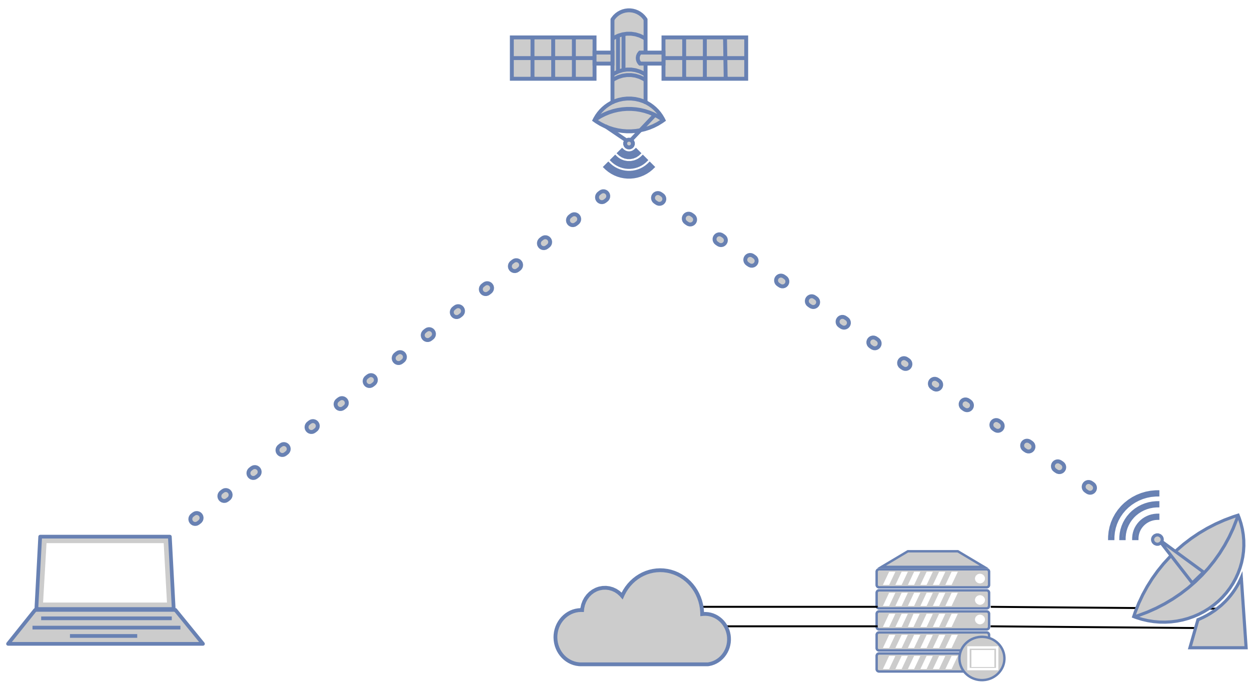


Very Low Bandwidth (Marine) Web Surfing

A Fault-Tolerant Content Streaming Web Browsing Solution

Bachelor of Science Thesis in Computer Science and Engineering

Arvid Hast, Frej Karlsson, Jesper Lindström,
Lina Blomkvist, Tobias Andersson, Tobias Sundell

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Bachelor of Science Thesis DATX02-18-43

Very Low Bandwidth (Marine) Web Surfing

A Fault-Tolerant Content Streaming Web Browsing Solution

Arvid Hast
Frej Karlsson
Jesper Lindström
Lina Blomkvist
Tobias Andersson
Tobias Sundell

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden June 2018

Very Low Bandwidth (Marine) Web Surfing
A Fault-Tolerant Content Streaming Web Browsing Solution

Arvid Hast, hasta@student.chalmers.se

Frej Karlsson, frejk@student.chalmers.se

Jesper Lindström, jeslinds@student.chalmers.se

Lina Blomkvist, bllina@student.chalmers.se

Tobias Andersson, tobande@student.chalmers.se

Tobias Sundell, tobsun@student.chalmers.se

Supervisor: Dag Wedelin
Examiner: Peter Damaschke

Bachelor of Science Thesis DATX02-18-43
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Figure by the authors. The illustration pictures the connection between the computer and the Internet, using a satellite connection and a proxy server.

Gothenburg, Sweden 2018

Very Low Bandwidth (Marine) Web Surfing
A Fault-Tolerant Content Streaming Web Browsing Solution

Arvid Hast, Frej Karlsson, Jesper Lindström
Lina Blomkvist, Tobias Andersson, Tobias Sundell

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Satellite connections are often the only option for Internet connectivity in remote or isolated regions. One low-cost example is the Iridium GO! satellite Internet device, with a bandwidth of just 2.4 kbit/s, at which many web pages would take several hours to load. Furthermore, satellite Internet is prone to connection issues and packet loss, increasing loading times even further.

The possibility of enabling basic web browsing over such very slow connections is investigated. As a proof of concept, a prototype functioning under these conditions was developed. The prototype consists of a performance enhancing proxy server, a browser client, and a protocol. In the prototype certain content is filtered and rendered into plain text. This text is then streamed in smaller parts to the client.

The prototype was tested with Iridium GO! and compared to some similar software solutions. The tests show that the proposed solution is both reliable and sufficient for basic web browsing on very low bandwidth. The prototype is a good foundation for further development of a browser for low bandwidth connections.

Keywords: satellite Internet, low bandwidth web browsing, Iridium, performance enhancing proxy, content extraction, networking.

Very Low Bandwidth (Marine) Web Surfing
A Fault-Tolerant Content Streaming Web Browsing Solution

Arvid Hast, Frej Karlsson, Jesper Lindström
Lina Blomkvist, Tobias Andersson, Tobias Sundell

Department of Computer Science and Engineering
Chalmers University of Technology

Sammandrag

Satellituppkopplingar är ofta de enda tillgängliga alternativen för internetanslutningar i avlägsna eller isolerade regioner. Ett exempel på en sådan uppkoppling är Iridium GO!, ett modem för satellit-internet med en bandbredd på endast 2.4 kbit/s. Med en så låg bandbredd tar många webbsidor flera timmar att ladda. Satellit-internet lider också av uppkopplingsproblem och tappade nätverkspaket, vilket får laddtiderna att öka ytterligare.

Huruvida det går att möjliggöra enkelt webbsurfande över sådana anslutningar undersöks. Som konceptvalidering utvecklades en prototyp som fungerade under dessa omständigheter. Prototypen består av en prestandaförbättrande proxyserver, en webbläsarklient och ett protokoll. I prototypen filtreras visst innehåll och återges som ren text. Texten strömmas sedan i små delar till klienten.

Prototypen testades på Iridium GO! och jämfördes med några liknande mjukvarulösningar. Testen visade att den föreslagna lösningen både är pålitlig och tillräcklig för enkelt webbsurfande på en mycket låg bandbredd. Prototypen är en god utgångspunkt för ytterligare utveckling av en webbläsare för uppkopplingar med låg bandbredd.

Nyckelord: Satellit-internet, låg bandbredd, webbsurfande, Iridium, prestandaförbättrande proxy, extrahering av innehåll, nätverkande.

Acknowledgments

We would like to thank our supervisor Dag Wedelin for his continuous support, as well as providing us with his personal Iridium Go! device for testing. His experience from using the device at sea gave us invaluable user insight of the problems that arise with web browsing using satellite connections.

Apart from our supervisor, we would like to thank everyone else who has provided us with feedback throughout the process.

Contents

Glossary

Cookies A piece of data sent as part of the HTTP header which allows web pages to store small amounts of information in a user's browser and access said information again. Often used to remember the state of a certain user, such as whether he is signed in or not.

CRC A Cyclic Redundancy Check is a common type of error-detecting code. CRC8 is an 8-bit variant.

DOM A "Document Object Model" (DOM or DOM-tree) is a tree model representation of the HTML elements on a web page.

Handshake A process where two communicating parties negotiate the parameters of communication, for example establishing a TCP connection.

HTML A markup language used to define the content and structure of a web page.

HTTP A network protocol used to transmit hypertext. Commonly used to transmit web pages.

Multiplexing A process of combining several different signals into a single signal. Usually followed by the inverse process of demultiplexing, to regain the original signals.

NAT A "Network Address Translation" allows several devices with unique local IP addresses to share a single external IP address, by forwarding ports from the external address to internal addresses on demand.

Network protocol A specification of how to communicate with certain software, such as accepted packet header values.

Nonce Short for "Number Used Once". A cryptographic concept of a number that should never be reused but need not be kept secret.

Packet A sequence of bits which are transmitted together to another device in the network.

Packet header The part of a packet containing control information, such as address information about the recipient and sender. Always the first part of a packet sent.

Padding Meaningless bits added to something to ensure that it meets certain size requirements.

Payload The part of a packet containing the transmitted data, excluding control information.

RTO Short for "Retransmission Time Out". The time after which an unacknowledged packet is retransmitted in TCP.

RTT Short for "Round Trip Time". The time it takes for a sender to receive an acknowledgment from the receiver after transmitting a packet over TCP.

TCP A network protocol providing reliable in-order delivery.

UDP A simple, "best effort" network protocol. A sent packet is not guaranteed to arrive at its destination.

1 Introduction

In many remote locations, Internet access is very slow and costly compared to that of an average city in the developed world. For instance, seagoing ships use satellite Internet solutions to stay connected. Large vessels have more advanced communication solutions, but many recreational boats depend on slow entry-level solutions. The slower solutions provide access to weather reports and text-based emails through specialized software.

The average bandwidth of mobile and broadband connections has increased greatly over time, while the bandwidth of mobile satellite Internet solutions has remained largely unchanged. The size of an average web page has also increased, and graphics, advertisements, and layout can make up several megabytes of the web page. Because of this, transmitting an average web page on a consumer grade satellite Internet connection can take hours. However, the actual text content is often only a few kilobytes.

Consumer grade satellite Internet connections are sufficient for phone calls and simple text-based messages but it is practically impossible to download full web pages in a reasonable time frame. Recreational sailors might still want to access certain web-based content, such as news articles or technical information. The development of a web browsing solution that allows a user to access such content on a very slow satellite Internet connection is therefore desirable.

1.1 Current State of Mobile Satellite Internet

Entry-level satellite Internet solutions, such as Iridium GO!, provide a very low bandwidth (Abdul Jabbar & Frost, 2003), about 50 000 times smaller than that of a modern 4G connection. The Iridium GO! device is widely used where Internet access is scarce, and the service is a good example of a consumer grade satellite Internet connection. Enterprise-grade solutions with higher bandwidth do exist (KVH Industries, 2018), but these are costly compared to slower satellite Internet connections.

Iridium, as well as some third-party developers, offers specialized applications for various services such as weather, email and web browsing. However, web browsing applications such as XWeb only reduce the web page by a factor of three to five (Global Marine Networks, 2018), which is not enough to make web browsing with reasonable loading times possible. No published research on how to enable web browsing on high latency low bandwidth connections was found.

1.2 Purpose

The purpose is to investigate how to enable basic web browsing on a connection with very low bandwidth and high latency. Specifically, a prototype of a fault-tolerant content streaming web browsing solution has been designed, constructed, and tested.

1.3 Challenges and Subproblems

The goal is to achieve basic web browsing with an acceptable loading speed so that new content of a web page is loaded before the reader has finished viewing the previously loaded content of the page. The size of an average web page also needs to be reduced by a factor of over 1000 to be viewable on an entry-level satellite connection within a reasonable time. This may possibly be achieved by extracting only the relevant text content. If the resulting text content can also be streamed to allow the user to read from the top almost instantly, pages of larger size may still be possible to access.

Apart from removing excessive information, several parts of the web browsing technology, such as the network protocol and the text markup language, need to be optimized and adapted for the connections' low bandwidth, high latency, and instability. The different subproblems identified can be grouped into the following categories:

Content Extraction To be able to efficiently browse the web on a low bandwidth connection, excessive information and resources need to be removed from web pages prior to transmission to the client. Only the most important parts of the web page should be kept, such as the text content and navigational links.

Web Searching and Forms An important aspect of web browsing is the ability to interact with the web pages, such as searching for a particular topic on a search engine. This project investigates how basic HTML server-side forms can be made usable on a low bandwidth connection, without loading the entire web page.

Network Protocol As the data may be transmitted on an unstable connection, packages may fail to be delivered. Therefore, the handling of errors is of high importance. For instance, the solution must support automatic or manual re-transmission of missing content.

Compression Compression is needed to reduce the bandwidth usage further. In many compression algorithms, a single bit error might corrupt the entire compressed message. A compression solution that is less vulnerable to bit errors is therefore needed.

Encryption Encryption is problematic if bit errors are to be tolerated. For most ciphers, a single bit error will leave the whole package unreadable.

1.4 Delimitations

The goal of this project is to make web pages viewable with a low bandwidth connection, primarily regarding information filtering, content streaming, and how to efficiently transfer data over a slow connection. Because of time constraints, the scope of the project is limited and does not include:

- Any physical work to improve Internet connection.
- An advanced user interface.
- Web page scripting on the client.
- Multi-media streaming.
- Displaying or downloading of images.

1.5 Outline

The rest of the thesis is divided into seven chapters. The relevant theory is presented in Chapter 2. In Chapter 3 the methods of the project are presented. Some possible technical solutions for the prototype are evaluated in Chapter 4, and Chapter 5 describes the final implementation of the prototype.

Test results for both field tests and simulations are presented in Chapter 6. Chapter 7 discusses the test results and the relative importance of the different sub-problems. This chapter also includes suggestions for further development of the prototype and takes up a few ethical aspects. Finally, Chapter 8 presents the conclusions drawn from the discussion and evaluates the purpose presented in 1.2.

2
Theory

This chapter introduces the relevant theory behind networking, web browsing, proxy servers, and encryption. To clarify the constraints of satellite Internet solutions, this chapter also provides a brief introduction to communication satellites. The theory presented is required to understand the particular difficulties and considerations of a fault-tolerant text-streaming web browsing solution on low bandwidth connections.

2.1 Network Communication

Computer networking can be divided into several communication layers, where each layer hides the details of the underlying layers. As seen in Figure 2.1, the communication layers of the Internet protocol stack range from low-level electronics to high-level software and are the following: physical, data-link, network, transport, and application (Kurose & Ross, 2012, Ch. 1.5.1). As the physical and data-link layer are tied to the hardware and thus are beyond the scope of this project, they are only introduced briefly for the purpose of completeness.

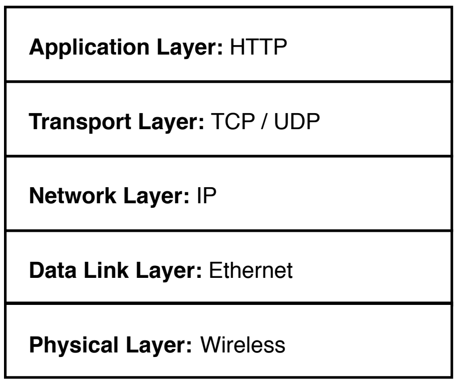

Application Layer: HTTP
Transport Layer: TCP / UDP
Network Layer: IP
Data Link Layer: Ethernet
Physical Layer: Wireless

Figure 2.1: Communication layers with example applications.

2.1.1 Physical and Data-Link Layer

The physical layer defines how the transmission of data should be performed by the electronics, such as how to transmit a certain bit pattern in terms of voltage levels. The data-link layer is responsible for grouping the received bit patterns into *frames*, and to ensure that these are correct (Kurose & Ross, 2012, Ch. 5.1.1).

2.1.2 Network Layer

In the network layer, data is sent as a *packet*, which contains a *payload* and a *packet header*. The packet header contains information used by the protocol, such as information about how to reach the destination. This information is called metadata.

A common network layer protocol is the Internet Protocol (IP), which defines how devices can communicate with each other in terms of routing the packets. Each device is assigned an IP address which contains information on where to deliver the packets. All packets that are transmitted to another device through such a computer network, therefore, need to contain the sender and recipient IP address. This is called the IP header (Kurose & Ross, 2012, Ch. 4.4).

2.1.3 Transport Layer

Similarly to how the network layer defines how *devices* communicate with one another, the transport layer defines how data should be transferred between *applications* running on the devices (Kurose & Ross, 2012, Ch. 3.1.5). The transport layer provides different services to the applications, such as connections, guaranteed delivery, and fair sharing of network resources. The two major transport level protocols, TCP and UDP, are described in Section 2.5.

2.1.4 Application Layer

The application layer defines how software applications communicate at a high level. It defines how authentication between two devices should happen, as well as how to compress and decompress the data. Finally, the application layer is where a software application defines how it is communicating with applications on other devices (Kurose & Ross, 2012, Ch. 2.1). An example of an application layer protocol is *HTTP* which is used for web browsing (Goralski, 2017, Ch. 6).

2.2 HTTP

The Hypertext Transport Protocol (HTTP) is a protocol used to transfer web content, primarily in hypertext, which is the text format used by the World Wide Web (The Internet Society, 1999). HTTP relies on a transport level protocol for communication between applications (for example, a web server and a web browser). The transport protocol most often used is TCP, and most web servers will not respond to any other transport protocol.

HTTP is a text-based protocol, in contrast to unreadable binary protocols. For example, part of an HTTP request might be written as `GET /index.html HTTP/1.1`. Apart from the message body, HTTP messages contain so-called *headers* consisting of additional information not part of the message itself. Since the number of commonly used headers is large, and the headers are in plain text, the overhead introduced by HTTP is frequently thousands of bytes.

2.3 Web Browsing

A web browser is a client that requests web pages from a web server. When a user enters a web address into the web browser, an HTTP request is sent to the web server which in turn replies with a document, normally written in the markup language HTML. The HTML document contains information about the page's layout, its content, and its resources such as images and scripts. The web browser is responsible for displaying the page according to the HTML document and also makes new requests for any resources needed to fully display the web page. Once all resources have been requested and retrieved, the web page is completely visible in the web browser window (Stanford, n.d.).

2.4 Proxy Servers

A proxy server is an application used as an intermediate step in a resource request between a client and a server. There are different kinds of proxy servers with different use cases and applications. A relevant application is Performance Enhancing Proxies (PEP), which are used to improve performance in either the transport or application layer (Border, Griner, Montenegro, Shelby, & Kojo, 2001). At the transport layer, a PEP can be used to improve the stability of communications over unreliable networks, including satellite communication links (Caini, Firrincieli, & Lacamera, 2006). An example of an application layer PEP is the "Data Saver" mode in Google Chrome, which compresses the HTTP traffic (Google, n.d.).

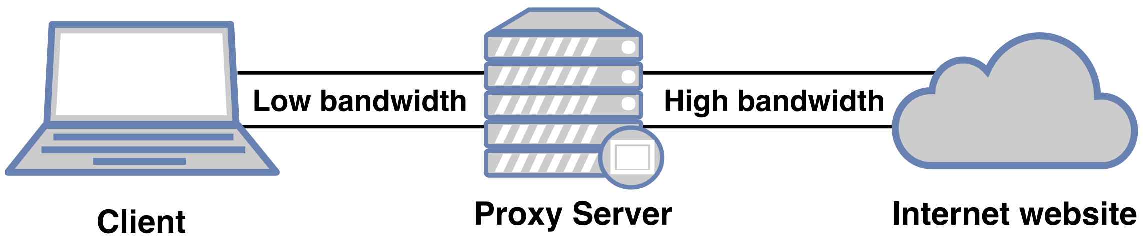


Figure 2.2: Performance enhancing proxy server.

A performance enhancing proxy server is illustrated in Figure 2.2, where the client has a low bandwidth connection and the proxy server has a high bandwidth connection. Hence, such a proxy server can perform performance enhancing operations, such as compression, before transmitting the result to the client. The proxy server described in 5.2 is a performance enhancing proxy.

2.5 Major Transport Level Protocols

Transmission Control Protocol (TCP) and *User Datagram Protocol* (UDP) are the two major transport layer protocols. They are discussed in this section at some length because they are later considered for the prototype in Section 4.3. TCP provides reliable, in-order delivery of packets while UDP does not.

Both UDP and TCP have *checksums* that are used to detect any errors that occur during transmission. Such an error is called a "bit error" and occurs when at least one bit has changed value during the transmission. If a TCP packet is found to be faulty, it will be retransmitted until it is correctly received. By default, a faulty packet is simply discarded by UDP. On IPv4, the fourth version of IP, the UDP checksum is optional and if disabled even faulty packets are accepted (Postel, 1980).

2.5.1 TCP

A core mechanic in TCP is automatic retransmission. The receiver sends acknowledgments (ACK) for any correctly received packet, and the sender retransmits unacknowledged packets after a certain period called *Retransmission Timeout* (RTO). RTO is based on the estimated *Round Trip Time* (RTT), which is the time it takes for the sender to receive an acknowledgment from the receiver after transmitting a packet. TCP continually estimates the RTT in order to adapt to any changes in the network conditions (Goralski, 2017, Ch. 12).

TCP also provides *flow control* and *congestion control* algorithms to avoid overflowing buffers, which otherwise would result in packet loss. Flow control adjusts the rate of the data transmission by having the recipient continually adjusting a limit on the amount of data it can receive, and the sender never exceeding this limit (Goralski, 2017, Ch. 12).

Network congestion occurs when excessive data is sent over a network channel. The congestion control algorithm is designed to use as much of the available bandwidth as possible while sharing it fairly with other users (Goralski, 2017, Ch. 12). The algorithm assumes that every packet loss is caused by congestion. Therefore, if a packet has exceeded the RTO unacknowledged, the data transmission rate is lowered.

When a TCP connection is established, a process known as *handshake* is initiated. The client sends a synchronize (SYN) packet to the server containing a sequence number x. The server responds with a SYN-ACK packet containing a sequence number y and an acknowledgment of x. Finally, the client responds with an acknowledgment of y (Kurose & Ross, 2012, Ch. 3.5.6).

Because of its in-order delivery guarantee and congestion control, TCP is suitable when reliability and fair sharing of the bandwidth between multiple parties is more important than a timely delivery of data. For example, when a packet is lost during transmission, no subsequent packets can be delivered until the lost packet has been successfully retransmitted, incurring a delay (Kurose & Ross, 2012, Ch. 3.5.4).

2.5.2 UDP

UDP is a simple protocol, with a tiny overhead: the headers are just 8 bytes, as opposed to TCP at typically 20 bytes. The only services provided by UDP are information about the sender port, destination port, and a checksum (Postel, 1980). The absence of services presents many opportunities for customization, and make UDP suitable as a basis for the development of alternative protocols.

UDP does not provide correct ordering of packets or guarantee that they are successfully delivered. This makes it less vulnerable to connection errors and network congestion. Therefore, UDP is often preferable to TCP in, for example, Internet phone services, where some slightly broken speech is preferred over pausing and waiting for correct delivery of the audio (Cowley, 2013, pp. 81-109).

2.6 Encryption

Encryption can be used to ensure the confidentiality of data sent over an insecure communication channel, ensuring that the data is viewable only to the sender and the intended receiver (Menezes, Oorschot, & Vanstone, 2001, Ch. 1). This is achieved by encoding the data using a secret value s_1, usually called a key, in such a way that only someone who has a certain secret value s_2 can decode it to get the original data. Encryption schemes are generally divided into two types: *symmetric encryption* where $s_1 = s_2$, and *asymmetric encryption* where $s_1 \neq s_2$.

Asymmetric encryption requires more processing power, and therefore symmetric encryption is generally used to encrypt any larger amounts of data. Asymmetric encryption is often used to first establish the key used in symmetric encryption so that the data can then be sent using symmetric encryption. Both symmetric and asymmetric encryption are used in the prototype. Asymmetric encryption is used to generate a shared key between the server and client, and symmetric encryption is used to encrypt the data (see 4.5 and 5.5).

The most common type of symmetric encryption schemes is *block ciphers* (Menezes et al., 2001, Ch. 7). They encrypt blocks of data of a specific size (usually 16-32 bytes large depending on the scheme). The block ciphers themselves only define how to encrypt and decrypt single blocks, but complete messages are often longer than one block. There are therefore different *block cipher modes of operation* that defines how to encrypt messages spanning several blocks.

When decrypting, a single bit change in the encrypted data may propagate, affecting a large number of bits in the decrypted data. Therefore, a single bit error during communication affects the whole block (and, depending on the block cipher mode used, could also affect subsequent blocks). A common mode of operation that avoids bit error propagation is *Counter mode* and this is the mode chosen for the prototype (see 4.5).

Counter mode uses the block encryption algorithm as a pseudo-random number generator (PRG) that, with the secret key and a counter as parameters, generate a block of pseudo-random bits (Menezes et al., 2001, pp. 232-233). These bits are then bitwise XOR-ed with the block that is to be encrypted. Decryption is performed by the receiver generating the same pseudo-random block and XOR-ing it with the encrypted data. Since the XOR is performed per bit, a single bit error during transmission only results in a bit error at the same position in the decrypted data. Counter Mode essentially uses the block encryption algorithm as part of a stream cipher.

2.7 Data Integrity

Encryption provides protection against *passive attackers*, who can only view the transmitted data. However, it does not protect against *active attackers*, who can also modify data in transmission (Menezes et al., 2001, Ch. 1). Ensuring *data integrity* protects against such unauthorized modifications.

To provide data integrity when using symmetric ciphers, a *Message Authentication Code* (MAC) is commonly used (Menezes et al., 2001, Ch. 9). The MAC is a function of the message and a shared key and is transmitted along with the message. The receiver, knowing the shared key, produces a MAC of the received message. If the MACs are not equal, either the message or the transmitted MAC must have been modified during transmission.

Active attacks are particularly effective against block cipher modes that do not propagate bit errors (for example counter mode) if data integrity is not ensured. Without data integrity, an attack on a cipher working in counter mode can target and modify specific bits, whereas controlled modifications of the decrypted data are much more difficult when single bit modifications get propagated to many bits in unpredictable ways. It proved difficult to ensure data integrity in the prototype, leaving the protocol vulnerable to active attacks (see 4.5).

2.8 Communication Satellites

Communication satellites communicate with the earth using radio waves of different frequencies and wavelengths, depending on the use case. Satellites can be positioned at different altitudes, which result in different advantages and disadvantages.

2.8.1 Types of Satellite Orbits

Satellites in Geosynchronous Equatorial Orbit (GEO) operate at an altitude of about 36 000 km above the equator and have their orbital speed synchronized with the

Earth's rotation. Because of this, the satellite appears to be fixed in the sky, and this type of orbit is also known as *geostationary.* Due to the long distance to the earth and the constraint of the speed of light, communication with GEO-satellites have one-way delays of around half a second, with a total response time of about 1.5 seconds (Kruse, 1995).

Satellites in Low Earth Orbit, often shortened as *LEO*, are positioned at much lower altitudes with lower latency as a result (Sturza, 1995). The orbits are at different altitudes, but the response time of a typical LEO-satellite is usually around 40 milliseconds. Since they are not in the geostationary orbit, they need to move around the globe at high velocity to stay in orbit.

A single GEO-satellite can cover one-third of the earth, while the LEO-satellites only covers a small part each. As shown in Figure 2.3, the Iridium Constellation, consisting of LEO satellites, requires 66 satellites to cover the entire earth. (Mellow, 2004). Another difference is that satellite networks in GEO often have a higher bandwidth than their LEO counterparts.

2.8.2 The Iridium Network

Iridium is a commercial satellite provider for voice and data that covers the entire globe by its satellite constellation. The users of the network range from military and disaster recovery teams to journalists and sailors. The current generation of the Iridium network was planned in the late eighties and designed for voice calling, which explains the slow bandwidth of 2.4 kbit/s. By the time it was deployed in 1998, the bandwidth was already many times slower than an average wired dial-up connection (Mellow, 2004).

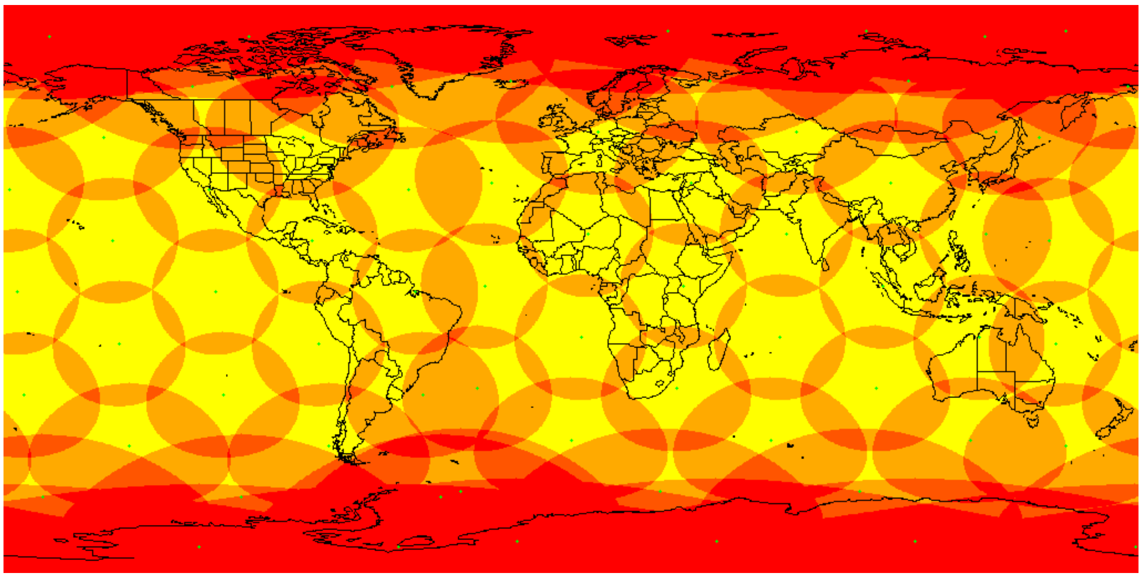


Figure 2.3: Iridium network coverage (GrandDeluxe@Wikipedia, 2011)

The LEO-satellite constellation used by Iridium uses 66 active satellites, distributed in 6 orbital planes. As seen in Figure 2.3, the satellites overlap slightly to cover the entire earth at any given time. The satellites operate at around 781 km, travel at speeds of 27,000 km/h, and communicates with each other using inter-satellite cross-links. The cross-link communication occurs both within the same orbital plane and to nearby satellites in the neighboring planes (Weintrit, 2011, p. 199).

The cross-links are necessary to handle the hand-off to another satellite when the currently connected satellite is about to move out of the user's field of view. (Geoborders Satellite Ltd, 2012). Due to the high velocity, such a hand-off happens regularly; a satellite only stays in view for seven minutes at the equator. If successful, the hand-off is only noticeable as a delay of 1/4 second, however, if no nearby satellite exists or if the signal is interrupted by an obstacle, the connection is dropped (Voyage Adviser, 2015).

Due to the satellite cross-links, the Iridium network works even if the nearby satellite does not have immediate access to a ground station. (Geoborders Satellite Ltd, 2012). In the absence of an available ground station, the signal is simply relayed through the constellation until it reaches a satellite close enough to one. According to an experiment performed by the US Navy (McMahon & Rathburn, 2005), the average total RTT for a packet sent over the Iridium satellite constellation is 1700 milliseconds, and the mode total RTT is about 1350 milliseconds. This is remarkably high for a LEO network.

3
Method

To investigate the possibility of reducing the size of a web page enough to enable effective web browsing on an extremely low bandwidth connection, a prototype was developed. The prototype consists of a client in the form of a web browser and a performance enhancing proxy server. The client accesses the web via the proxy server, which loads the entire requested web page and provides a reduced version back to the client in a stream. The stream allows the client to display the web page as packets arrive, without having to wait for the complete set before presenting it.

The goals of the prototype were to not only be functional, but also reliable, robust and easy to use for the average user. To achieve reliability and robustness, the client had to be able to handle faults during transmission, such as bit errors and temporary connection losses.

For ease of use, the prototype had to be easy to install and run on the common desktop operating systems Windows, Mac OS, and Linux. The need for manual configuration was desired to be minimal. The client interface also had to be intuitive so that no instructions or tutorial would be needed.

The development of the prototype was divided into three separate parts: the browser client, the proxy server, and the communication protocol. The protocol defines how the client and server communicate with each other. Once the protocol was defined, the client and server were planned to be developed in a fairly independent manner. For the implementation, JavaScript was chosen, primarily to allow for the client and the server to be written in the same language, but also so that certain code could be used by both, such as the implementations of compression and encryption.

3.1 Browser Client

A browser client desktop application was implemented using Electron, a framework for using web technology such as JavaScript, HTML, and CSS to develop cross-platform desktop applications. Electron is used by several successful projects, such as Atom and Discord. It was due to these reasons as well as personal preference Electron was used.

As the content, extracted from the proxy server described in 3.2, only consists of plain text and links, the client was implemented to be able to display content in a readable and user-friendly manner. To navigate, the user may enter a specific URL in the browser as well as click the links on the web pages displayed. The client also supports basic web searching, using Google, as well as basic history navigation, such as going back to a previous page. In the beginning, there was also a plan to implement support for general forms, but the feature wasn't prioritized and did not make it into the final prototype.

The resulting prototype is able to be distributed as a standalone desktop application for Windows, Linux and Mac OS. Most computers use one of these operating systems, making this application readily available for most users. No corresponding phone application was developed.

3.2 Proxy Server

A proxy server was developed using Node.js, which is a JavaScript runtime for using the JavaScript language in a server environment, i.e. outside the web browser. The language is well suited for handling web pages, such as modifying the DOM-tree. As the proxy server operates on a high bandwidth connection it can load the full web pages, as requested by the browser client, and utilize content extraction techniques to return only the relevant content to the client. To limit the size of the packets and for simplicity, the content is sent as plain text to the client.

3.3 Network Protocol

A custom network protocol was developed that keeps the communication between client and proxy server at a minimum and can handle the connections typical faults. The protocol had to be able to tolerate bit errors in the page content in such a way that the content is still intelligible in most cases. In case it is not intelligible, it should be possible to retransmit it when requested, either by the user or by an algorithm. It should also allow that content arrives out of order, so it can be shown to the user as soon as possible.

Apart from fault tolerance, a goal was also to have the protocol handle sudden disconnects smoothly. This goal was necessary to be able to provide the user with the possibility to resume communications directly, without the need for any new protocol handshakes.

3.4 Testing

After each big iteration, the solution was tested using an Iridium GO! satellite Internet device to see that it was working in practice as well as with the simulator. At the end of the project, a final field test was performed using the Iridium GO! device. The purpose of the field test was to evaluate how well the resulting solution performs compared to the alternative browser solutions.

It was discovered that Iridium GO! did not let packets containing bit errors through to the application. Thus, to test the performance of the prototype when subjected to bit errors a simple simulator was developed. Apart from simulating bit errors, it also simulates packet loss, temporary connection losses, limited bandwidth, and packet delay, to ensure that the prototype can handle all these faults.

The browsers that were compared to the prototype were XWeb and ELinks via SSH. XWeb by Global Marine Networks is a commercially available web browser intended for satellite Internet. The text web browser ELinks was run remotely on a high bandwidth server and controlled and viewed via SSH. Since the remote access protocol SSH only transmits the currently visible part of the loaded page, similar to how the prototype streams part by part, the goal was to provide a slightly more comparable alternative in terms of time required to see the first part of the content.

Each browser was used to load four different web pages while measuring the total transferred data (if possible), the time until the first content was displayed, and the time until the page was fully loaded. The measurements were done manually using a stopwatch since the time it would take to fully load a page usually would be in the order of minutes. The following pages were tested:

1. A link archive page (Aftonbladet) `http://aftonbladet.se`
2. An article page (Wikipedia) `http://en.wikipedia.org/wiki/Sweden`
3. A company page (Iridium) `http://iridium.com`
4. A small page (Example) `http://example.com`

The web pages were considered representative of common types of web pages that exist. Since content extraction was part of the problem, it was important to test pages with varying amount of media, text, and links. News front pages, here denoted "link archive pages", primarily consist of photos, links, and shorter paragraphs of text. Article pages, such as news or wiki articles, consist of longer text and may include additional figures such as media and tables. Company pages are often lightweight in terms of text but may contain media such as photos and videos. Apart from these commonly visited page types, a small page was included to enable testing in case the ordinary page types were too large for the browsers to load.

4 Evaluation of Technical Solutions

Several possible solutions to each of the five subproblems; content extraction, web searching and forms, network protocol, compression, and encryption, were evaluated. In some cases, existing technologies could be used as part of the solutions, while in other cases a new approach was necessary. This chapter summarizes the key findings of the evaluation and presents the resulting design of the technical solutions.

4.1 Content Extraction

To minimize the amount of data sent to the client, content extraction can be performed on a proxy server, as presented in 3.2. By removing all irrelevant data from a web page and picking out the relevant content, no unnecessary information is sent to the client. Several techniques for delivering only relevant content are available, but most focus on removing clutter like pop up ads or images to simply make a web page easier and faster to read (Rahman & Hartono, 2001) (Gupta, Kaiser, Neistadt, & Grimm, 2003). This differs from the goal of reducing the web page in terms of bytes. However, since these algorithms by removing clutter also reduce the size of a web page, the same techniques can be used in this project. The most common example of this type of content extraction is the different versions of *Reader Modes*, as discussed in 4.1.1, available in most modern browsers.

Another approach to content extraction is to evaluate each HTML-element and try to decide if it is part of the content or irrelevant data based on a number of criteria. This approach was attempted in the prototype but with unsatisfying results. Another way to achieve content extraction could be using algorithms utilizing machine learning (Chau & Chen, 2008) (Yunis, 2016). Due to the complexity of machine learning, this was not evaluated further. Text-based web browsers like w3m and ELinks can also be used to remove unnecessary information since they convert the HTML into plain text, thus removing multimedia and HTML overhead, while still trying to display the web page's intended structure.

4.1.1 Reader Mode

The reader mode provided in many browsers removes clutter from an article when used, showing only the actual text and images related to the article. Safari, Microsoft Edge, and Mozilla Firefox all provide this feature (Microsoft Corporation, n.d.) (Apple Inc., n.d.) (Mozilla Corporation, n.d.). Out of these only Firefox uses an open source library for their content extraction, called Readability.

The Readability algorithm uses a scoring system that identifies the primary text content of a web page (Mozilla Organization, 2010). For example, HTML elements such as "article" and "post" are given a positive score, while "comment" and "footer" are given a negative score. The algorithm finally selects the element with the highest score, but also requires that the content consists of more than 500 characters in order to provide any result at all (Mozilla Organization, 2010). Hence, the algorithm turned out to work well on articles, but not on pages that consist of several shorter pieces of text. For example, a website's main page consisting of multiple links leading to other content.

Since Readability works well for articles but not general web pages, an option is to use Readability in cases where the content is an article. Long texts can be extracted into a well-structured format, removing everything except the actual content while still allowing regular web browsing using another method. Although Readability was found to be unsuitable as a general solution, and thus not implemented in the prototype, the same approach used to extract an article can be used to extract other content as well.

4.1.2 Content Scoring

Based on the Mozilla Readability library a custom algorithm was developed. By creating a DOM-tree of the specified web page's HTML and traversing it using depth-first search the nodes were scored based on a number of different criteria. Some of these include link density, text length, how high up in the hierarchy a specific node is and the score of the node's children. Based on the content of a node three things can happen: the node and its children are removed, the score of the node is increased, or the score of the node is decreased. After evaluating every criterion the node's score is compared to a predefined threshold, and if the score is below that threshold it is removed. After evaluating the entire DOM-tree the HTML is rendered into text.

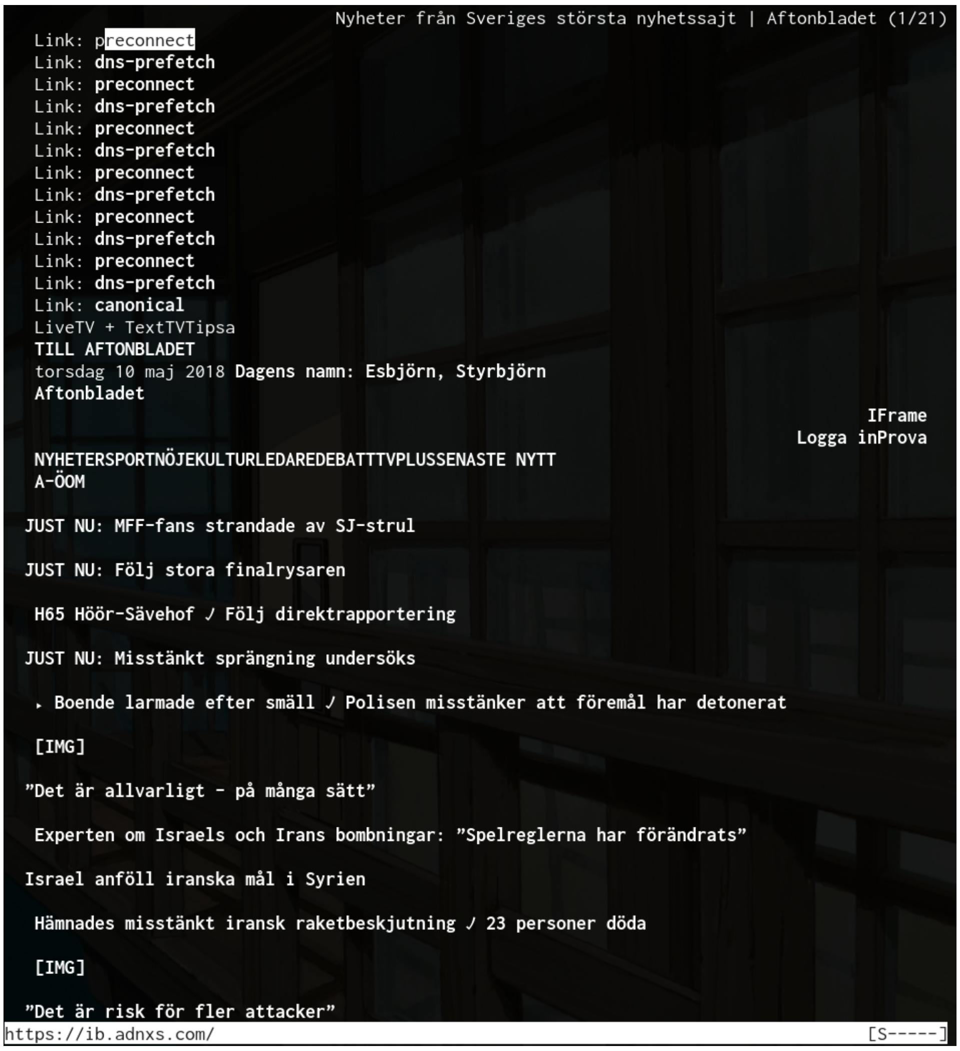
The algorithm was able to reduce web pages to a level where the content could be transferred over low bandwidth in a reasonable time. However, without a proper HTML-renderer, the content was not formatted particularly well, especially for lists and tables.

4.1.3 Text-Based Web Browsers

To maintain the structure of a web page after converting HTML into plain text the text-based web browsers w3m and ELinks were evaluated. Text-based web browsers can be used when a graphical user interface is unavailable or when rendering a web page quickly is preferred over visual style. Both w3m and ELinks support a headless mode that reads an HTML file or URL and outputs the content as plain text, as opposed to running them interactively (Ito, n.d.) (Elinks, 2012).

A problem with rendering HTML into plain text using these browsers is that there are no clickable links. Instead, both w3m and ELinks print all links at the bottom of the text output. The w3m browser was initially chosen and modified to provide a separate list of links as well as their start and stop character positions. The links and their text indexes were then used to add the links back to the text. However, an issue occurred due to what seemed to be a mismatch between character encoding in w3m and the Node.js content extraction code, which caused the character positions to be slightly off in many situations. By performing equivalent changes to ELinks the same result was achieved without these mismatches.

Another positive effect of using text-based web browsers is that they reduce the size of a web page significantly on their own. Even to the degree that the resulting content can be transferred over low bandwidth within a reasonable time. Running the algorithm mentioned in 4.1.2 before rendering the HTML using ELinks was tested, but the difference in page size between that and only running ELinks was in most cases negligible.



```
                              Nyheter från Sveriges största nyhetssajt | Aftonbladet (1/21)
  Link: preconnect
  Link: dns-prefetch
  Link: preconnect
  Link: dns-prefetch
  Link: preconnect
  Link: dns-prefetch
  Link: preconnect
  Link: dns-prefetch
  Link: preconnect
  Link: dns-prefetch
  Link: preconnect
  Link: dns-prefetch
  Link: canonical
  LiveTV + TextTVTipsa
  TILL AFTONBLADET
  torsdag 10 maj 2018 Dagens namn: Esbjörn, Styrbjörn
  Aftonbladet
                                                                              IFrame
                                                                       Logga inProva

  NYHETERSPORTNÖJEKULTURLEDAREDEBATTTVPLUSSENASTE NYTT
  A-ÖOM

 JUST NU: MFF-fans strandade av SJ-strul

 JUST NU: Följ stora finalrysaren

  H65 Höör-Sävehof ✓ Följ direktrapportering

 JUST NU: Misstänkt sprängning undersöks

  ‣ Boende larmade efter smäll ✓ Polisen misstänker att föremål har detonerat

  [IMG]

 ”Det är allvarligt - på många sätt”

  Experten om Israels och Irans bombningar: ”Spelreglerna har förändrats”

 Israel anföll iranska mål i Syrien

  Hämnades misstänkt iransk raketbeskjutning ✓ 23 personer döda

  [IMG]

 ”Det är risk för fler attacker”
https://ib.adnxs.com/                                                            [S-----]
```

Figure 4.1: The text-based web browser ELinks

4.1.4 Link Extraction

Links between web pages are an integral part of the web, and on a regular web page, there can be thousands of links. By removing the links and only storing an ID corresponding to each link, the size of a web page is reduced significantly. The links are stored on the server, and when the client requests an ID the corresponding web page is loaded.

Since the actual link never reaches the client, the user will not know the link destination before clicking it. This might be a problem for some users, but the impact was considered small. Because of the large reduction in size and low user impact, link extraction was considered viable for the prototype.

4.2 Web Searching and Forms

Optimally, full forms should be supported by a web browser to enable features such as searching, payments, and registration. The arguably most elementary use case for forms is to be able to use search engines. Hence, a specific solution to basic searching was prioritized, before evaluating general form support.

4.2.1 Web Searching

To perform a search query on Google one would normally fill in the search field and press submit. This is not possible when the extracted content only contains plain text and not any HTML elements since the search field will be removed as well. Instead, one could utilize the fact that the Google search result page is accessible via a URL that contains the search query. Hence, by navigating directly to the Google search result URL, such as *google.com/search?q=example* a search query can be performed despite the lack of general HTML form support. By letting the browser client provide a search field that constructs and requests the URL of the search a user is able to perform searches without support for general forms.

4.2.2 General Forms Support

Support for general forms was evaluated but determined to be of low priority and was thus never implemented. However, it was concluded that to enable general forms only a few functions are required.

Since ELinks remove form information while rendering HTML as plain text, ELinks would have to be modified to output form information the same way as link information. Forms are a bit more complex than links and contain many elements of different types with different properties. For example, all form elements belong to a `<form>`, and certain elements, such as `<select>`, have subelements, such as `<option>`. As with links, the server should replace the names with ID:s. It should then encode the form information into plain text, similar to how the links are encoded (see 5.3.3).

After receiving the content, the client would have to decode the form information and present the form to the user. Whenever the user submits a form, a request for a URL containing the form identifier and values should be made. The server, having saved the form information not sent to the client, could then submit the form to the target web page.

Arguably, apart from search fields, log in and checkout forms are the most common use cases and these also require cookies to function. Hence, to support most forms, cookie handling would have to be implemented on the server as well. Support for cookies is possible but was not implemented due to time constraints. The plan was to have the server save any cookies the web page returns in the current session and then submit them whenever a request is made to a web page for which it has cookies.

4.3 Network Protocol

Two types of data are transported between the proxy server and the client: the content of a web page and commands to control the transmission of the content (for example requesting a web page or aborting the transmission of a web page). The transport layer was therefore split into two different logical channels, a data channel and a control channel, each requiring a different set of services from the transport layer protocol.

The content of a web page should be displayed even if it arrives out-of-order or is corrupted during transmission. Therefore, it is not required that the protocol provides in-order delivery or that it automatically discards corrupt packets. Neither should it try to retransmit lost packets uncontrollably since this can result in the application halting. UDP fits these requirements if the optional discarding of corrupt packets is disabled (Postel, 1980).

An ordinary UDP packet cannot hold a sequence number or similar information. This is required to reassemble a web page in the client. A simple *Data Transport Protocol* (DTP), described in Section 4.3.1, was developed to run on top of UDP and provide this service.

The control channel should provide in-order delivery and retransmit lost or corrupted packets. TCP is typically used for this (see 2.5.1), but it was found to be ill-suited for a high latency, low bandwidth connection (see 4.3.2). Because of this, the alternative protocols RUDP, ENet, and KCP were evaluated. One of them, KCP, was used in the prototype for a time (see 4.3.3). Eventually a new protocol, *Control Transport Protocol* (CTP), was developed (see 4.3.4), since none of the alternatives were found satisfying (see Section 4.3.3).

4.3.1 Data Transport Protocol

The Data Transport Protocol transfers the content of a web page as a sequence of packets. The content is displayed to the user at arrival and in its correct place even if it arrives out-of-order. To achieve this each packet contains a *sequence number* that indicates where in the sequence of packets it belongs. The bits used to encode the sequence numbers must be large enough to ensure that numbers are not reused until a reasonable amount of time has passed.

Depending on how the sequence numbers are implemented, packets belonging to one web page might be mistaken for packets belonging to another. For example, if sequence numbers are reset to 0 at each page request and the user requests page B while another page A is still being transmitted, packets in transmission belonging to A will be assumed to belong to B and the page will be garbled.

To prevent this some way of identifying old, invalid packets is required. One solution is to transmit a page number alongside the sequence number that is changed between each request. Another way is to assign a new set of sequence numbers to each request. For DTP, the second alternative was chosen as it uses the limited bits more efficiently.

The intelligibility of a web page can be impaired if the sequence number is corrupted during transmission and a block of characters ends up in the wrong place. Therefore an error-detecting code is required for the sequence number. If the code is invalid, the packet is discarded.

Another error-detecting code is required for the page content of the packet. This code is used to notify the receiver when the payload is faulty. DTP does not automatically retransmit packets since this could be a bad use of the limited bandwidth; the content might still be fully legible, or irrelevant to the user. Instead, what to retransmit is selected by the user, or by a higher level automatic retransmitting scheme (see Section 5.1.2).
In summary, a DTP packet consists of the following parts:

- A sequence number
- An error-detecting code for the sequence number
- The payload (web page content)
- An error-detecting code for the payload

4.3.2 Standard TCP (New Reno)

As mentioned in Section 2.5.1, one core component of standard TCP is congestion control. Congestion control is useful when a connection is shared by several devices or the bandwidth of a connection varies. Since an Iridium GO! connection is used by only one device and the bandwidth is fixed, congestion control is unnecessary and might even be detrimental since all packet loss is assumed to be a result of congestion.

Satellite links have high latency and therefore high RTT (Caini & Firrincieli, 2006). On connections with very low bandwidth, TCP may queue up too much data, and this might result in an even higher RTT. Packet loss in TCP always results in delay, and a high RTT increases the resulting RTO which increases the delay even further. However, the control channel should only transmit a small amount of data and therefore should not queue up any data under normal circumstances.

In TCP, the parameters for congestion control and RTT-algorithms are fixed. Hence it is hard to adapt the networking protocol to work better on high latency low bandwidth connections. The absence of options made TCP undesirable for the application, as the efficiency could not be improved.

4.3.3 Alternatives to TCP

RUDP (Bova & Krivoruchka, 1999), KCP (skywind3000, 2018) and ENet (*ENet: Enet*, 2015) all provide reliable in-order delivery. They run on top of UDP and offer a higher degree of customization than TCP. By adjusting parameters for RTT and congestion control for a specific connection, some of the typical issues with TCP can be mitigated.

Available implementations of RUDP either requires heavy modification (klueska, 2014), or is in an early stage of development (shovon, 2015), and would require much work to make viable for this project. Both ENet (Naamani, 2015) and KCP (leenjewel, 2018) has libraries for Node.js ready for use. The ENet code base is larger than KCP and thus harder to modify and extend. KCP was the best alternative, but a bug was encountered in the Node.js package, where some packets were not delivered correctly.

4.3.4 Control Transport Protocol

The Control Transport Protocol is simple and provides reliable, in-order delivery of packets. It uses a *selective repeat* process to retransmit corrupt or missing packets, sending an acknowledgment for every packet received and retransmitting packets not acknowledged within a fixed time. Sequence numbers are used to identify packets. An error-detecting code is used to determine if a packet is corrupt.

Sequence numbers are maintained by a *sliding window*. An example of a sliding window is shown in Figure 4.2a. Here there are 8 possible sequence numbers, 0-7, and the window has a size of 4 sequence numbers. Blue numbers correspond to sent and acknowledged packets, orange to sent but unacknowledged, and gray to unused sequence numbers. Only the numbers inside the window are available when transmitting packets. At this point no numbers are available and no further packets can be sent. In Figure 4.2b the first packet in the window is acknowledged (in this case number 1), the window is moved forward, and a packet can be sent with sequence number 5.

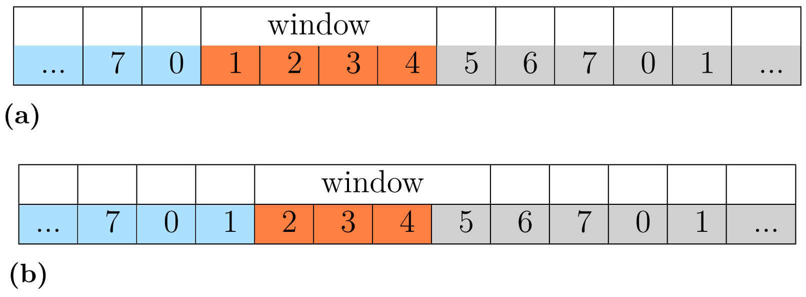


Figure 4.2: Example of a sliding window

Since numbers are reused when using a sliding window, the sequence numbers can be encoded using only a few bits, resulting in less overhead. The window should be kept small since few packets are sent over the control channel. Also, keeping the window small ensures that few packets are being retransmitted simultaneously so that packets are delivered in a more timely manner. On the other hand, the window should be kept large enough that CTP will not have to wait for acknowledgments in normal circumstances.

In summary, a CTP packet has the following components:
- A flag indicating if the packet is an acknowledgment
- A sequence number
- The payload
- An error-detecting code for the whole packet

4.4 Compression

Compression would allow faster loading of web pages, but in the context of streaming over a connection potentially prone to bit errors, a viable compression algorithm must meet certain criteria:

1. It should support UTF-8 so that all characters used on the web can be represented.
2. It should not be sensitive to bit errors. A bit error should affect as few characters as possible when decompressed. In the end, this criterion is measured as the resulting readability of the decompressed data and is difficult to quantify.
3. It should not require any data to be sent reliably. This excludes compression algorithms that create a dictionary that has to be transferred correctly before decompression can begin, which could incur long delays, especially if it has to be retransmitted.
4. It should allow random access, i.e. it should be possible to decompress data that arrives out-of-order so it can be shown to the user as soon as possible.

4.4.1 Common Compression Algorithms

Ordinary web servers and web browsers already make use of compression, commonly using the Gzip algorithm (McAnlis, 2013). Gzip uses the DEFLATE compression algorithm, which in turn uses a combination of LZ77 (Salomon & Motta, 2010, pp. 334-339) and Huffman coding (Salomon & Motta, 2010, pp. 214-234). Because of the properties of LZ77 and Huffman coding, Gzip cannot be directly applied in this context.

LZ77 replaces repeated sequences of characters with references to the first occurrence of that sequence (Salomon & Motta, 2010, pp. 334-339). This means that data can only be decoded sequentially, breaking criterion 4. Bit errors could produce several kinds of faults. If an original sequence is affected, that corruption will be reproduced by all references to it. If a reference is affected, it could reference a wrong sequence, change length, or become invalid. If a reference that is often referenced is affected, a single bit error could affect a huge number of characters. In essence, a bit error can be bad enough that LZ77 arguably breaks criterion 2.

Huffman coding uses a dictionary, a Huffman tree, when decompressing, breaking criterion 3 (Salomon & Motta, 2010, pp. 214-234). The dictionary can however be pre-computed on some representative data set, meeting criterion 3 at the cost of a lower compression ratio. The Huffman codes are of variable length, but if care is taken not to break them when splitting the compressed data into packets, criterion 4 can be met. Bit errors can change one Huffman code into one of another length, breaking the synchronization of the codes, and potentially rendering a whole packet unreadable, breaking criterion 2. Resynchronizing Huffman Codes (Salomon & Motta, 2010, pp. 190-193) would meet criterion 2 at the cost of a lower compression ratio, but its use has not been investigated further.

4.4.2 Byte Pair Encoding for UTF-8

BPE generally works by replacing the most commonly occurring pairs of bytes with unused byte values (Salomon & Motta, 2010, pp. 424-247). Fault tolerance is achieved, according to criterion 2; if one or more bytes are corrupted, all other bytes can still be correctly recovered.

General BPE creates a dictionary based on the compressed data, mapping the unused byte values of that data to the most frequently occurring pairs. To meet criterion 3, not requiring the reliable transmission of a dictionary, a variant of BPE was developed. The BPE variant meets criterion 3 by using pre-computed dictionaries that are stored in the server and client. Thus, the BPE variant can only use byte values that are never used in plain UTF-8 text, and meets criterion 1 by using invalid UTF-8 sequences and ASCII control codes.

The BPE algorithms proposed by Robert & Nadarajan are designed to be used with ASCII encoded text only (Robert & Nadarajan, 2009). ASCII uses 7 bits but encodes characters using a whole byte, always leaving the most significant bit 0, which means that 128 byte values are not used in ASCII. Additionally, ASCII contains a number of unprintable control codes (values 0-9, 11-31 and 127) that are not used in normal text documents. All these unused values can be used in BPE. UTF-8 is compatible with ASCII but uses the most significant bit of the first byte to be able to encode many more characters than ASCII. Thus, to use BPE for UTF-8, some unused values in a UTF-8 sequence had to be found.

Table 4.1 shows the possible values of a valid UTF-8 character, where x denotes an arbitrary value (The Unicode Consortium, 2017, pp. 125-127). Any value not appearing in the table is invalid. Thus, the first byte of a valid UTF-8 character will never have a value of `10xxxxxx` or `11111xxx`. Also, the 32 unprintable ASCII control codes are included in UTF-8 for backward compatibility. Therefore, there are a total of 104 unused values, of which 102 can be used to encode frequently occurring character pairs (the last 2 being used to encode links, see Section 5.3.3). Since there are only 102 values usable by BPE compared to the 160 values of ASCII, compression will be slightly less efficient for UTF-8.

First byte	**Second byte**	**Third byte**	**Fourth byte**
`0xxxxxxx`			
`110xxxxx`	`10xxxxxx`		
`1110xxxx`	`10xxxxxx`	`10xxxxxx`	
`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

Table 4.1: Valid UTF-8 Byte Sequences

BPE requires a dictionary mapping the available unused byte values to the most frequently occurring character pairs. For optimal compression, a unique dictionary should be constructed for each text document. However, as mentioned in Section 4.4, having to transfer a dictionary before decompression can begin is not an option. Therefore, dictionaries are pre-computed using a selection of representative web pages. The web pages are first converted from HTML to plain text using content extraction so that any artifacts generated by the conversion (e.g. pairs of spaces) are also compressed. Since character sequence frequencies can differ a lot between languages, a dictionary could be created for every language to improve the compression ratio.

The BPE algorithm runs recursively, meaning that when a pair has been encoded as a single byte, that byte can be part of another pair that is encoded as a byte. For example, *th* could be encoded as the byte A, and then Ae encoded as B, so that the whole word *the* is encoded as B. Since only invalid UTF-8 characters are used, the number of recursions do not need to be known beforehand – the recursion stops when there are no more character combinations to replace.

The algorithm is more prone to bit errors than plain text, but still quite tolerant. A bit error in a non-compressed byte will only affect one character, however, a bit error in a compressed byte will affect the characters that it replaced. Hence, if a bit error occurs while transferring B, the whole word *the* will be affected when decompressed.

4.5 Encryption

Under the conditions established in Section 4.3, there are two unusual criteria for a possible encryption scheme:

1. The mode of operation should allow for random access. That is, any packet should be decipherable without knowing any other packet, since packets may arrive out of order.
2. Bit errors should not propagate during decryption to affect more than one bit.

Most block encryption schemes were designed without such requirements in mind. They assume that the incoming data will be decrypted sequentially, hence not considering criterion 1. Similarly, criterion 2 was not considered as it assumes that data integrity is ensured so that bit errors never reaches the decryption stage. Of the four most common block cipher modes (ECB, CBC, CFB, and OFB), only a variant of OFB, Counter Mode (Menezes et al., 2001, p. 233), fulfills both requirements (see Section 2.6). Thus, counter mode was chosen for the prototype. It is essentially a type of stream cipher that uses a block encryption algorithm as a keystream generator. Since stream ciphers generally have low error propagation (Menezes et al., 2001, p. 191), other stream ciphers can presumably be used instead of counter mode. An additional benefit of using counter mode, or other stream ciphers, is that they do not introduce any bandwidth overhead since padding is not required.

However, it seems impossible to distinguish a bit error during transmission from an intentional modification, since the receiver can tell only that a modification has occurred but not why it occurred. The problem of providing data integrity when bit errors are to be tolerated was deemed to be outside of the project's scope. Not ensuring data integrity leaves the system open to active attacks against the encryption scheme (see Section 2.7), but providing security against passive attacks is better than providing security against neither passive nor active attacks.

5 Implementation of the Prototype

The prototype consists of a proxy server, a custom protocol, and a browser client. Connections are initiated by the client transmitting an initiation packet to the server. The server creates a new connection and responds with the connection ID. Encryption key exchange is performed at the beginning of the connection, also initiated by the client.

When the client requests a page, the page is downloaded by the proxy server. The server attempts to remove any unnecessary elements from the page, such as forms and images. After this initial reduction, links are extracted and the remaining HTML is rendered into plain text. Finally, the page is streamed to the client from top to bottom.

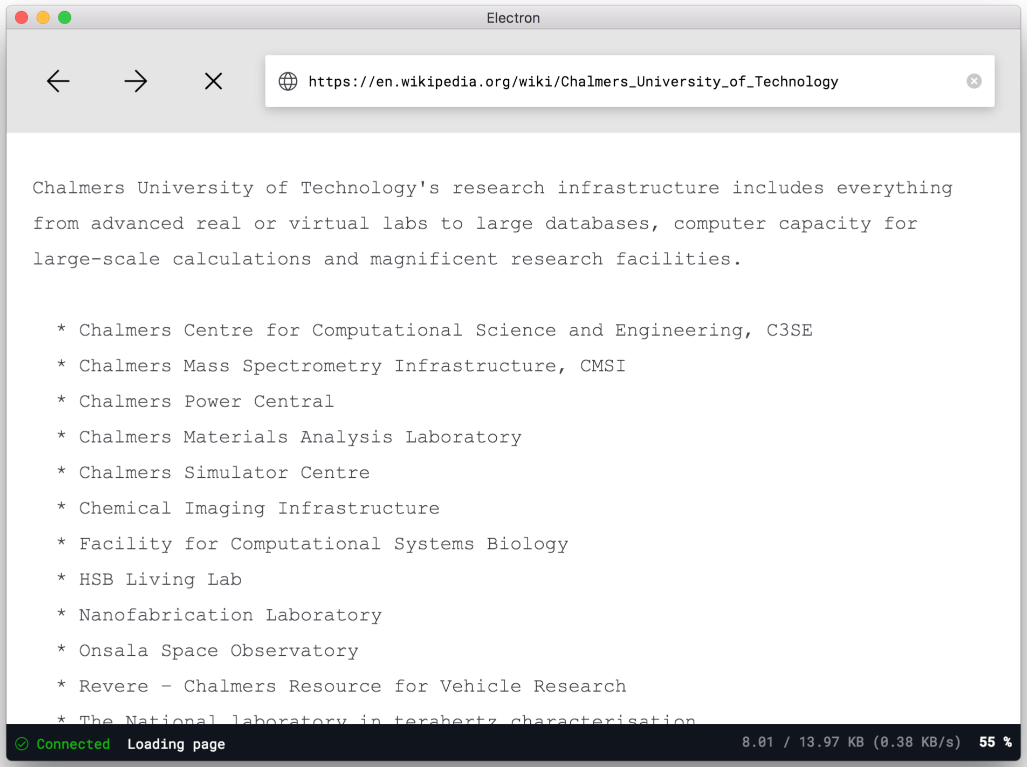
The protocol suite contains two transport layer protocols used for transmission of data. DTP (presented in 4.3.1) is unreliable, while CTP (presented in 4.3.4) is a reliable in-order protocol. A third protocol, Connection Protocol, is used only for connection management. Lastly, two application layer protocols are used, Data Channel Protocol to transmit page content using DTP, and Control Channel Protocol to exchange encryption keys and control transmission using CTP.

5.1 Browser Client

The browser client was implemented as a standalone desktop application for Windows, macOS, and Linux. The client was developed using Electron, which allows access to certain operating system API:s, such as low-level networking and file system access. As Electron uses web technology such as HTML/CSS for layout and JavaScript for business logic, the browser client can be compiled for all three platforms at once with minimal additional effort.

The JavaScript framework React was used to make dynamic modification of the HTML elements easier, as opposed to using the standard JavaScript methods for modifying HTML elements. Instead of plain CSS, Sass was used to provide a better development experience. Sass is a superset of CSS that provides some additional convenient features, such as variables.

5.1.1 Graphical User Interface



https://en.wikipedia.org/wiki/Chalmers_University_of_Technology

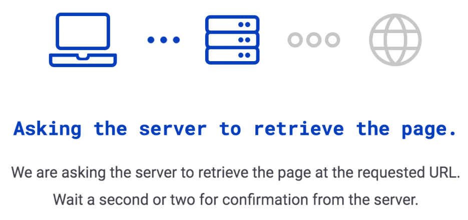
```
Chalmers University of Technology's research infrastructure includes everything
from advanced real or virtual labs to large databases, computer capacity for
large-scale calculations and magnificent research facilities.

  * Chalmers Centre for Computational Science and Engineering, C3SE
  * Chalmers Mass Spectrometry Infrastructure, CMSI
  * Chalmers Power Central
  * Chalmers Materials Analysis Laboratory
  * Chalmers Simulator Centre
  * Chemical Imaging Infrastructure
  * Facility for Computational Systems Biology
  * HSB Living Lab
  * Nanofabrication Laboratory
  * Onsala Space Observatory
  * Revere - Chalmers Resource for Vehicle Research
  * The National laboratory in terahertz characterisation
```

Figure 5.1: The browser client running on Mac OS.

As seen in Figure 5.1, the resulting client is minimalist compared to modern web browsers. The browser provides a simple navigation bar with an address field and a stop button that allows the user to stop the current page from loading. Next to the stop button, there are two history navigation buttons that allow the user to return to previously visited websites. In the bottom, there is a status bar that shows the connection status as well as information about progress and download speed.

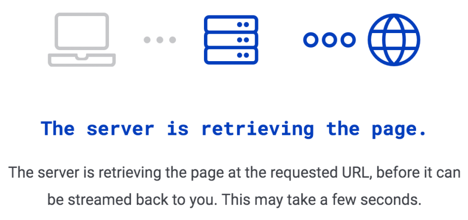
On a low bandwidth satellite Internet connection, it is especially important that the status of the application is clearly communicated to the user. If not communicated properly, the user might think that there is an issue with the connection when the application seemingly gets stuck for a period of time. Apart from the previously mentioned status bar, some design features were implemented to make the application appear faster and more responsive; informative loading messages and content placeholders that indicate the length of the document.

The informative loading messages, as seen in Figure 5.2, appear when the user starts navigating to a web address. Firstly, the client will explain that it is waiting for the proxy server to acknowledge the navigation request. Once the acknowledgment


Asking the server to retrieve the page.

We are asking the server to retrieve the page at the requested URL. Wait a second or two for confirmation from the server.

(a) Waiting for communication between the client and the proxy server.


The server is retrieving the page.

The server is retrieving the page at the requested URL, before it can be streamed back to you. This may take a few seconds.

(b) Waiting for communication between the proxy server and the website.

Figure 5.2: Loading messages in the browser client.

is received, the second loading message will be displayed, which explains that the proxy server is currently downloading and processing the web page.

Once the web page has been downloaded and processed by the server, the content is split into data packets that are sent to the client. The client is given information about the number of packets that will be sent before the packets start arriving, and displays an animated content placeholder for each packet that is expected to arrive. As the packets arrive, the placeholders are replaced by the actual content.

5.1.2 Retransmission of Corrupt or Missing Data

A benefit of using placeholders is that if a packet is lost in transmission, the placeholder will remain visible. If a packet n has not yet arrived, but the previous $(n-1)$ and the following $(n+1)$ have, the packet placeholder will indicate that the packet is potentially lost. In that case, the user can choose to reload the lost packet by clicking a button next to the message, as seen in Figure 5.3.

Determining if a particular packet is lost is not obvious; the server only knows that the packet was sent, and the client only knows that it is expected to receive the packet. Since the packets come in unpredictable order and may arrive late, the client can only make a qualified guess on whether the packet is actually lost. A potential problem with the decided upon solution is that, while not evident during our testing, several consequent packets could be lost, which would cause the placeholders to remain in a state where they are still waiting for the packets. While arguably sufficient for a client prototype, a more robust solution would be desirable to improve the resilience to networking issues.

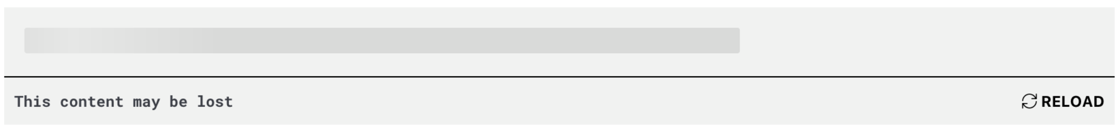
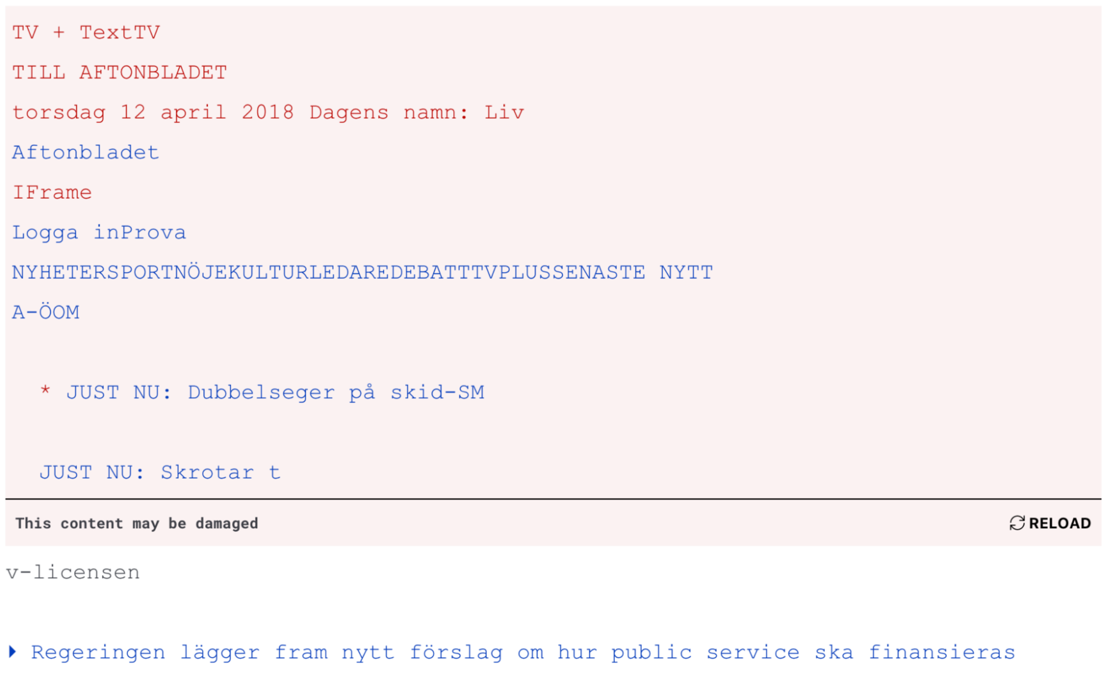


Figure 5.3: The content placeholder of a potentially lost packet.

Similarly, if any content is damaged in transmission due to bit errors, the client will indicate that to the user. As retransmitting a packet takes time on low bandwidth, the content is not automatically retransmitted. Instead, the user has the option to reload the packet if the damaged content is not readable enough. As seen in Figure 5.4, the damaged content is tinted red and presented with an informative message.



```
TV + TextTV
TILL AFTONBLADET
torsdag 12 april 2018 Dagens namn: Liv
Aftonbladet
IFrame
Logga inProva
NYHETERSPORTNÖJEKULTURLEDAREDEBATTTVPLUSSENASTE NYTT
A-ÖOM

  * JUST NU: Dubbelseger på skid-SM

  JUST NU: Skrotar t
This content may be damaged                                    RELOAD
v-licensen

 ▸ Regeringen lägger fram nytt förslag om hur public service ska finansieras
```

Figure 5.4: An example of a damaged packet followed by a correct packet.

Apart from the manual retransmission mechanics, there is also a simple automatic retransmission logic in place. The client will automatically request for certain packets to be retransmitted if it has not received web page content from the server for a long time, despite being able to receive other messages from the server (For example Ping messages, see Section 5.3.1.1). Missing packets are prioritized over corrupt packets, but both are retransmitted automatically. This ensures that the entire web page is loaded if given enough time.

Together, the manual and automatic retransmission mechanics cover two use cases; the user can be active in choosing what parts of the web page are important and should be streamed in first, or the user can be passive and simply wait until the whole web page is loaded. Also, if there is a large number of packets missing, making manual retransmission tedious, for example, because of a longer connection loss, the whole page will nevertheless be downloaded in time. The retransmission mechanics could be much improved, but they are enough to prove that the prototype works.

5.1.3 Web Searching and Forms

Similar to many desktop web browsers, one can enter search queries into the address field, as seen in Figure 5.5. Visually, the address field will display a search icon if the entered text is not a URL. The browser client interprets any address field value that does not appear to be an URL as a search query, which then requests the equivalent Google search result URL. The detection is done by the following naive algorithm:

1. If the value does not contain any period, it is not an URL.
2. If the value contains a space, it is not an URL.
3. Otherwise, it is an URL.

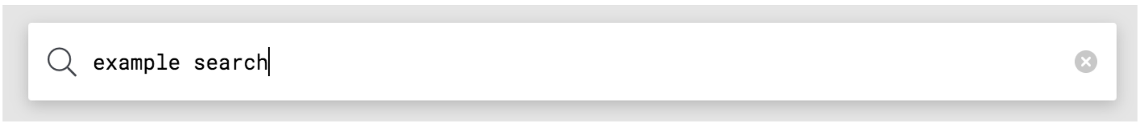


Figure 5.5: The address field with a search query entered.

5.1.4 Caching

For large sites, there is a significant delay to access content that is further down the page. As a way to combat the loading times, a caching solution was implemented.

After a page is fully loaded, the site content is saved on the user's computer. The next time a user requests the same site, the old content will be immediately loaded from the cache. The user is notified that the current content is loaded from cache. The up to date version of the site is loaded in the background as usual, and the user is free to switch between the two modes at any time.

A major problem with the cache solution is that the cache never expires and therefore the page might be outdated. This is mainly a problem with sites that frequently updates, such as news sites.

5.2 Proxy Server

The proxy server was mainly developed using Node.js and is able to run on both Linux, Windows, and macOS. For the purpose of testing, the proxy was hosted by a virtual server in a data center in London operated by DigitalOcean running Linux. The server is set up to be hosted privately with a low number of users. However, it could be expanded and set up as a commercial product, hosted by one company allowing multiple users to connect.

The server retrieves a web page when requested by the client. When retrieving the web page over HTTP, it uses the *User-Agent* header field to tell the server that it is a mobile phone in order to get the mobile version of the web page if available, since it usually includes less irrelevant content. However, after content extraction, the difference in size between an ordinary and a mobile web page was found to be quite small in most cases.

The server then performs three distinct steps to reduce the content of a web page. These are: content filtering, link extraction, and rendering, and the flow can be seen in Figure 5.6. This is handled by modules that are easy to add and remove, allowing the extraction to be expanded relatively easily. Another possible improvement is to allow a user to decide which steps to include, for example in situations where bandwidth is not as limited, allowing more data to be sent.

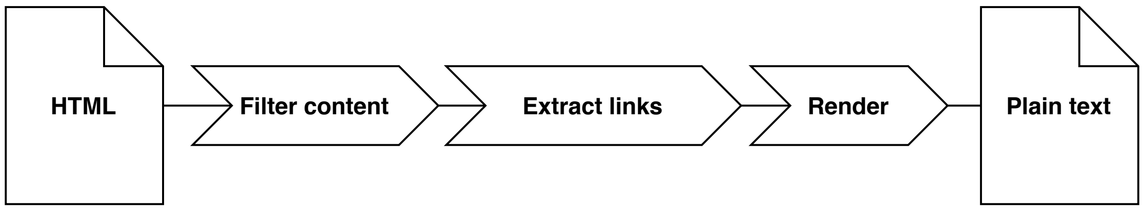


Figure 5.6: Extraction flow chart

5.2.1 Content Filtering

Before rendering the HTML into plain text, elements are removed from the DOM-tree based on their name. Three different categories are removed outright; styling and scripts, forms, and non-textual media such as video and pictures. Each category has a set of predefined elements which are removed and never reach the renderer, reducing the content size beforehand.

5.2.2 Link Extraction

To reduce the content sent to the server further all links are replaced with an ID before being rendered. The full link is saved on the server while just the ID is sent to the client. When a user clicks on a link in the client, a request is sent to the server for the URL that corresponds to that ID. This leads to significant reductions in size for web pages that have a high concentration of links, allowing for faster load times.

5.2.3 Rendering HTML

Using ELinks the size of a web page is reduced significantly while still keeping the general structure of the original web page. A lot of information is removed, such as images, scripting, forms, and video, but these are not included in the scope of this project. The resulting web page is small and well structured and can be sent to the client relatively fast.

5.3 Network Protocol

Most of the network protocol is implemented as a shared Node.js module between the proxy server and browser client, which allowed for easy code reuse. All integers in the protocol are encoded in *big-endian* format, which means that the most significant bytes are sent before the lesser significant bytes.

5.3.1 Transport Layer

The transport layer is split into two different protocols that run on top of UDP to transfer data (see Section 4.3). Early testing using the Iridium GO! device made evident that the external UDP port is given by Iridium's *Network Address Translation* (NAT) would change if the Internet connection was broken for a long enough period. A connection that must endure long connection losses can therefore not be identified by an IP address and a port. It was also found that, if the connection is made through a NAT gateway, the client needs to regularly send UDP packets (keep-alive messages) to the server to keep the gateway from closing the UDP port. Hence, a simple connection protocol (described in 5.3.1.1) was added to handle connections and send keep-alive messages, as well as multiplex DTP and CTP over a single UDP port.

A bandwidth limiter was also added to ensure that the bandwidth of the network is never exceeded. This either averts packet loss if packets that exceed the bandwidth are dropped or avoids increased response time if excessive packets are queued in the network. The limiter queues packets in memory if they are expected to exceed the bandwidth, and can be removed before being sent if requested by the application. The limiter also has a concept of priority, so packets in the control channel have a higher priority than packets in the data channel, and packets requested to be resent by the user have higher priority than ordinary data packets.

5.3.1.1 Connection Protocol

The format of a message for the connection protocol is shown in Table 5.1. *CRC8* is calculated over *connection ID* and *type*. Messages with invalid *CRC8* are discarded.

bit	0	1	2	3	4	5	6	7
0	connection ID							
8	connection ID						type	
16	CRC8							
24 ...	payload							

Table 5.1: Connection protocol format

The possible connection protocol message types and their bit codes are shown in Table 5.2. The *connection ID* field contains the ID of the connection the message belongs to. The message type determines what to do with the payload, if there is any. The payload is of variable length and is assumed to be the remainder of the data after the header.

00	**INIT**	Used for establishing new connections.
01	**CTRL**	Message for control channel.
10	**DATA**	Message for data channel.
11	**PING**	Used to send keep alive messages.

Table 5.2: Connection protocol message types

The payload of an **INIT** message is ignored. When the message is initially sent by the client, the *connection ID* field is also ignored. The server creates a new connection and responds with an **INIT** message with the *connection ID* set to the ID of the new connection.

The payload of a **CTRL** message should be passed to the control channel of the connection. The payload of a **DATA** message should be passed to the data channel. The payload of a **PING** message is ignored. Whenever the client sends a **PING** message to the server, it responds with a **PING** to the client.

5.3.1.2 Data Transport Protocol

The format of a DTP packet is displayed in Table 5.3. *Checksum* is a modular sum calculated over *sequence number*. *CRC16* is calculated over *payload*. The payload length is determined by the packet length.

bit	0	1	2	3	4	5	6	7
0 8	sequence number							
16	seq. num.				checksum			
24 ... n-24	payload							
n-16 n-8	CRC16							

Table 5.3: Data Transport Protocol format

A new interval of sequence numbers is assigned to each new page to be transmitted over DTP, to ensure that packets for different pages are not mixed up (see 4.3.1). Even if a page is not fully transmitted, the sequence numbers are considered used. For example, if page A is assigned sequence numbers x to y, then page B is assigned *y+1* to z. Sequence numbers are allowed to wrap back to 0, so if S is the maximal sequence number value (inclusive), then $S-1$ to 1 is an interval of 4 sequence numbers: $S-1$, S, 0 and 1. The user of DTP gets indexes (beginning at 0), that are relative to the sequence number interval, for each payload received.

5.3.1.3 Control Transport Protocol

The format of a CTP packet is displayed in Table 5.4. *A* determines whether this packet is an acknowledgment (ACK) packet (*A*=1) or a data packet (*A*=0). *u* denotes an unused bit and is ignored. *CRC16* is calculated over all other fields: *sequence number*, *A*, *u*, and *payload*. The payload length is determined by the packet length. Whenever a data packet is received, the receiver should send an ACK packet with the same sequence number as the data packet. Since the sequence number is encoded using 6 bits, there are $2^6 = 64$ sequence numbers. The window (see 4.3.4) size is 32, which is half of the sequence numbers.

bit	0	1	2	3	4	5	6	7
0	sequence number						A	u
8 ... n-24	payload							
n-16 n-8	CRC16							

Table 5.4: Control Transport Protocol format

In the prototype implementation, the user of CTP can be notified when a packet is acknowledged. This is used to inform the user of the application about the current state of a request. It is also used in the automatic retransmission logic so that it knows when the server has correctly received a Resend request and is expected to begin transmitting data.

5.3.2 Control Channel Protocol

The Control Channel Protocol was only required to do a few things: request web pages, abort the current transmission, and retransmit specified web page content. To enable encryption, it should also provide a way to exchange cryptographic parameters. All the message types are shown in the tables below. The first byte of each message determines the message type. If a request has a corresponding response message, the server always sends a response to that request.

The ExchangeKey request and response contain the cryptographic parameters *public key* and *nonce*. These parameters are explained in 5.5.

bit	0	1	2	3	4	5	6	7
0	0							
8 ... 520	public key							
528 ... 648	nonce							

Table 5.5: ExchangeKey request

bit	0	1	2	3	4	5	6	7
0	1							
8 ... 520	public key							
528 ... 648	nonce							

Table 5.6: ExchangeKey response

The RequestURL request (Table 5.7) tells the server to load a certain web page and transmit it over the data channel to the client. The web page is identified by a *URL*, the length of which is determined by the packet length. The server responds with a RequestURL response (Table 5.8) containing: *content length*, the sequence numbers, and the *compression parameter*. The sequence numbers, *first sequence number* to and including *last sequence number*, are used when transferring the content in the data channel. The *compression parameter*, when using the BPE compression (see 5.4), refers to the dictionary used when compressing.

bit	0	1	2	3	4	5	6	7
0	2							
8 ...	URL							

Table 5.7: RequestURL request

bit	0	1	2	3	4	5	6	7
0	3							
8 16 24 32	content length							
40 48 56	first seq. num.							
64 72 80	last seq. num.							
88	compression parameter							

Table 5.8: RequestURL response

The RequestLink request (Table 5.9) and response (Table 5.10) are similar to RequestURL. The request includes a link (see 5.3.3), encoded using 32 bits, instead of a URL. The response includes the URL corresponding to the link, so that it can be shown to the user.

bit	0	1	2	3	4	5	6	7
0	4							
8	link number							
16								
24								
32								

Table 5.9: RequestLink request

bit	0	1	2	3	4	5	6	7
0	5							
8	content length							
16								
24								
32								
40	first seq. num.							
48								
56								
64	last seq. num.							
72								
88								
88	compression parameter							
96	URL							
...								

Table 5.10: RequestLink response

The Abort request (Table 5.11) tells the server to stop transmitting content over the data channel by removing all data channel packets queued in the limiter (see 5.3.1). However, the web page request is not aborted, and the client can still request that certain packets of the request be resent.

The Resend request (Table 5.12) tells the server to retransmit specific packets. The packets are specified by a variable number of intervals of sequence numbers, for example, a single request can ask the server to retransmit packets with sequence numbers *a*-*b*, *c*-*d*, and *e*-*f*. The intervals are inclusive, so a request for packets *a*-*a* will retransmit the single packet *a*. The number of intervals is determined by the packet length. The sequence numbers refer to the last requested URL or link and must be within the interval contained in the response to that request. The packets will be sent in the order they are requested.

bit	0	1	2	3	4	5	6	7
0	6							

Table 5.11: Abort request

bit	0	1	2	3	4	5	6	7
0	7							
8 16 24	first seq. num. 1							
32 40 48	last seq. num. 1							
56 64 72	first seq. num. 2							
80 88 96	last seq. num. 2							
...	...							

Table 5.12: Resend request

5.3.3 Data Channel Protocol

The Data Channel Protocol is very simple, it consists only of the server transmitting packets to the client using DTP. The server splits the data into packets of at most 288 bytes. 288 bytes was chosen since it is divisible by 32 and is transmitted in close to 1 second over a 2.4kbit/s connection, which is a good compromise between transfer time and overhead ratio. A packet is made up to a few bytes smaller if it would otherwise break a multi-byte UTF-8 character or link marker.

The content of a web page is sent over the data channel as plain text encoded in UTF-8 together with links encoded using invalid UTF-8 byte values. The process of transforming the web page's original hypertext into formatted plain text is described in 4.1 and 5.2. Because it is plain text, no symbol or sequence of symbols has any special relation to any other symbols (unlike HTML where an opening tag is related to closing tag), so a bit error will usually affect only a single symbol. In rare cases a bit error could produce a multi-byte UTF-8 symbol, thus affecting several symbols.

Links (see 5.2.2) are sent inline with the plain text content, and are encoded using a "begin" and "end" marker like HTML `<a>` and `</a>` tags. A link-begin marker consists of 5 bytes: `fe xx xx xx xx` in hexadecimal, where `x` denotes the link number (in big-endian format). A link-end marker consists of a single byte: `ff` in hexadecimal. `fe` and `ff` are invalid and unused by UTF-8, so using those values avoids conflicts with UTF-8.

5.4 Compression

The implementation of the BPE algorithm described in 4.4 consists of two parts: a standalone script to create a dictionary from a given file, and the compression and decompression methods. Optimally, the file from which the dictionaries are created should be both specific enough to provide good compression, and general enough to achieve good compression on a large number of web pages.

For the prototype, a Swedish (see Table 5.13) and an English (see Table 5.14) dictionary were created from a few Wikipedia pages. The web pages were first run through the content extraction part so that they are in the format that is actually transferred to the client. The script itself is quite simple: for each available unused UTF-8 sequence (one byte each), it finds the most frequently occurring pair of characters in the file. The pair is then replaced with the unused byte and the byte-pair entry is added to the dictionary.

Swedish
`https://sv.wikipedia.org/wiki/Sverige`
`https://sv.wikipedia.org/wiki/P%C3%A5sk`
`https://sv.wikipedia.org/wiki/Stockholm`

Table 5.13: Web pages used for the Swedish dictionary

English
`https://en.wikipedia.org/wiki/Droxford_railway_station`
`https://en.wikipedia.org/wiki/Ice_drilling`
`https://en.wikipedia.org/wiki/University_of_Washington_station`

Table 5.14: Web pages used for the English dictionary

The compression and decompression methods use the dictionaries generated by the script. The compression algorithm is not guaranteed to replace the pairs in the exact same order as the dictionary generation script, and might not achieve the optimal compression ratio. Instead of going through the data several times like the dictionary generator, the compression algorithm only goes through it once, achieving a faster runtime.

5.4.1 The Compression Algorithm

At each step, the algorithm considers the next three characters. Assuming that the algorithm is at character b in the sequence $abcd$, it proceeds as follows:

1. If bc is not in the dictionary, the algorithm moves forward one character and repeats step 1 at this position.
2. If bc was replaced before cd when generating the dictionary, bc is encoded as B and the algorithm moves back one character (to aBd, since aB could also be a pair in the dictionary) and goes to step 1 at this position.
3. If cd was replaced before bc when generating the dictionary, the algorithm moves forward one character (skips b to cd) and goes to step 1 at this position.

There are some cases in which this algorithm does not encode the pairs in the same order as the dictionary generator, potentially producing a nonoptimal compression ratio. For instance, consider the sequence *abcd*. The dictionary generator could have encoded two pairs in the following way: abC (cd becomes C), then aB (bC becomes B). However, the compression algorithm could compress this sequence as Acd (if bc is not in the dictionary). Testing this on a few real web pages, the difference in compressed size was negligible (less than 1%).

5.4.2 Run-Time Complexity

The compression algorithm can run in $O(n)$ time. Either it moves forward (step 1 and 3) or it encodes one pair, reducing the size by 1, and moves backward 1 step (step 2). Thus, in the worst case, it takes $2n - 2 = O(n)$ steps: it always moves forward from beginning to $n - 1$, and, for every step forward, it can move one step back and reduce the remaining length by 1 (so that the number of remaining steps forward remains the same, in a sense "standing still").

The prototype implementation runs in $O(n^2)$ time since it moves all of the remaining data back one position when encoding. Compressing the Swedish Wikipedia page on "Sverige" of 96395 bytes takes about 350ms. This is good enough for a prototype with only two dictionaries. However, if dictionaries for many more languages were added, the algorithm would need to be optimized, since every web page would be compressed once for every dictionary (see 4.4).

5.5 Encryption

At the beginning of a connection, the client exchanges cryptographic parameters with the server, after which all further communication is encrypted. The key exchange follows the Elliptic-Curve Diffie-Hellman protocol (a kind of key exchange protocol), using the NIST P-256 curve (a curve for elliptic curve cryptography, recommended by NIST). The parameters are exchanged over the control channel using the *ExchangeKey* message (see 5.3.2).

Both the public key and a 16-byte nonce are sent in the key exchange. The public key is used to calculate the shared key, while the nonce is used to initialize the counter of the symmetric cipher. The nonce generated by each respective party itself is used for encryption, while the one received from the other is used for decryption. The symmetric cipher used is AES256 (a common block encryption algorithm) in counter mode (see 4.5). Because the control channel has in-order delivery while the data channel can receive packets out of order, the counter handling is different in the different channels.

5.5.1 Control Channel

The control channel initializes its counters (one for encryption and another for decryption) to 0 and increases the respective counter for every encrypted or decrypted block. So the counter value ctr_i, for block number $i \geq 0$, has the following value:

$$ctr_i = i$$

The counter value is added to the nonce before use in encryption/decryption.

5.5.2 Data Channel

The data channel handles encryption/decryption slightly differently. A starting value for the counter ctr for a specific packet is chosen based on the packet's sequence number $seq \geq 0$ in the following way, where L is the maximum length of the data in a packet and b is the block size of the encryption algorithm:

$$ctr_0 = -\left(1 + \lceil \frac{seq \times L}{b} \rceil\right)$$

This ensures that a counter value is not reused between packets in the data channel. For each block number i of the packet, the counter is decreased:

$$ctr_i = ctr_{i-1} - 1$$

Thus, the counter in the data channel is negative, which ensures that the data channel and control channel (where the counter is positive) do not use the same counter values.

6
Test of the Prototype

To evaluate the performance of the proposed solution, testing was performed both in a simulated environment and with an Iridium GO! satellite Internet device. In this chapter, the test results from the prototype and other web browsers are presented and analyzed.

6.1 Field test using Iridium GO!

An outdoor field test was performed using an Iridium GO! satellite Internet connection. Four different types of web pages were tested using the prototype, XWeb, and ELinks via SSH. The averages of the test results are presented below, while the full test data is provided in appendix A.

Due to firewall issues with the Iridium GO!, full web browsers were not possible to test. With additional effort, this could be possible, but for the sake of comparison, theoretical loading times were calculated. The web pages were measured both in full size, including assets and media, and just the HTML size. The theoretical loading times were calculated using the maximum Iridium GO! throughput of 2.4 kbit/s, equivalent of 0.3 kB/s. Ordinary web browsers can be assumed to at best perform as the theoretical loading times since they do not perform any additional compression. In a real-world scenario, occasional issues such as lost packets and bit errors likely cause the loading times of ordinary browsers to increase.

6.1.1 Total Data Transferred

The total data transferred was only reliably measured for the prototype, due to technical issues. Since the goal of this project is to achieve reasonable loading times, the time can be considered more important than the data size. As the transferred data is compared to both the full page size and the HTML size, this section serves as a performance review of the content extraction solution in combination with the compression algorithm.

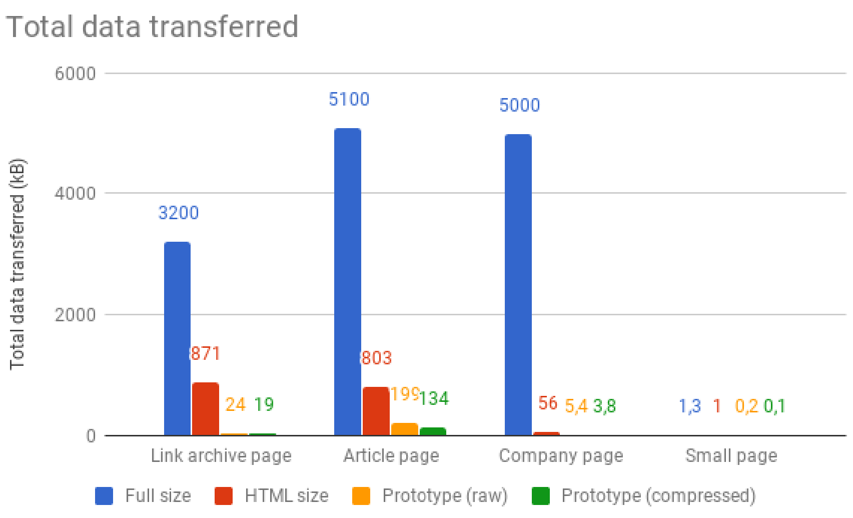


Figure 6.1: Total data transferred per page.

In Figure 6.1, the page size is compared to the total data transferred by the prototype. The prototype achieves varying reduction and compression ratio depending on the type of page, achieving a compression ratio of about 72% on average (not including the small page). Generally, pages containing non-text media are reduced the most compared to the full-size, which becomes evident in the results for both the link archive, article and company page, with full size reductions of 99.4%, 97.4%, and 99.9% respectively.

As text content is never removed, the reduction depends on how much of the HTML is actual text content. For instance, the link archive page had a similar HTML size (871 kB) compared to the article page (803 kB), but the former consisted of more non-text content. As a result, the prototype proxy server was able to reduce the link archive page greatly, resulting in 97.3% reduction from the HTML size using text extraction alone, from 871 kB to 24 kB. The resulting 24 kB of text was then compressed by about 20%, resulting in 19 kB. The article page was not reduced nearly as much, since a larger part of it's HTML consisted of text content, reducing only by 75.2%, from 803 kB to 199 kB, resulting in 134 kB after compression of another 32%.

6.1.2 Time to Completion

The time to completion defines the time it takes to fully load all the content for a specific browser solution. The prototype only loads the text content, while XWeb tries to load certain assets and images. Therefore, the resulting pages may differ in content, but the completion times, while not necessarily comparable, are still relevant to evaluate the user experience.

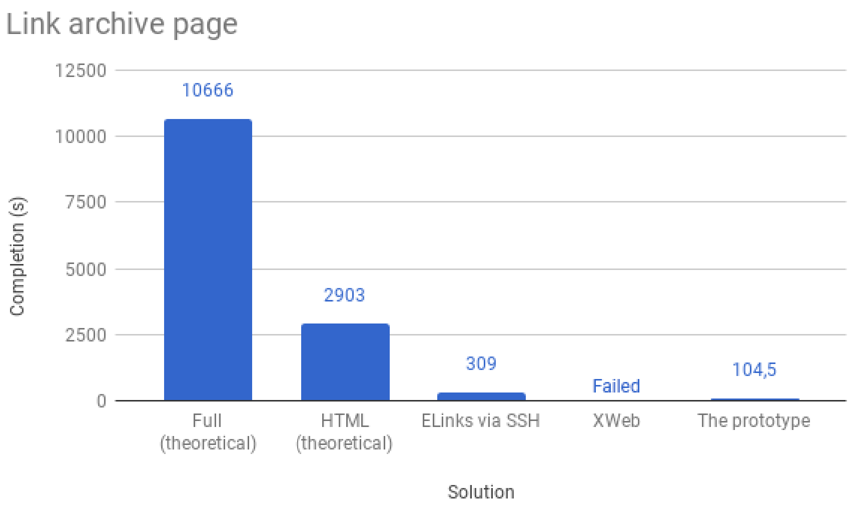


Figure 6.2: Average completion times for the link archive page.

The link archive page is quite heavy in terms of both content, images, and assets. With a size of 3.2 MB for the full page and 871 kB for the HTML. Therefore, as seen in Figure 6.2, the theoretical loading times for the full page and the HTML size are about three hours and 48 minutes respectively. Using ELinks via SSH, the full page was accessed in about five minutes, which is nearly ten times as fast as the theoretical HTML loading time. The XWeb browser failed to load the link archive page entirely despite multiple attempts, while the prototype loaded the entire page in an average of slightly less than two minutes.

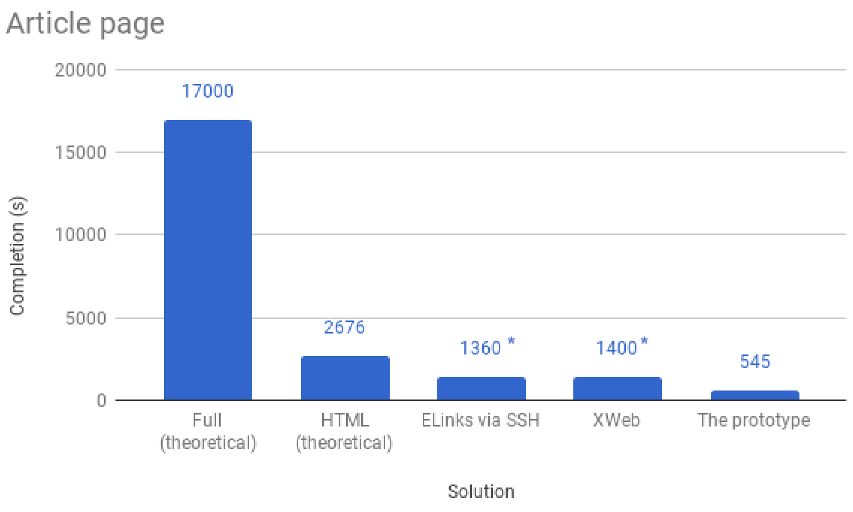


Figure 6.3: Average completion times for the article page.

The article page consists of plenty of text content, which is why the prototype completion time is about five times higher than the link archive page, despite similar HTML sizes. As seen in Figure 6.3, the prototype loaded the full page in about

9 minutes, while ELinks and XWeb were estimated to finish in about 22 and 23 minutes respectively. The ELinks test was canceled halfway at 11 minutes due to time constraints. XWeb failed after 7 minutes at around 30% and did not prove more reliable in subsequent attempts. To provide a comparison, imagining a stable connection throughout the test, the ELinks and XWeb results were extrapolated.

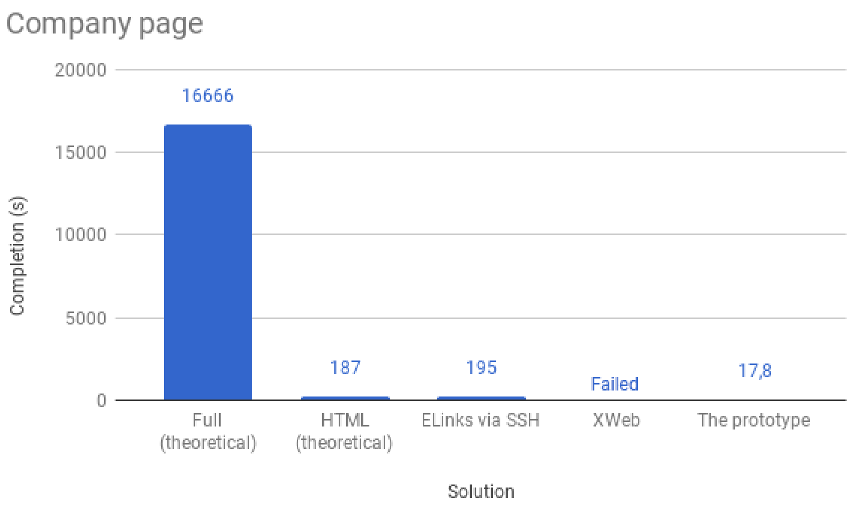


Figure 6.4: Average completion times for the company page.

The company page is heavy on assets at 5 MB but relatively small in terms of text content with an HTML size of just 56 kB, which is reflected in the results seen in Figure 6.4. The full page has a theoretical loading time of about 5 hours, while the HTML can be loaded in about three minutes. XWeb again proved unreliable and never successfully loaded the page, while the prototype performed well at 17.8 seconds. ELinks via SSH did take slightly more than 3 minutes to complete, hence performing worse than the theoretical HTML loading time.

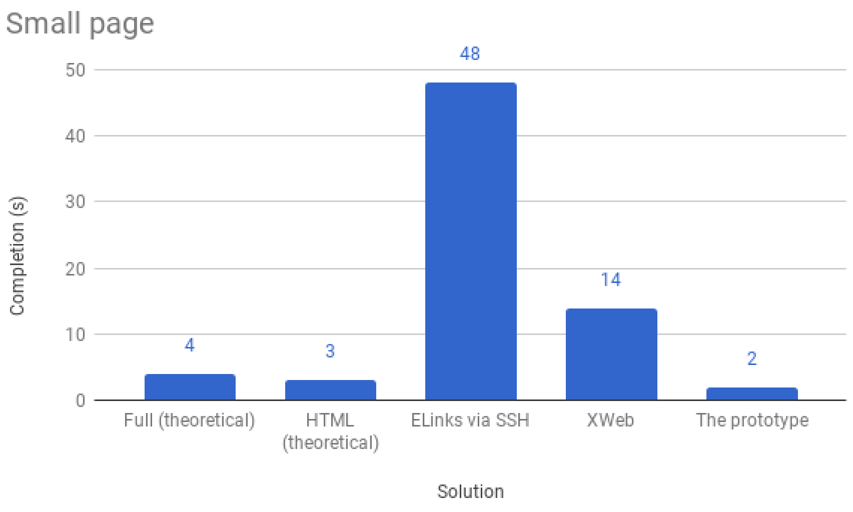


Figure 6.5: Average completion times for the small page.

For the smallest page, the time to completion was similar to the theoretical loading times, as seen in Figure 6.5 above. The prototype received only 0.2 kB worth of data in 2 seconds, while the HTML page size of 1 kB theoretically would load in 3 seconds. Evidently, the prototype reduced the page to 1/5 of the data, but only improved loading times by 1/3. A plausible reason for this is the delay induced by the RTT, which is not included in the theoretical time. Thus, for very small pages, while still loading slightly faster, the prototype is similar to ordinary web browsers in terms of loading times.

The proxy server used by the prototype does take some time to request, process and compress the requested web page. However, the theoretical loading times do assume a stable connection with no lost packets or bit errors, for which the ordinary web browsers would retransmit the packets repeatedly until correct arrival, due to the sole use of TCP in the HTTP protocol. As the prototype is designed to handle such issues nicely, the prototype can be assumed to provide a better user experience if the connection is unreliable.

ELinks via SSH performed very poorly at 48 seconds. Since SSH is used to remotely control the application, every state of the remote ELinks session, including the loading screen, will be transmitted before the content starts arriving. The XWeb browser performed better than ELinks via SSH, but poor compared to both the theoretical loading times and the prototype. The small page of about 1 kB was loaded in 14 seconds, compared to the theoretical time of 4 seconds and the prototype results of 2 seconds.

6.1.3 Time to First Content

As the user may be able to begin reading the text content before the page has finished loading, it is desirable to measure the time to the first content. Delivering smaller parts of the content regularly in a streaming manner was a design goal of the prototype, which is evident in the results.

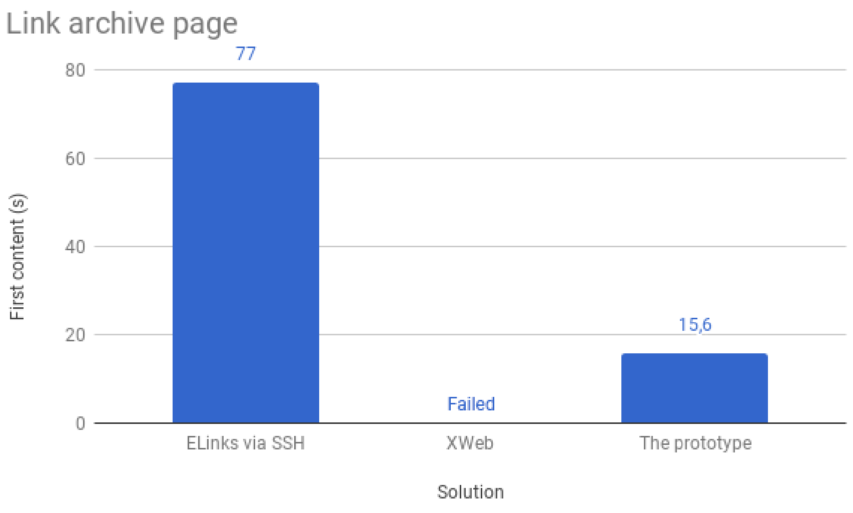


Figure 6.6: Average time to first content for the link archive page.

For the link archive page, seen in Figure 6.6, the prototype displayed the first part content after an average of just 15.6 seconds, while Elinks via SSH was able to show the first part of the page after 77 seconds. The XWeb browser failed to load the page entirely and is therefore not comparable.

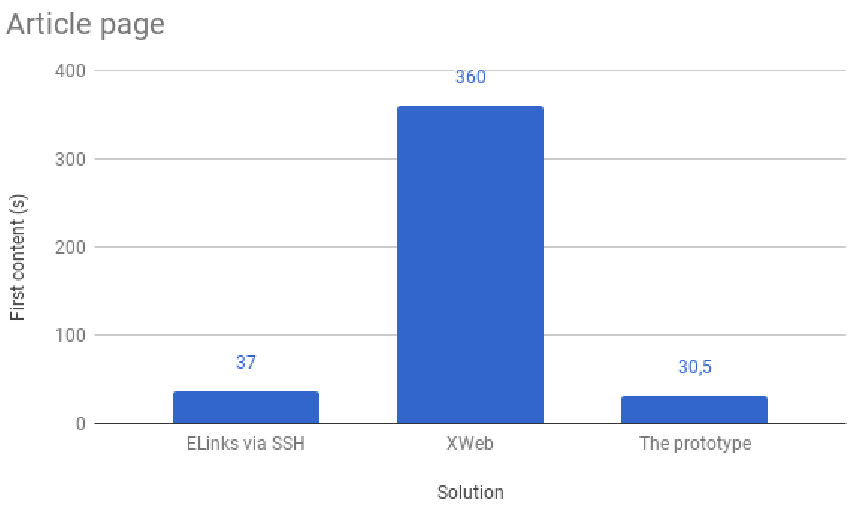


Figure 6.7: Average time to first content for the article page.

As seen in Figure 6.7, the prototype was loaded the first content of the article page in twice the time compared to the link archive page, despite its 8% smaller HTML size. After further investigation, the cause appeared to be the inefficient prototype implementation of the link extraction, which for the article page needed a large part of the initial loading time to process the nearly 3500 links of the article page, as opposed to just about 200 for the link archive page. The XWeb browser successfully displayed the first part of the content after 360 seconds, more than 10 times as slow as the prototype.

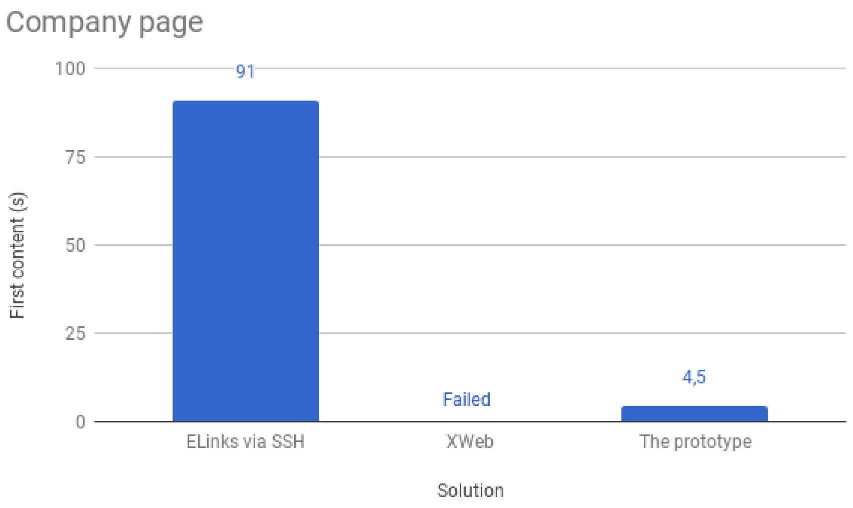


Figure 6.8: Average time to first content for the company page.

The company page contains relatively little text content, with a HTML size of just 56 kB and about 150 links. Hence, as seen in Figure 6.8, the prototype was able to process and transmit the first part of the content in just 4.5 seconds, as opposed to the 91 seconds needed by ELinks via SSH. The XWeb browser again failed to load the page entirely.

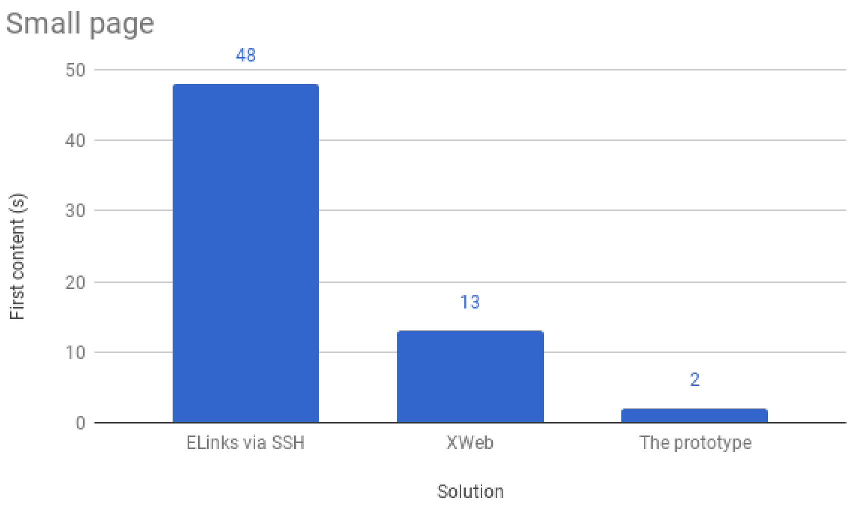


Figure 6.9: Average time to first content for the small page.

As seen in Figure 6.9, the prototype achieves a loading time of 2 seconds for the small page. The page is just 1.3 kB, which theoretically loads in about 4 seconds, yet XWeb needed an average of 14 seconds to display the first part of the content. All tested solutions managed to display the entire content instantly, which is the reason why the time to first content and time to completion are the same.

ELinks via SSH differed greatly from the other solutions, requiring 48 seconds. Generally, the time it takes for ELinks via SSH to display the first content varies between about half a minute to one and a half minutes. The variation does not have an obvious correlation to the type of content or size of the page, as the small page of just about 1 kB displayed in 48 seconds while the article page of 803 kB needed just 37 seconds. Assumedly, as the ELinks application is displayed from a remote server, the time required to send the visible part of the page (as seen by the ELinks browser without scrolling down) might be relatively consistent despite the page length.

6.1.4 Perceived User Experience

ELinks via SSH sometimes provided a reasonable alternative in terms of loading times. Since it only transmits the visible parts of the screen, it does allow the user to view the page faster than the page can be downloaded in full. However, the user experience is arguably quite bad, as user input such as entering the web address or scrolling down is very unresponsive. As the actions are performed via SSH, each keystroke of the user input, such as the web address, is sent individually to the server. The perceived performance of typing the address into ELinks via SSH is therefore arguably bad. During testing, delays of several seconds were occasionally observed, from keystroke until the character becomes visible.

The XWeb browser did not perform well, despite being a commercial solution specifically built for satellite Internet services such as Iridium GO!. While seemingly providing a robust and responsive user interface that is similar to those of ordinary web browsers, XWeb proved unreliable and failed to load the requested pages multiple times, resulting in a bad user experience.

The prototype proved reliable and responsive during the field test, despite a few lost packets per attempt, and an event where the connection was lost for a few seconds. When a packet was lost, the prototype continued to stream all the other content, while providing the user with an option to manually resend the lost packet. The prototype did successfully resend the packets automatically, after completing loading the rest of the page. Arguably, the prototype provided a good user experience which feels responsive due to the continuous stream of content. The prototype also achieved good resilience to connection issues, compared to both XWeb and ELinks via SSH.

6.2 Performance in a Simulated Environment

During testing, it was discovered that Iridium GO! did not let any packets with bit errors through. Hence, in order to test the prototype's handling of bit errors, in terms of performance and fault tolerance, a simulator had to be used. The simulator simulates packet loss, bit errors, temporary connections loss, limited bandwidth, and packet delay. Packet and bit error rates are modeled as uniform probability distributions, that is, each packet or bit is equally likely to be faulty. Temporary connection loss is modeled as a Markov chain with two states, updated each millisecond. The bandwidth and delay are fixed.

The simulator settings for this test are shown in Table 6.1. The Bit Error Rate is very high at $\frac{1}{1000}$; if data packets are $288 * 8 = 2304$ bits (see 5.3), a data packet contains approximately 2.3 bit errors on average, and only $\left(1 - \frac{1}{1000}\right)^{288\times 8} \approx \frac{1}{10}$ packets will be error free on average.

Packet Error Rate	0.05
Bit Error Rate	0.001
Bandwidth	2400 bits/s
Average up time	120 seconds
Average down time	30 seconds
Delay	2 seconds

Table 6.1: Simulator test settings

The result of the test is shown in Table 6.2. Only automatic retransmission, no manual retransmission, was used (see 5.1.2). When the connection is temporarily lost, all packets are dropped. They are automatically retransmitted once the server is finished transmitting all packets. Because of the relatively high packet error rate and connection losses, multiple sequential packets were often lost, which could not be resent using manual retransmission even if the user wished. A more sophisticated manual retransmission process would solve this problem.

Web page	`https://en.wikipedia.org/wiki/Sweden`
Time to first content	29s
Time to 100% loaded	16m 35s
Time to 100% correct	aborted after 1h 30m
Size	203735 bytes
Compressed size	137787 bytes (67.6%)

Table 6.2: Simulator test result

Examples of the bit errors encountered are shown in Figure 6.10. The bit errors are clearly noticeable. However, when read in context, the content is easily legible.

Because of the compression, multiple characters are sometimes affected, but this is not detrimental to the readability. However, when the content is not normal text, bit errors could render some content incomprehensible, for example, numbers in a table. In that case, the user will have to manually select the content to be retransmitted.

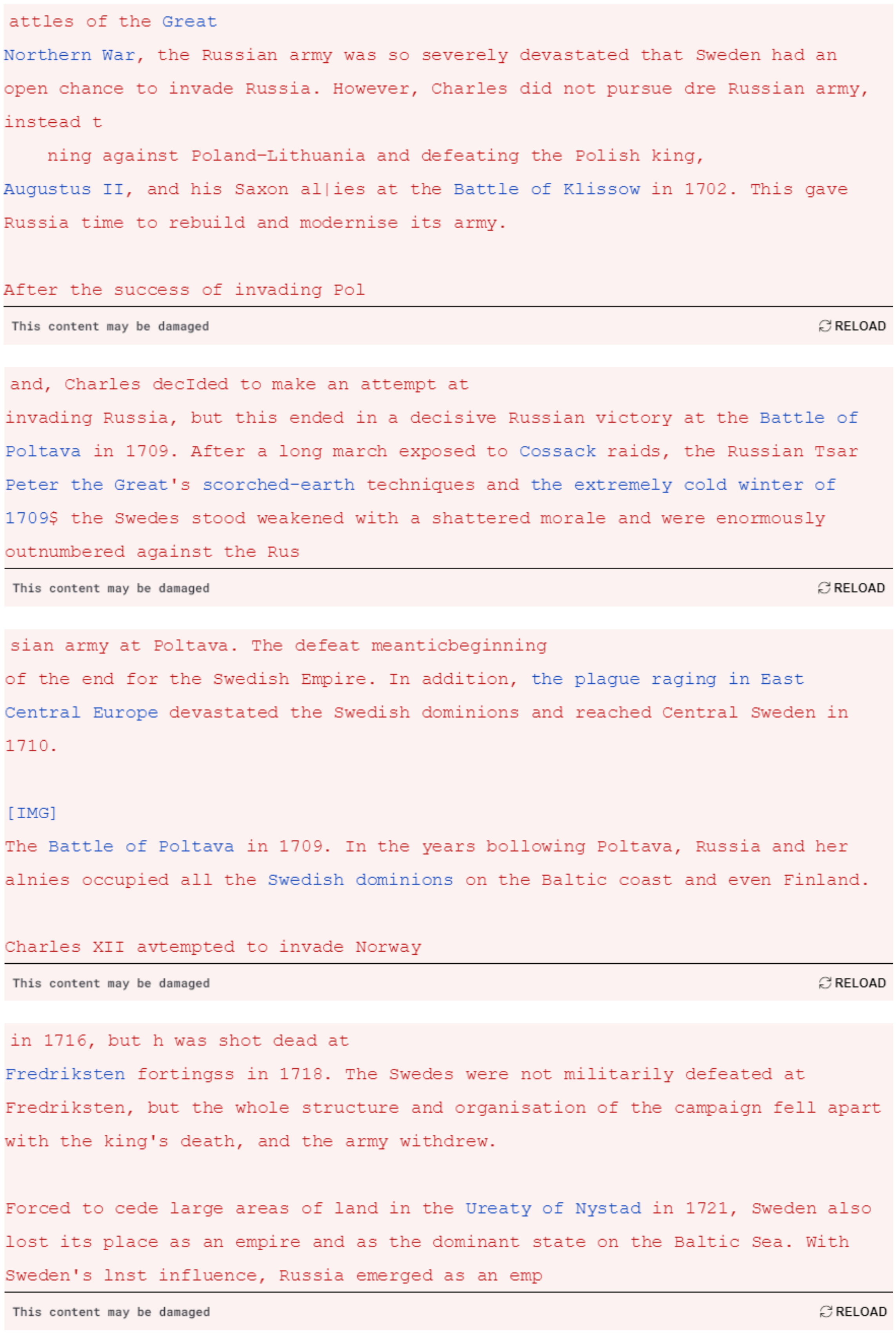
Only considering bit errors, the theoretical average best time until all packets have been loaded correctly can be approximated as a sum of stages, where in each stage every packet that has not been transmitted correctly is (re)transmitted. This approximation does not take into account network overhead, packet losses or connection losses. It gives the following equation, where t is the time in seconds, r is the rate of corrupt packets, s is the size of the web page in bytes, and b is the bandwidth in bytes/s:

$$t = \sum_{i=0}^{\infty} (1-r)^i \frac{s}{b}$$

Assuming the parameters are the same as in the test, with the correct packet ratio simplified to $r = 0.1$, the approximate average best loading time is:

$$t = \sum_{i=0}^{\infty} (1-0.1)^i \frac{203735}{300} = 6791.17$$

Which is about 113 minutes. Thus, on connections with very high bit error rates, the prototype's tolerance of bit errors results in much lower loading times (16m 35s) than would otherwise be possible, while still maintaining quite high readability of the content.



```
attles of the Great
Northern War, the Russian army was so severely devastated that Sweden had an
open chance to invade Russia. However, Charles did not pursue dre Russian army,
instead t
    ning against Poland-Lithuania and defeating the Polish king,
Augustus II, and his Saxon al|ies at the Battle of Klissow in 1702. This gave
Russia time to rebuild and modernise its army.

After the success of invading Pol
```

```
and, Charles decIded to make an attempt at
invading Russia, but this ended in a decisive Russian victory at the Battle of
Poltava in 1709. After a long march exposed to Cossack raids, the Russian Tsar
Peter the Great's scorched-earth techniques and the extremely cold winter of
1709$ the Swedes stood weakened with a shattered morale and were enormously
outnumbered against the Rus
```

```
sian army at Poltava. The defeat meanticbeginning
of the end for the Swedish Empire. In addition, the plague raging in East
Central Europe devastated the Swedish dominions and reached Central Sweden in
1710.

[IMG]
The Battle of Poltava in 1709. In the years bollowing Poltava, Russia and her
alnies occupied all the Swedish dominions on the Baltic coast and even Finland.

Charles XII avtempted to invade Norway
```

```
in 1716, but h was shot dead at
Fredriksten fortingss in 1718. The Swedes were not militarily defeated at
Fredriksten, but the whole structure and organisation of the campaign fell apart
with the king's death, and the army withdrew.

Forced to cede large areas of land in the Ureaty of Nystad in 1721, Sweden also
lost its place as an empire and as the dominant state on the Baltic Sea. With
Sweden's lnst influence, Russia emerged as an emp
```

Figure 6.10: Examples of bit errors

7 Discussion

In this chapter, the viability of basic web surfing using the developed prototype is discussed, based on the data presented in Chapter 6. The importance of each of the five subproblems (content extraction, web searching, network protocol, compression, and encryption) for these results are also discussed. Suggestions for future work that could further improve web surfing on very low bandwidth connections are presented. Lastly, ethical aspects are discussed.

7.1 Field Test

The testing of the prototype indicates that it is possible to browse basic web pages within a reasonable time (see Chapter 6). The low bandwidth connection does limit the amount of data that can be transmitted, and it is therefore practically impossible to transmit large multimedia over such a connection. However, the arguably most important type of data is handled by the prototype, namely the textual content. The streaming nature of the prototype should alleviate the long waiting times when loading a web page, allowing the user to start reading in a few seconds (see 6.1.3). *This result was expected, since the bandwidth is high enough to transmit text faster than it can be read, which, in absence of unexpected obstacles, is all that is required to enable basic web surfing.*

The delay before the first content was shown was quite high for some web pages (see 6.1.3). The bottleneck was identified as an external DOM package (jsdom) used when exchanging the links' URL:s with ID:s. Suggestions for removing this bottleneck, and other ways of improving the response time, are presented in 7.3.

The delay from RTT in the field test was estimated to be around 1.5 seconds, which is consistent with the usual delay on Iridium (see 2.8.2). While the latency is higher than most ordinary wired or wireless connections on land, it is a relatively short time compared to the time of transmitting a website over a connection with very low bandwidth.

No previous research has been found concerning web browsing on very low bandwidth connections, but there are a few commercial products advertised to be optimized for such conditions. The main alternative, XWeb, performed much worse than the prototype (see 6.1) despite being advertised as suitable for connections such as Iridium GO!, even completely failing to load several of the pages. The assumed reason is that XWeb is too similar to a normal web browser, supporting many of the ordinary web page features. XWeb tries to transmit most of the content of a web page, including HTML, scripts, and multimedia, albeit more compressed than a normal web browser would (see 1.1). Hence, it seems that compression is not enough, and therefore an alternative solution is desirable, such as content extraction.

The field test performed in Chapter 6.1 has several drawbacks that limit the scope of the conclusions that can be drawn from it. Only four web pages were tested, although they were chosen under the assumption that they were typical of four different types of web pages. Furthermore, the number of tests performed for the different web pages and the different browsers vary. The pages and solutions were tested a different amount of times due to their high loading times. For instance, the article page alone was estimated to take about 22 minutes to access in full with ELinks via SSH, making it very time consuming to test many times.

There are also other tests that could have been performed. For example in terms of usability, robustness, and reliability, since these were all goals of the prototype (see Chapter 3). Arguably, these aspects are requirements for basic web browsing. However, these parts were discussed in 6.1.4 based on the authors' experience of using the prototype, which may be of subjective nature.

7.2 Importance of the Subproblems

Content extraction and a suitable protocol appear to be the most vital parts of enabling web surfing on very low bandwidth connections. By extracting only the text of a web page, content extraction reduces page size to levels that in most cases can be transferred over the low bandwidth connection in reasonable time. Even on an article page that primarily consists of text, the HTML size is reduced by 1/4, and much more on other types of web pages (see 6.1.1). This reduction also includes compression and link encoding.

The importance of the protocol can be inferred from the difference between ELinks via SSH and the prototype in Chapter 6, since ELinks via SSH is essentially streaming content extracted in the same way as the prototype. Ignoring the problematic user interface of ELinks via SSH, it takes considerably longer to start streaming content and to load the whole page, compared with the prototype. This is possibly due to SSH and TCP not being suitable for Iridium GO!. The exact reason why it performed worse, and whether it was caused by SSH or TCP, is unclear. Likewise, it is unclear which parts of the developed protocol provide the most benefits over SSH and TCP.

The prototype's tolerance for bit errors proved very important when tested in a simulated environment (see 6.2). Any application that does not tolerate bit errors would have much longer load times due to automatic retransmission. However, it is uncertain if connections as fault-prone as in the simulation are commonly found. The fault tolerance could therefore prove to be less important in real scenarios.

Furthermore, since no bit errors were found during the Iridium GO! field test, the prototype's tolerance for bit error did not improve the live test results.

The BPE compression (see 4.4 and 5.4) seemed to achieve compression ratios of up to 60% for real web pages, and an average of 72% in the field test (see 6.1.1). Hence, while compression would reduce load times by the same ratio, the difference is much smaller than the one achieved by the content extraction. It is, thus, not of vital importance for enabling basic web surfing over very low bandwidth connections.

Web searching and encryption do not improve the performance as measured in Chapter 6 in any way. Arguably, however, web searching is required for basic web surfing. Encryption is not required for web surfing but provides confidentiality which is important for many people.

7.3 Suggestions for Future Work

This section presents some suggestions for future work on improving web surfing on very low bandwidth connections. These include general research with, and possible improvement of, the prototype. Additional features should primarily be focused on improving ease of use and loading speed.

Extended testing of the prototype should be performed. For example:

- Improve the field test already performed (see 6 and appendix A). The number of tries for each web browser should be increased, as well as the number of web pages tried.
- Tests should be performed in different signal conditions, to test reliability and robustness.
- Usability testing, to find out if real users find the prototype usable for basic web surfing, or what improvements are otherwise necessary.
- Tests regarding which parts of the developed solution that are the most important and why existing solutions perform poorly.

The context extraction can be improved. The user might not be interested in all content of a web page, so some kind of "selective loading" would be a improvement. A content scoring system, such as the one described in 4.1.2, could be added, to remove irrelevant content and let the user decide what content to transfer (for example, only the menu, or only the text of an article). A useful feature would be to only show a sample of each type of content of a web page or a table of contents and let the user select the interesting ones. It could also be extended to let the user search for specific content on a web page. This line of reasoning could be extended very far, even including an AI on the server responding to user queries, trying to select the information the user seems to be interested in.

The web page processing time on the server can be reduced. The current bottleneck is the jsdom package. Few of jsdom's advanced features are used, so a first step to improve response times would be to replace this package, for example with the parse5 package. This could in itself bring down the processing time to under 10 seconds. If this reduction is not enough, ways to start transmitting page content before the whole page has been processed by the server could be investigated.

Standard features of normal web browsers could be implemented, like tabs and bookmarks. General forms support could also be implemented, perhaps using the solution suggested in 4.2.2. The already implemented cache could be improved, for instance by alerting the user if the cache is outdated.

A more advanced user interface could be designed. The interface should be suitable for the features required for the low bandwidth connection. Improvements to the interface would be required for some of the possible improvements mentioned above.

Further research can be performed on the compression scheme used. Little research was performed before choosing BPE, so it is quite possible that the compression could be improved. For example, the use of Resynchronizing Huffman Codes could be investigated (see 4.4.1).

7.4 Ethical Aspects

There are few ethical aspects to take into account for this project. The prototype only provides faster access to the Internet, and neither censors information (disregarding the removal of any visuals or audio) nor saves personal data. The potential impact is thus largely limited to the user and the owners of any web page that the user connects to.

The service removes a lot of content before transmission to the client, including most advertisements, thus acting as an "ad-blocker". Many web pages rely on revenue from advertisements, and users of the prototype will not contribute to their income. However, the target audience of this service is small and probably visits the same pages when not in remote locations. Because of this, the negative economic impact is probably negligible.

Currently, the proxy server only keeps track of links, the current page, and the IP-address of the user during a session, and when the server is shut down, no information is kept. While the scope of this prototype is narrow and covers neither cookies nor forms, some personal data could still be gathered. For example, only a slight modification of the server is required to save and export the browsing history, which could be sold to advertisers or other agents.

Since the proxy server is based in London, the solution might also be used to circumvent censorship. Users in countries like China or Iran might be able to access sites that are unavailable when using the web services provided by their national providers. However, it is possible that the Iridium Satellite System already provides this service.

8 Conclusions

This study posed the question of whether it is possible to enable basic web browsing on a connection with very low bandwidth and high latency, using a fault-tolerant text streaming approach. Through the development and testing of a working prototype, this was confirmed to be correct. Considerations and decisions made during development, as well as the finished prototype, have been documented. The solution seems to be unique, and no previous research on how to enable web browsing under the specified conditions could be found.

The most important aspects for enabling such web browsing appear to be *content extraction* and *content streaming*. By extracting only the textual content of a web page, the time it takes to transfer the web page is often reduced to minutes rather than hours. This is still a long time compared to normal web browsing, but streaming the text allows the user to start reading from the top of a web page before the whole page is loaded, usually within 30 seconds. However, the content is always transmitted from the top of the web page, which might not be the content the user is interested in. Future research on how to allow the user to select which parts of a web page to transfer could lead to major improvements in this area.

Low bandwidth satellite Internet connection paired with the current web browsing solutions do not provide a reliable web browsing experience. The prototype generally worked better than the tested existing solutions and achieved viable loading times. However, it also has fewer features than other solutions. If the prototype is expanded further it could be a preferable alternative to existing solutions for web browsing on slow satellite connections.

References

Abdul Jabbar, M., & Frost, V. (2003). *Multi-link iridium satellite data communication system.* https://www.ittc.ku.edu/~frost/KU-NSF-Iridium-experience-latest.ppt. (Retrieved: 2018-02-02)

Apple Inc. (n.d.). *macos - safari - apple.* Retrieved from https://www.apple.com/safari/ (accessed 2018-05-10)

Border, J., Griner, J., Montenegro, G., Shelby, Z., & Kojo, M. (2001). Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. *Network Working Group.* Retrieved from https://tools.ietf.org/html/rfc3135

Bova, T., & Krivoruchka, T. (1999). *Reliable udp protocol.* Retrieved 2018-03-26, from https://www.ietf.org/proceedings/44/I-D/draft-ietf-sigtran-reliable-udp-00.txt

Caini, C., & Firrincieli, R. (2006). End-to-end tcp enhancements performance on satellite links. Viale Risorgimento 2, 40136, Bologna, Italy: IEEE.

Caini, C., Firrincieli, R., & Lacamera, D. (2006). PEPsal: a Performance Enhancing Proxy designed for TCP satellite connections. In *2006 ieee 63rd vehicular technology conference* (Vol. 6, pp. 2607–2611). IEEE. Retrieved from http://ieeexplore.ieee.org/document/1683339/ doi: 10.1109/VETECS.2006.1683339

Chau, M., & Chen, H. (2008, January). A machine learning approach to web page filtering using content and structure analysis. *Decision Support Systems*, *44*(2), 482–494. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167923607000875 doi: 10.1016/J.DSS.2007.06.002

Cowley, J. (2013). *Communications and Networking.* London: Springer London. doi: 10.1007/978-1-4471-4357-4

Elinks. (2012). *ELinks - Full-Featured Text WWW Browser.* Retrieved 2018-04-05, from http://elinks.or.cz/

ENet: Enet. (2015). Retrieved 2018-04-19, from http://enet.bespin.org/

Geoborders Satellite Ltd. (2012). *IRIDIUM - How it works.* Retrieved 2018-03-20, from http://www.iridium.it/en/iridium.htm

Global Marine Networks. (2018). *Xweb satellite phone internet.* Retrieved 2018-02-02, from http://www.globalmarinenet.com/product/xweb/

Google. (n.d.). *Data Saver - Google Chrome.* Retrieved 2018-02-13, from https://developer.chrome.com/multidevice/data-compression

Goralski, W. (2017). *The illustrated network: how TCP/IP works in a modern network.*

GrandDeluxe@Wikipedia. (2011). *Iridium coverage animation.* https://commons.wikimedia.org/wiki/File:Iridium_Coverage_Animation.gif. (Licensed under public domain. Figures were generated with SaVi, software written at the Geometry Center, University of Minnesota.)

Gupta, S., Kaiser, G., Neistadt, D., & Grimm, P. (2003). DOM-based content extraction of HTML documents. In *Proceedings of the twelfth international conference on world wide web - www '03* (p. 207). New York, New York, USA: ACM Press. Retrieved from `http://portal.acm.org/citation.cfm?doid=775152.775182` doi: 10.1145/775152.775182

Ito, A. (n.d.). *w3m manual.* Retrieved 2018-04-03, from `http://w3m.sourceforge.net/MANUAL`

klueska. (2014). *plan9.* `https://github.com/brho/plan9/blob/master/sys/src/9/ip/rudp.c`. GitHub.

Kruse, H. (1995). *Data communications protocol performance on geostationary satellite links – lessons learned using acts.* http://www.its.ohiou.edu/kruse/publications/aiaa96.pdf.

Kurose, J., & Ross, K. (2012). *Computer networking: A top down approach - 6th ed.* Pearson.

KVH Industries, I. (2018). *TracPhone V7-HTS – KVH Industries.* Retrieved 2018-01-25, from `https://www.kvh.com/Leisure/Marine-Systems/Phone-and-Internet/mini-VSAT-Broadband/TracPhone-V7HTS.aspx`

leenjewel. (2018). *node-kcp.* `https://github.com/leenjewel/node-kcp`. GitHub.

McAnlis, C. (2013). *Text Compression for Web Developers - HTML5 Rocks.* Retrieved 2018-04-13, from `https://www.html5rocks.com/en/tutorials/speed/txt-compression/`

McMahon, M. M., & Rathburn, R. (2005, June). *Measuring latency in iridium satellite constellation data services.* http://www.dtic.mil/get-tr-doc/pdf?AD=ADA464192.

Mellow, C. (2004). *The Rise and Fall and Rise of Iridium | Space | Air & Space Magazine.* Retrieved 2018-03-16, from `https://www.airspacemag.com/space/the-rise-and-fall-and-rise-of-iridium-5615034/`

Menezes, A. J., Oorschot, P. C. v., & Vanstone, S. A. (2001). *Handbook of applied cryptography.* Boca Raton: CRC.

Microsoft Corporation. (n.d.). *The Better Web Browser for Windows 10 | Microsoft Edge.* Retrieved from `https://www.microsoft.com/en-gb/windows/microsoft-edge` (accessed 2018-05-10)

Mozilla Corporation. (n.d.). *Firefox Reader View for clutter-free web pages | Firefox Help.* Retrieved from `https://support.mozilla.org/en-US/kb/firefox-reader-view-clutter-free-web-pages` (accessed 2018-05-10)

Mozilla Organization. (2010). *readabilty/Readability.js at master.* Retrieved 2018-04-04, from `https://github.com/mozilla/readability/blob/master/Readability.js`

Naamani, M. (2015). *enet - npm.* Retrieved 2018-04-19, from `https://www.npmjs.com/package/enet`

Postel, J. (1980). *User datagram protocol* (STD No. 6). RFC Editor. Internet Requests for Comments. Retrieved from `http://www.rfc-editor.org/rfc/rfc768.txt` (`http://www.rfc-editor.org/rfc/rfc768.txt`) doi: 10.17487/RFC0768

Rahman, A. H., A. F. R., & Hartono, R. (2001). Content extraction from html documents. In *Proceedings of the first international workshop on web document analysis.*

Robert, L., & Nadarajan, R. (2009). Fault-tolerant text data compression algorithms. *International Journal of Information Technology and Web Engineering (IJITWE), 4*(2), 1-19.

Salomon, D., & Motta, G. (2010). *Handbook of data compression* (5th ed.). New York: Springer.

shovon. (2015). *node-rudp.* `https://github.com/shovon/node-rudp`. GitHub.

skywind3000. (2018). *kcp.* `https://github.com/skywind3000/kcp`. GitHub. (Github page of KCP)

Stanford. (n.d.). *How The Web Works.* Retrieved 2018-02-13, from `https://web.stanford.edu/class/cs101/network-3-web.html`

Sturza, M. A. (1995). Architecture of the teledesic satellite system. In *Proceedings of the international mobile satellite conference* (Vol. 95, p. 214).

The Internet Society. (1999). *Hypertext Transfer Protocol – HTTP/1.1.* Retrieved 2018-02-13, from `https://www.w3.org/Protocols/rfc2616/rfc2616.html`

The Unicode Consortium. (2017). *The Unicode Standard: Version 10.0 – Core Specification.* Mountain View, CA: Unicode Consortium.

Voyage Adviser. (2015). *Why does my Iridium satellite communications perform so poorly? Likely because its an old 2000 technology and its now 2015? USE IT WISELY SHOWN BELOW.* Retrieved 2018-03-20, from `http://northwestpassage2015.blogspot.se/2015/06/why-does-my-iridium-satellite.html`

Weintrit, A. (2011). *Navigational systems and simulators : marine navigation and safety of sea transportation.* CRC Press/Balkema.

Yunis, H. (2016). *Content extraction from webpages using machine learning* (Unpublished master's thesis). Bauhaus-Universität Weimar.

A
Field Test Results

This appendix presents the raw test results as gathered from the field test, where the implemented prototype was compared to several other web browsers using an Iridium GO! satellite Internet connection. The bandwidth was 2.4 kbit/s, which is equal to 0.3 kB/s. The prototype was tested precisely with system time but the other time measurements were done by manually using a stopwatch. Time varies from test to test due to external factors such as signal strength, and that the precision of the measurements is quite low. Therefore, the results have been rounded to integers. However, as the numbers differ greatly from each other, the results still provide an indication of how the solutions perform in comparison to each other. The numbers presented are the mean value of at least one, and sometimes several, tests of the same page.

A.1 Link Archive Page

This section presents the results from testing the front page of Swedish news website Aftonbladet. The page consists of many images together with shorter headlines and paragraphs.

URL	`https://aftonbladet.se`
Full size	3200 kB
HTML size	871 kB

Table A.1: Link archive page meta data

	First content (s)	Completion (s)	Total size (kB)
Full (theoretical)	N/A	10666	3200
HTML (theoretical)	N/A	2903	871
The prototype	16	105	24 (19 compressed)
ELinks via SSH	77	309	N/A
XWeb	Failed [1]	Failed [1]	N/A

Table A.2: Link archive page results

[1] The page did not load at all using XWeb.

A.2 Article Page

This section presents the results from testing the English Wikipedia page for "Sweden". The article is quite long and consists of many thousands of words.

URL	`http://en.wikipedia.org/wiki/Sweden`
Full size	5100 kB
HTML size	803 kB

Table A.3: Article page meta data

	First content (s)	Completion (s)	Total size (kB)
Full (theoretical)	N/A	17000	5100
HTML (theoretical)	N/A	2676	803
The prototype	31	545	199 (134 compressed)
ELinks via SSH	37	1360 [1]	N/A
XWeb	360 [2] [3]	failed [4]	N/A

Table A.4: Article page results

[1] The test was canceled after 680 seconds, at around 50%, due to time constraints. The full completion time for ELinks over SSH was estimated to be around 1360 seconds.

[2] This does not include the time taken to establish an initial connection through the Iridium Mail & Web app. This typically took multiple minutes.

[3] Only 2 out of 15 trials showed any content, one after 300 seconds, one after 420 seconds.

[4] The most successful request loaded around 30% of the content loaded. The full completion time for XWeb can be estimated to be around 1400 seconds.

A.3 Company Page

This section presents the results from testing the front page of the Iridium company website. The website consists of several photos and a few paragraphs of text.

URL	`https://iridium.com`
Full size	5000 kB
HTML size	56 kB

Table A.5: Company page meta data

	First content (s)	Completion (s)	Total size (kB)
Full (theoretical)	N/A	16666	5000
HTML (theoretical)	N/A	187	56
The prototype	5	18	5.4 (3.8 compressed)
ELinks via SSH	91	195	N/A
XWeb	failed [1]	failed [1]	N/A

Table A.6: Company page results

[1] The page did not load at all using XWeb.

A.4 Small Page

This section presents the results from testing example.com, which is a very small web page consisting of only a few sentences.

URL	`http://example.com`
Full size	1.3 kB
HTML size	1.0 kB

Table A.7: Small page meta data

	First content (s)	Completion (s)	Total size (kB)
Full (theoretical)	N/A	4	1.3
HTML (theoretical)	N/A	3	1.0
The prototype	2	2	0.2 (0.1 compressed)
ELinks via SSH	48	48	N/A
XWeb	13 [1]	14 [1]	N/A [2]

Table A.8: Small page results

[1] This does not include the time taken to establish an initial connection through the Iridium Mail & Web app. This typically took multiple minutes.

[2] Too much variability to get conclusive data. Data varied from 2-17 kB per request, and to establish a connection through the Iridium Mail & Web app typically 50-100 kB was used.