply.swiftcore.org/mongrel.html*

[55] Ian Hickson, ed. “HTML 5”, WHATWG Working Draft, 2008