Analysis and Evaluation of Multiplayer Communication Channels in Browser Games

Author: Jan Discart
Promotor: Prof. dr. Wim Lamotte
Co-promotor: Prof. dr. Peter Quax

A thesis presented for the degree of
Master in computer science

Expertise Centre for Digital Media
Hasselt University
Belgium
2013 - 2014

Abstract

Many games that are released today come with some form of multiplayer. For some genre of games, it is only the multiplayer aspect that matters. Games for the web browser have been present for a long time, but multiplayer has always been noticeably absent. With the web browser evolving in becoming an even more versatile and powerful platform for all kinds of multimedia, it is not unthinkable that developers may soon target the web browser as a platform to release their game on. However, little research is done in the potency of the web browser as a platform for playing multiplayer games.

In this thesis, an overview of all possible connection options for multiplayer games in the web browser is composed. These connection options are analysed and evaluated in their properties that are relevant in the context of multiplayer gaming, to determine their applicability for various genres of games that can be played online. They are tested in a series of several round-trip tests in a simple client-server architecture, to measure their resource usage of the system and network, as well as their behaviour when operating under sub-optimal network conditions.

By matching the connections to the requirements of several genres of multiplayer games, it is concluded that the web browser is ready to support even the more demanding games, at least on the network side of things. However, a trade-off that will often have to be made is the one between optimizing bandwidth usage, or latency reduction.

Abstract (Dutch)

Veel games die vandaag de dag uitgebracht worden bevatten steeds meer de mogelijkheid om het spel ook in multiplayer te spelen. Voor sommige genres van games is de multiplayer zelfs het meest belangrijke onderdeel van het spel. Games spelen in de web browser kon al zeer lange tijd, maar multiplayer is een opvallende afwezigheid in veel games. Nu de web browser meer en meer evolueert naar een krachtiger en flexibeler platform voor allerlei soorten van multimedia, is het niet meer ondenkbaar dat game ontwikkelaars binnenkort ook de web browser zullen zien als een mogelijk platform om hun spel op te lanceren. Er is echter nog weinig onderzoek gedaan naar de potentie van de web browser als platform dat gebruikt kan worden voor het spelen van multiplayer games.

In deze thesis wordt een overzicht opgesteld van de verbindingsopties die mogelijk gebruikt zouden kunnnen worden voor multiplayer gaming in de web browser. Deze verbindingsopties worden geanalyseerd en geëvalueerd in hun eigenschappen met betrekking tot multiplayer gaming, om zo hun bruikbaarheid te toetsen aan de noden van de verschillende games. Ze worden op de proef gesteld in een reeks van testen in een eenvoudige client-server architectuur, waarbij hun gebruik van systeem en netwerk resources wordt gemeten, alsook hun gedrag wanneer ze worden blootgesteld aan suboptimale netwerk omstandigheden.

Door het vergelijken van eigenschappen van de verbindingen aan de behoeften van verschillende genres van multiplayer games, wordt geconcludeerd dat de web browser klaar is om zelfs de meer veeleisende multiplayer games te ondersteunen, of toch aan de netwerk zijde. Er moet echter wel steeds een trade-off gemaakt worden tussen het optimaal gebruik maken van de netwerk bandbreedte, of het verminderen van de latency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A web browser may possibly be one of the most widely used pieces of software of all time. It is the visual interface to the greatest source of information: the *Internet* and the *World Wide Web*. In today's lives, the web browser is used for a multitude of things. It is now common to listen to music on the web, watch videos, or perform work-related tasks. It has long surpassed its original purpose, which is to easily search and consume information. It took not long though, before playing games would also be a part of what was expected one could do in a web browser.

Over the years, web browsers have slowly evolved and adjusted to their new tasks, but with *HTML5* to be the next set of web standards and other brave initiatives, development of web applications may shift into a higher gear. Web applications may achieve a higher level of quality and sophistication, while giving rise to new possibilities. One of these new possibilities, is playing multiplayer games as one would play them on his gaming console.

This chapter continues with an overview of how multiplayer games have evolved over the last decade, followed by what technologies would enable multiplayer gaming in the web browser. This chapter concludes with what one can expect to find further in this thesis.

1.1 Evolution of Multiplayer Gaming

The multiplayer games that are played today have come a long way. The internet was not always as accessible as it is now, and the systems on which games are played have not always been this powerful. However, playing games with others was possible ever since the very first games were available. *Pong*, for example, is one of the very first games ever created, featuring multiplayer!

This section will give a more general overview of how multiplayer games

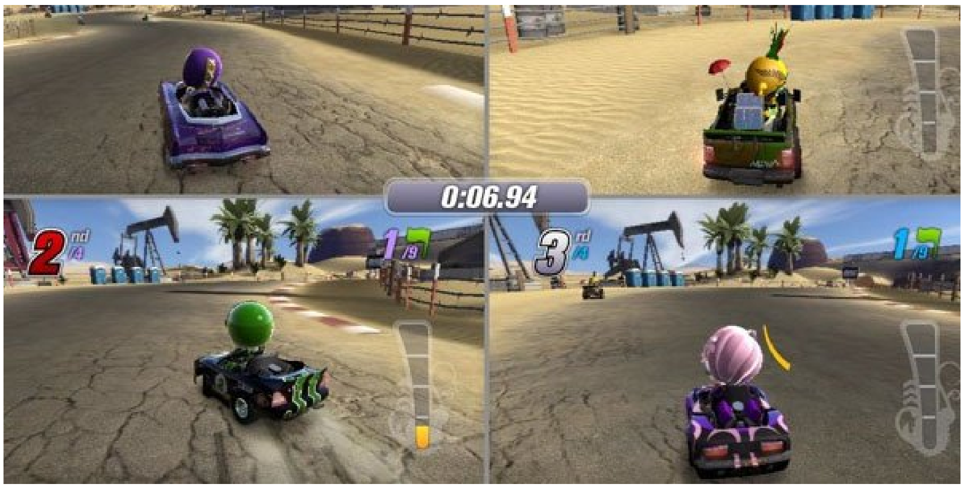


Figure 1.1: Split-screen multiplayer
Playing with 4 players on the same screen in *ModNation Racers*.
Source: `whatculture.com`

have evolved through the years, and how they became the games they are now.

1.1.1 From Local to LAN and the Internet

The first multiplayer games were played locally, in the same room and on the same screen. This is also commonly referred to as *split-screen* multiplayer when each player has his own view on the screen. Local multiplayer is an easy form of multiplayer to manage the game state, because all necessary information of each player is available locally on that system. It is, however, becoming a rare form of multiplayer, due to the increasing complexity of games in terms of visual presentation and game logic, making it difficult for one system to process and display multiple views at once. Some older examples of games that offer local or split-screen multiplayer are *Crash Team Racing*, *Halo*, and *Worms*. More recent examples of local multiplayer games are *Rayman Origins* and *ModNation Racers*.

When personal computers became a common thing in the household, and could be connected to a network, multiplayer over *local area networks* (LAN) could be played. Typically, one player in a LAN environment, hosts a game and other players can connect to this host when they want to join the game. This marked the birth of *LAN parties* were players come together physically, with each player bringing his / her own system and connect them to each other. With multiplayer over LAN, communication between these systems is needed to keep the game state consistent over all players in the game. This communication implies that a certain latency and overhead is created. However, in a proper LAN environment, this communication is usually very fast and very reliable (data errors seldom occur), which makes even naive or inefficient multiplayer algorithms perform good enough. Examples of popular LAN games are *Age of Empires*, *Starcraft* and *Unreal Tournament*.

With the Internet, it becomes possible for players to play games together without requiring them to be physically together. This benefit for the players is, however, a concern for the game developer, because the Internet is a very heterogeneous and unpredictable entity that needs to be taken into account. The Internet introduces delays and overhead that are of a few magnitudes higher compared to games over LAN. Not every player on the Internet has the same connection quality, and not every multiplayer game has the same requirements of the player's connection. Some games require a broadband connection with low latencies and overhead but can tolerate some loss or out-of-order arrival, while other games can cope with some latency but need reliability. The Internet also introduced the problem of scalability. Because the barrier of physical presence is removed, many potential players can play the same game, resulting in games where hundreds of players can join into the same virtual world at the same time. Most multiplayer games are played over the Internet today in every genre available. Popular multiplayer games over the Internet include *World of Warcraft*, the *Call of Duty* and *Battlefield* franchises, *Starcraft II*, *Need for Speed* and many more.

1.1.2 From Terminal to Simulation Clients

The design of a multiplayer game must also specify how much responsibility a client is given in managing the game state. A client system can be designed to be a 'dumb terminal' system that simply sends the player's input to a central game server, and visualizes the current game state. The game state is updated by incoming updates from the server. Here, the client system doesn't perform any game processing, but only visualizes the results. So its only responsibilities are sending input and rendering. The client system can therefore be a very cheap machine in terms of CPU processing power, and in some cases also in GPU processing power when the server sends fully rendered images! It requires, on the other hand, a powerful game server because it takes full responsibility for everything else, and must perform all tasks that could otherwise have been distributed over all participating client machines. The terminal game client is similar to the terminal applications found to remotely access other systems, e.g. *remote desktop* applications. One of the most famous multiplayer games that employs the terminal approach is *Second Life* [67]. Clients regularly receive new, detailed game state updates that can even contain information about how a character is moving its arms and legs.

On the other end of the spectrum, a client system can perform a full simulation of every action going on in the virtual world. The client system receives small updates from the game server or other clients, recomputes its local game state and renders the result. Such an approach puts more load on the client, but relieves the server. Depending on the complexity of the game, a player may need a powerful machine in terms of CPU and GPU processing power as well as a larger pool of memory. Although the load on the client machine is increased significantly compared to the terminal approach, it can give a higher interactive experience because much, if not all, of the game data is available on the client. An action of the player can be computed directly by the player's machine and present a rendered result. The action is also sent to the other clients, who will also compute those results and apply it to their game state.

Most multiplayer games developed now deploy the client as simulation systems. Because computing power and memory are still increasing and becoming cheaper each year, client machines have become powerful enough to take on these tasks. However, a server is still often used as an authoritative entity that validates the game updates so that cheaters can be countered.

1.2 The Web Browser as Gaming Platform

The web browser has long surpassed its original purpose and is now used to do all sorts of things. It already has quite a bit of history being used for games. This section will dig a little deeper in what kinds of games were played in the web browser throughout the years.

1.2.1 The Early Browser Games

The first browser-based multiplayer games were text-based and image-based games that were (are) played over Internet fora. They are very similar to traditional pen-and-paper games where a scenario is written out and players play the game by writing down their actions in turn. The browser did not need to provide any special functionality to make such games playable. Playing them could be done on a regular Internet forum, through standard web pages. *Neopets* for example, was a popular multiplayer game that is played using mainly text and images.

1.2.2 Plug-in Games

However, from the moment that a game developer wanted to create a game that involved real-time visuals and animation, the browser alone proved to be inadequate. Web browsers were not designed to render real-time graphics. A web developer could then resort to plug-ins such as *Java Applets* or *Adobe Flash* to create their real-time games. This spawned web sites that offered a variety of games built with plug-ins. One such popular web sites is *Newgrounds*, offering games and interactive videos. Compared to the numerous single player games, multiplayer games can be considered a rarity. The Java Applet and Adobe Flash plug-ins impose some restrictions on network capabilities that make many multiplayer games hard to develop. Nonetheless, two notable examples of plug-in-based multiplayer games exist.

- *RuneScape* is a full 3D *Massively Multiplayer Online Role Playing Game* (MMORPG) developed using Java Applets.
- *Tibia* is also an MMORPG, but is a top-down 2D game. This game is developed using Adobe Flash.

With the *Unity Webplayer* plug-in, games for browsers had another surge. The Unity development suite is specifically designed to create 3D games, and allows them to be exported to many platforms, including the web. Some games even have their own plug-in. *Quake Live*, for example, is a 3D first-person shooter that was designed to run specifically in the web browser and used its own plug-in design.

1.2.3 Browser-native Games

A major drawback for using plug-ins, is that a developer or a player, is dependent on the plug-in developer to make it available for certain operating systems and web browsers. This means that users of smaller or less popular browsers and platforms are left out.

The lack of standardization and this dependency on third-party developers led to the development of several *HTML5* technologies, to decrease the use of plug-ins for games and other interactive content such as audio, video and chat. Standardization also forces developers of web browsers to implement those technologies or they risk losing market share to those that do. A few of the most exciting additions of HTML5 to the browser for game development are *WebGL*, *WebWorkers* and *WebSockets*.

- WebGL: rendering of fancy 3D graphics and applying shaders. Using WebGL in its pure form can be daunting for new developers. Many wrappers around WebGL exist to ease development, e.g. *Three.js*.

- WebWorkers: offloading of computational-heavy tasks to the background, so that the user interface can remain responsive. WebWorkers allow for basic multi-threaded applications. Communication between WebWorkers and the main user interface thread is done using message passing.
- WebSockets: a persistent connection that allows full duplex and bidirectional communication between client and server.

For browser games in general, each of these new additions are very welcome, but for multiplayer, WebSockets are especially important. A game developer targeting web browsers had very little options in communicating data between clients. Latency and efficiency where a major problem in that department. More details on WebSockets are given in chapter 2.

Just as with plug-ins, several web sites popped up that offer small and simple games, developed entirely with HTML5 technologies. Several HTML5 game development frameworks have also emerged to kick-start its usage. At the time of this writing, however, no major titles are available in HTML5 yet. This could change when large and widely used game development kits such as Unity and *Unreal Engine* also support exporting a game to HTML5. A noteworthy, and recurring, game title that is available in HTML5, is Runescape. This game offers an HTML5-only client next to its Java Applet client.

1.2.4 Beyond HTML5

WebSockets are a good step forwards for making multiplayer games possible in the browser without using plug-ins. However, some real-time applications and games would benefit more from having a connection directly to the other client, either because of security or privacy reasons, or to reduce the latency introduced by relaying a message over a server.

WebRTC is an open source and experimental project by Google that aims to set up peer connections between clients for real-time communication. WebRTC is, at the time of this writing, implemented by the Google Chrome, Mozilla Firefox and Opera browsers, but it can also be implemented in other applications. This allows web browsers to communicate with other applications and systems. More details about WebRTC are found in chapter 2.

Because WebRTC is still an experimental project, its not widely used yet. However, the *Mozilla Developer Network* released an open source multiplayer shooter game, called *Bananabread*, using WebGL for rendering and WebRTC for exchanging game updates. Figure 1.2 shows a screenshot of the game.

Figure 1.2: Bananabread screenshot
A multiplayer shooter game using WebGL and WebRTC.
Source: `developer.mozilla.org`

1.3 This Thesis

When consulting a list of popular HTML5 game development frameworks [11], a common absent feature is support for networking, and thus multiplayer. Now that, with HTML5 WebSockets and the WebRTC experiment, more options are available for game developers to create multiplayer games for the web browser, one must be able to compare these options to make the right decision for their game. This thesis aims to provide an overview of the current options for multiplayer, analyse them in their features and shortcomings, and evaluate them in their performance and behaviour.

This is then brought in the context of developing multiplayer games. Of course, not every type of multiplayer game has the same requirements in regards to networking, so the nature of a game is also taken into account. To summarize this thesis' contributions:

- Analysis of the connection options available for multiplayer browser games
- Evaluation of these connections in their features and shortcomings
- Measurement of their performance in the context of multiplayer browser games

- Discussion of their applicability for different genres of multiplayer games

To summarize, this thesis is a comparative study of the communication channels for browser-based multiplayer game development. The thesis aims to help game developers that are seeking solutions to implement a multi-player aspect for their browser-game, by providing them an overview of the characteristics and performance of these connections, and in what genre of games they are optimally applied.

One can expect to find answers to the following research questions in this thesis:

- What communication channels are available for multiplayer data exchange?
- Which of these connections performs the best for a certain genre of game?
- Can any type of multiplayer game now also be developed and played in the web browser?

1.4 Approach

The remainder of this text starts of with chapter 2 that will provide some more background information about relevant technologies and associated terms. Chapter 3 will discuss the related work and research that has been done in the fields of optimizing multiplayer and web experiences. This chapter will also discuss any similarities and differences found in the fields of multiplayer development for native platforms and web development. Chapter 4 will provide a high-level overview of how the different connections will be tested. More in-depth details about the implementation and software used are found in appendix A. The results of the different tests are shown and analysed in chapter 5, followed by a discussion of the results in chapter 6. Finally, a conclusion is drawn and the research questions and expectations are revisited to see whether multiplayer game development for the web browser is now possible for all types of games.

Chapter 2

Connection Building Blocks

This chapter will dig deeper into the different relevant technologies and terms used further in this thesis. Firstly, real-time communication in general is discussed, where it is used and why it is important for today's web applications and multiplayer games. A short overview of relevant transport protocols is also given to see how real-time communication is made possible. Finally, details about WebSockets, WebRTC and plug-ins are highlighted to better understand their internals and behaviour. Older HTTP techniques are also shallowly explored due to their early contribution of making a dynamic web.

2.1 Real-Time Communication

The term *real-time* is widely used by many applications that require some form of fast or responsive communication, as it is experienced by the user. The type of application has a great influence on what is considered to be real-time. Streaming a video over the internet can be considered real-time communication when the user clicks the "play"-button and the video starts playing a few hundred milliseconds later. The user should not have wait for a long period of time before playback starts, or experience any long playback hiccups. A general chat application is also said to be real-time, but it can typically tolerate a few hundreds of milliseconds to a few seconds of delay. The messages just need to arrive in a reasonable time frame, and in the correct order. Multiplayer games themselves come in various flavours, and each have their communication requirements to make it feel real-time.

A real-time communication protocol exists, aptly named *Real-time Transport Protocol* (RTP), developed to facilitate real-time communication over a network. Next to RTP, another protocol is used to send control information about the real-time data, called *RTP Control Protocol* (RTCP). The data carried by RTCP contains control and statistical information about the data streams and the receiver's connection, so that the sender may adjust the stream accordingly. The RTP and RTCP protocols are, however, mainly designed to stream audio and video content, and less for arbitrary data such as positional information of a player's avatar. This limits its applicability for real-time multiplayer games, although real-time audio communication can be found in an increasing number of multiplayer games.

Because real-time video and audio applications commonly have the same requirements in how data is communicated, the RTP and RTCP protocols are generally good candidates to build such an application. Due to the variety in multiplayer games however, no single communication protocol is perfect for every type of game. These communication requirements come in the form of reliability of message delivery and message ordering. Often, to stay within the timing and bandwidth limits, trade-offs have to be made in reliability and ordering. They are further elaborated on in sections 2.1.1 and 2.1.2. Another note is that games, and other real-time applications in general, can use more than one communication channel to transfer all relevant data. For example in multiplayer games, there is often the option to chat with other players while playing. In many cases, this chat traffic will go over a separate channel while other game data goes over a different channel with its specific requirements.

2.1.1 Reliable versus Unreliable

The reliability of a communication protocol defines how the protocol copes with lost or delayed data. A reliable protocol will make sure that the recipient will eventually get the data that was sent by the sender. When a packet gets lost, or is held up somewhere in the network, the protocol will take notice in some way and try to resend the data that is lost or delayed. An unreliable protocol on the other hand, does not provide this guarantee, and will not attempt to retransmit the data.

Although reliability of the communication channel is usually a desired feature, it also implies that the channel must gain some knowledge about whether the data is actually arriving or not at the other end. Usually this is implemented using sequence numbers that are sent along with each packet. Whenever a receiver receives a packet, it can know by looking at the sequence number, which packets it has received yet, and which are late or dropped. But up until now, only the receiver has this knowledge. To let the sender know about late or lost packets, the receiver can send acknowledgements (*ACKs*) to confirm an arrival. These ACKs can be sent separately or can piggyback on a returning message. This system of ACKs also requires some bytes on the way back of the communication channel.

Besides requiring some extra bytes, a reliable protocol can also cause severe latencies. This usually happens when packets do get lost or run very late. Many protocols don't just keep sending data and, at the end of the session, check which data didn't make it. They usually operate through some *sliding window* protocol [64] that limits the amount of unacknowledged packets. When a packet does get lost or is severely late, then the sliding window will eventually stop sliding forward, and the sender must wait with sending new data. He must re-transmit the assumed-to-be-lost data until it receives an ACK for that data. It is this system that can cause the severe latencies. Note that, also ACKs can get lost or run late. So, although the data did arrive at the other end, the sender may decide to send the data again because it never received an ACK for that data. For certain real-time applications, these latencies mean that new data is already out-of-date when it finally reaches its destination!

For applications that can't cope with these potential delays, or can't afford the extra overhead, an unreliable protocol can be used. The sender then relies on the network to correctly deliver the packet, but no guarantees are made. Unreliable protocols don't need to have the protocol overhead of sequence numbers and sending acknowledgements, as well as no sliding window protocol that may limit sending new data due to lost packets. However, the sender can't be sure that the receiver actually received all of its data. When the receiver has some knowledge about the type of traffic he's receiving, he can provide feedback to the sender to adjust the stream of packets. This principle is applied by the RTCP protocol, in conjunction with RTP, by providing statistical data to the sender.

Reliably sending data also means that data arrives error-free. Many transport protocols also include the means for detecting errors introduced during transmission of the data. This comes in the form of checksums. Whenever the data does not match the calculated checksum provided in the packet, one can conclude that somewhere in the network the data (or the checksum) is altered. The protocol can then decide to attempt to repair the packet, if the protocol also supports error correction, or discard the packet. When discarding is chosen on a reliable protocol, the data is considered to be lost, and a re-transmission will be initiated. For unreliable protocols, this means that the data is lost.

Whether reliable or unreliable communication is used, usually depends on the type of application or the type of game, and in what network environment it is designed to operate in. When the real-time application is designed for LAN environments, then, a reliable communication protocol is certainly an option. Such links are usually very fast and stable, but if a packet does get lost or corrupted, it can be detected and retransmitted rapidly. When moving on to less predictable network environments such as MAN or WAN, then the nature of the application has a bigger influence on what protocol should be used. When the relevance of data has a very short time span, or when losing data is not as big of an issue compared to stalling it, then an unreliable protocol should be considered.

2.1.2 Ordered versus Unordered

Packets sent by the sender don't necessarily all take the same path to the receiver. Networks can react to congestion by altering their routing tables so that incoming packets may take a different turn. This may cause packets sent earlier get stuck in traffic, while packets sent later may have already arrived at the receiver. The choice of protocol also has an influence on how it handles out-of-order packet arrivals.

Ordered communication protocols guarantee that the order in which packets are sent, is also preserved when the application at the receiver receives this data. Although packets may still arrive out-of-order at the receiver, even if an ordered protocol is used, the protocol can enforce that data is being delivered in-order to the application-level above. One way to do this is by buffering the data and wait for earlier data to arrive so that data is ensured to be delivered in-order. This implies again that the protocol, just as with reliable communication, works with sequence numbers or timestamps because it must know how to order the incoming packets. However, reliability is not necessary for in-order delivery. For example, during a real-time video conversation, video frames arriving out-of-order can be buffered for a little while. When earlier video frames don't arrive on time for playback, they can be skipped. Then, the frames that did arrive on time, are played in their correct order when their playback time has arrived. If an out-of-date video frame does eventually arrive, it can be simply discarded because it is no longer relevant.

Enforcing both reliability and in-order delivery can cause latencies at the receiver-side, even if a large part of the data has already been received. When an early packet gets lost or is late, but later data is already received, then this data must wait to be delivered to the application until this early packet finally arrives.

For unordered communication, things are much simpler: data can be just delivered to the application as soon as it arrives. This also means that they don't require a sequence number present in the packet, saving some bytes of overhead. The application however, must be able to cope with the potential out-of-order delivery of data.

Ordered message delivery is often a desired feature for multiplayer games because in many cases, the order in which player actions are executed has an influence on the outcome of those actions. For example, take a shooter game where the player shoots at another player and jumps in the air. If the player shot first and jumped afterwards, then the bullet flies low and hits the target. Consider that each action, shooting and jumping, is sent to another player but arrive in a reversed order, then the player jumps first and shoots afterwards. This may result in the bullet flying over the target. (TODO: The *latency compensation* technique [6], developed at *Valve Corporation* and applied in several shooter games, can overcome this by "looking back in time" to see whether an out-of-order action was valid or not.)

2.1.3 Transport Protocols and Their Characteristics

The above described communication principles of reliability and ordering are commonly used to compare several communication protocols. For real-time communication applications, they are one of the most important aspects for (not) choosing a certain communication protocol. In this section, some common transport protocols are briefly introduced with their takes on reliability and ordering. Table 2.1 displays a brief summary of the different protocols

TCP

The *Transport Control Protocol* (TCP) is probably the most widely used protocol for transporting data over the internet. It is used in a variety of networking applications including: web browsing, sending email, chatting, and file transfer. TCP is a *connection-oriented* protocol. This means that both endpoints on the network inform and agree with each other that they are willing to send / receive data from each other. Once this agreement is made, the TCP connection is open and data can flow over the connection. This flowing of data could almost be interpreted literally. TCP sends data as a continuous stream of bytes that arrive in a **reliable** and **ordered** way. The application can then read the data from this stream. A drawback from this streaming approach is that it does not provide message boundaries. This has to be resolved on application-level by sending message delimiters or length prefixes along with each message. These extra delimiters or length prefixes also increase bandwidth usage with a few bytes. Attempts have been made to remove the strict ordering imposed by TCP. [40]

TCP also has a checksum field that is used to check the data for accidental errors that might have been introduced somewhere along the way due to a bad signal or faulty router. When the checksum does not match, the data is discarded and regarded as lost. A new copy of the data is then expected to arrive soon when the sender detects that no ACK for the data is received.

A TCP socket on a machine can also be configured to use *Nagle*'s algorithm. This algorithm stalls small amounts of data for a little while to see if more data becomes available to fill up the TCP buffer for efficient transmission. This is very similar to update aggregation discussed in section 3.2.2. For multiplayer games, Nagle's algorithm is usually not desired when sending many, but small, update packets. The algorithm will stall the updates for a while, which increases the latency for the updates that are stalled first.

UDP

With everything that TCP offers, the *User Datagram Protocol* (UDP) does not. It can be considered to be the opposite transport protocol of TCP in many ways. First of all, UDP doesn't require a connection setup and agreement step like TCP. All two endpoints must do is set up an UDP socket and listen on them for incoming data. Because no agreement is made between two endpoints, one can send to and receive data from anyone addressing the socket appropriately.

Secondly, UDP is a *message-oriented* protocol. In contrast to TCP where no clear distinction is made between separate messages due to its streaming nature, UDP makes the distinction between messages very clear. Sending one message through an UDP socket at the sender will result in one message received at the receiver, if the packet is not lost somewhere along the way. Messages over UDP are sent **unreliably** and **unordered**. The UDP header does not have any notion of a sequence number. The UDP header is also very lightweight compared the TCP header.

UDP does have a checksum field to detect errors that might have slipped into the packet data. However, checking the checksum and data for errors is optional in *IPv4*. Then the checksum is filled with all zeroes. In *IPv6*, the checksum is mandatory. So although UDP is unreliable in delivery, it does provide the means to check for reliability in transmission.

Even though UDP on its own does not provide reliability or ordering, it is often chosen as a protocol to build custom protocols with, due to it being so lightweight. These custom protocols can then implement a mechanism to achieve ordering and / or reliability in their own way, with custom reordering and re-transmission rules. For example RTP, mentioned earlier, is not a transport protocol and thus is implemented in conjunction with a transport protocol. Usually, UDP is chosen for the job.

SCTP

The *Stream Control Transmission Protocol* (SCTP) is the youngest protocol of the three transport protocols highlighted in this section. SCTP is a protocol that borrows a lot of features form TCP but takes the message-oriented approach from UDP. Just like TCP, it requires a connection to be negotiated between two endpoints. In this negotiation, SCTP also discusses how many streams will be used. The protocol allows to multiplex multiple streams of data in a single SCTP connection. In contrast, for each stream of data in TCP, a separate connection is required or one must implement support for multiple streams on a higher level than TCP. Although the term stream is used here, it does not have the same interpretation as in TCP, because SCTP works with message boundaries.

The basic design of SCTP makes it a **reliable** protocol with the option to send messages either **ordered** or **unordered**. Whether a message is delivered in-order or out-of-order can be toggled using a bit-flag in the header associated with the message. An extension for SCTP is defined that also allows the protocol to transport data **unreliably** [59]. To be more precise, this extension adds *partial reliability* to SCTP. This means that the user / developer can configure how long, or how many attempts can be made to retransmit a lost packet. If these values are set to 0, then packets are sent truly unreliable.

However, only the sender knows about this unreliability for a piece of data. When the receiver notices that some data is missing, it will let the sender know through *selective acknowledgement*s (SACKs). These SACKs contain a series of sequence numbers that have been received, and gaps of sequence numbers that haven't been received. When the sender receives a SACK and notices that data is missing, it will check whether this data was configured to be sent reliably or not. If it was configured to be reliable, or partially reliable, then the data is sent again. Otherwise, a new packet is generated to let the receiver know that it can forget about the data. The receiver can 'forget' about this data by advancing its *transmission sequence number* (TSN) that corresponds to the value received in the packet generated by the sender.

In the main SCTP header, a 32-bit checksum field is used to check for transmission errors. This checksum is twice as large as in UDP and TCP.

Supporting all these features has a toll on the overhead created for the data. Due to the multiplexing feature, many sub-headers are introduced on top of the main header that require several more bytes for each chunk of data in a stream. Currently, SCTP also faces the problem that it is not supported natively on all major operating systems. Most notably, the *Windows* operating system does not have native support [8], making it hard for SCTP to really take off as a widely used protocol.

	Reliability	**Ordering**	**Error-free**
TCP	Yes	Yes	Yes
UDP	No / On higher level	No / On higher level	Optional (IPv4) / Yes (Ipv6)
SCTP	Configurable	Configurable	Yes

Table 2.1: Transport protocol overview.

2.2 HTTP-based Techniques

The HTTP-based techniques discussed here are a collective of techniques used to create dynamic content for web pages by (ab)using HTTP connections. These techniques are relevant in the current context because they can be considered to be the predecessors for real-time communication on the web. They are also still relevant due to legacy reasons: old web browsers are still widely used that don't support newer techniques, e.g. *Internet Explorer 6*. Now that the web is rapidly evolving towards more rich and interactive content, they quickly show their inefficiencies.

Many of these techniques are known under the umbrella-term *Comet* [62]. Initially, they were used to let the server push new data to the browser to update the content of the web page without having to reload the entire page. Later, the other direction, from client to server, became possible.

2.2.1 HTTP Polling

HTTP polling is the most primitive of the HTTP-based techniques. If the web page contains some dynamic content, the browser would create an HTTP request to the server and ask whether it had some new information available, hence the name of the technique. For each such a request, a new HTTP connection is set up. When the server had some new information, it would immediately send it over this connection. However, more often than not, the server doesn't have any new information and thus an empty response is sent back. When a response was received, empty or not, the connection would be closed and cleared. These polling requests could be initiated periodically or based on an event, e.g. a user pressing a button.

Its immediately clear that polling like this wastes a lot of effort and bandwidth when no new information is available, considering the entire HTTP connection setup and release process. Figure 2.1 shows a possible sequence of polling events using the HTTP polling scheme.

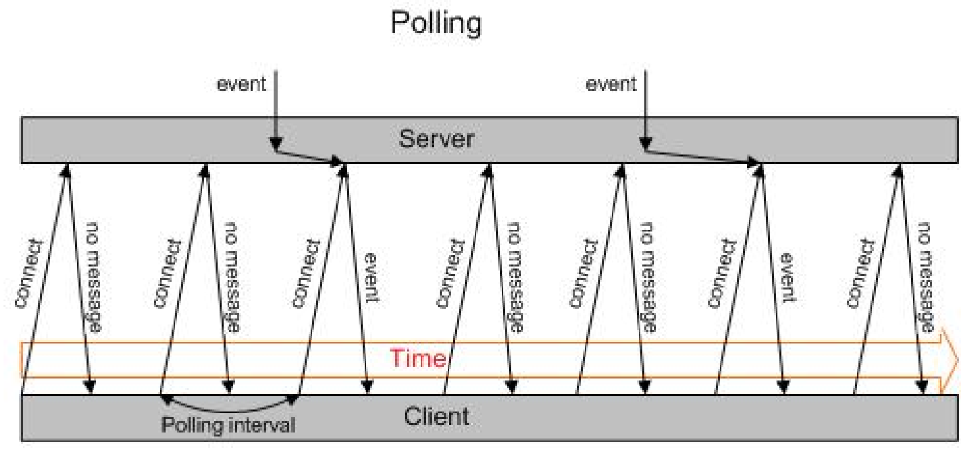


Figure 2.1: HTTP polling sequence
Source: `http://www.leggetter.co.uk`

2.2.2 Long Polling

To prevent the server from immediately returning an empty response, the HTTP connection can be held open for a limited period of time: *long polling*. Whenever, during this time period, some new information becomes available, the server then responds with this information, completes the request and closes the connection, after which a new one is immediately set up again. However, the connection here too is not held open indefinitely. When the connection almost reaches its timeout, then an empty response is sent back.

This achieves some efficiency improvement in terms of bandwidth and latency compared to regular HTTP polling. However, long polling increases the workload of the server with processes that are busy waiting. Figure 2.2 displays a sequence of long polling connections and events.

2.2.3 HTTP Streaming

Above implementations did not make use of *long-lived* HTTP connections that were introduced in HTTP version 1.1. Such a connection is only time bound, not both time bound and data bound as is the case with long polling. This allows the server to stream some data to the client for a period of time. After the time period expires, the connection is closed. How fast this connection is closed depends on the implementation. Usually it is the server that initiates this shut-down process. For example, the *Apache 2.2* web server already closes this connection after a default value of 5 seconds! [22]

There are a couple of reasons why this connection isn't just kept open. Firstly, all incoming data over a long-lived connection is accumulated and

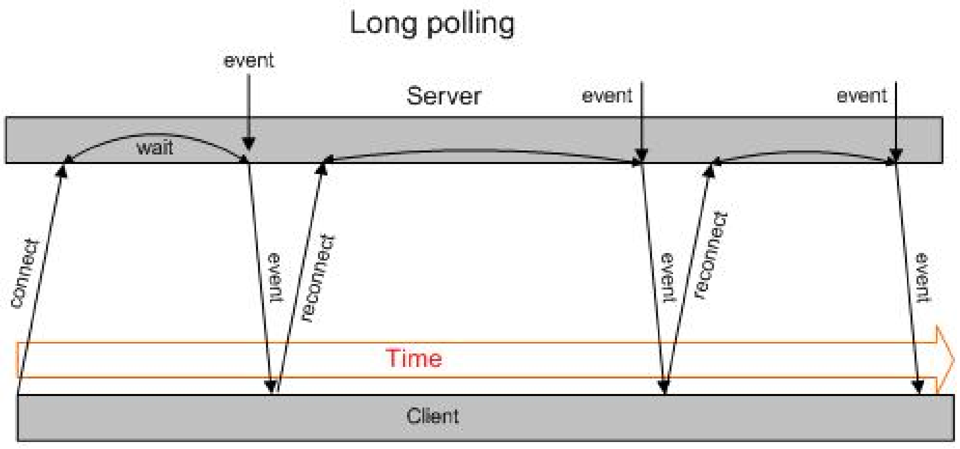


Figure 2.2: Long polling sequence
Source: `http://www.leggetter.co.uk`

stored in the browser for the remainder of the connection duration. Streaming lots of data over a single very long-lived connection may result in memory problems at the client. Secondly is the support in browsers. Not every browser supports the necessary function calls to set up or maintain a long-lived connection. Finally, there is the increase in resource usage on the server-side, which is the same remark as with long polling. Maintaining a large amount of long-lived HTTP connections increases CPU and memory usage due to the server being occupied with processes that are busy waiting. [1] Figure 2.3 illustrates the HTTP streaming process.

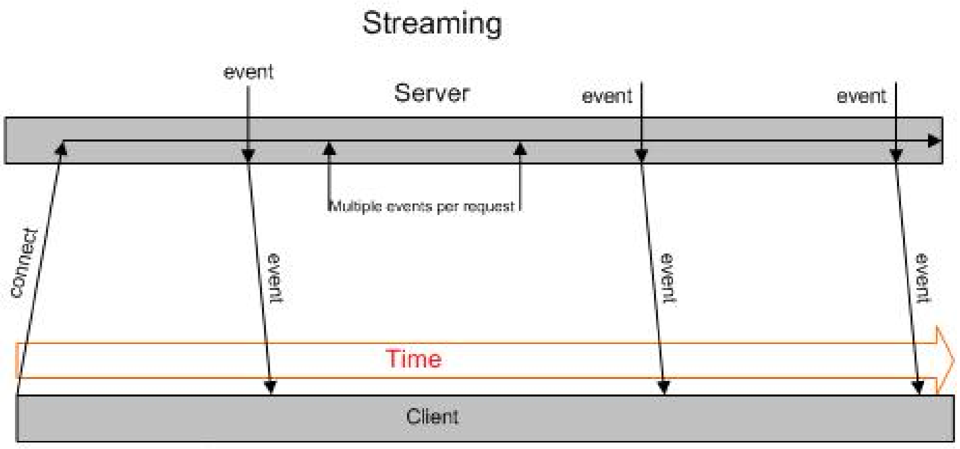


Figure 2.3: HTTP Streaming sequence
Source: `http://www.leggetter.co.uk`

2.2.4 BOSH

Bidirectional-stream over synchronous HTTP (BOSH) takes long polling a step further by adding a second connection that is used to send data over to the server. This achieves a bidirectional communication system. Initially, BOSH sets up the first connection as a long polling connection. Whenever the client has some data to send, the second connection is set up and made ready for transmission. When the server detects that this second channel is being opened, he sends an empty response back on the first channel and closes it. From the moment the data is received at the server on this second channel, the server will then use this channel as its long polling connection. The two long polling connections effectively switch roles. This process repeats itself whenever the client has some data to send. [44]

Although BOSH keeps the problems associated with long-lived HTTP connections at bay, it again has to set up a new connection every time new data at the client or server becomes available. For data streams containing only small amounts of information, or that are latency sensitive, this system still falls short. The connection setup time whenever the client wishes to send data can take up considerable time, while the communication overhead can overshadow the data actually sent. A better solution would be to have a persistent, bidirectional and full-duplex connection between client and server as is done in native implementations.

2.3 WebSockets

Before WebSockets were widely used, communication with the server to update content or providing the server with new information was a difficult process requiring many resources from the client, server and network. A WebSocket connection can establish a persistent, full-duplex bidirectional communication channel between client and server.

Another advantage over previous methods, is that WebSockets are standardized and are supported by all recent versions of popular and unpopular browsers. However, even though WebSockets are widely supported, they don't make the HTTP-based techniques obsolete. Many older versions of popular browsers don't support WebSockets, but still have a large market share in usage. Another reason is that some older proxies don't recognize the WebSocket connection set up process, and drop the packet or return the wrong result. In those cases, HTTP-based techniques are still used as a fallback mechanism.

WebSockets consists of two separate parts: the WebSocket protocol and WebSocket API. Each of them is further detailed below.

2.3.1 WebSocket Protocol

The WebSocket protocol is a very thin layer laid on top of the regular TCP protocol (minimum 2 bytes, maximum 13 bytes), but unlike TCP's stream-based behaviour, the WebSocket protocol header is used to make it a message-based protocol. Using TCP as the underlying protocol, implies that a WebSocket also behaves like TCP, meaning that data is sent reliably and ordered using TCP's retransmission and sequencing mechanisms. The data is also delivered error-free using TCP's checksum field.

A WebSocket's header can vary in size depending on the length of the payload. If a payload is less than `126` bytes, only `7` bits are used for the *payload length* header field. If the payload equals or is larger than `126` bytes, some extra bytes are padded to the end of the header which extends this payload length header field. This field can be further extended, to a maximum of `8` bytes, to carry very large messages.

The WebSocket connection is always initiated by the client. When a client wishes to create a WebSocket connection, he sends out an HTTP upgrade request to the server. The server interprets this request as a request to change the protocol used for communication. In this case, the WebSocket protocol. This upgrade request also contains extra information about which version of the protocol to use as well as the appropriate handshake values. Once the server responded positively to the request, the connection is ready for data transmission. A server can respond negatively, for example, when it does not support the requested WebSocket version, or WebSockets in general. An example of the content of an HTTP upgrade request sent to the server is shown below.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
...
Sec-WebSocket-Version: 25
```

Listing 2.1: WebSocket HTTP upgrade request

Once the connection is established, data can flow from both client and server, without the limitations of HTTP-based techniques discussed earlier. Whenever data goes from the client to the server, the WebSocket specification dictates that a client masks his data using a new, unpredictable, *masking key* for each packet. Unmasked data should not be accepted by a server. A server on the other hand is expected to send unmasked frames to the client. Because the initial packet of the WebSocket connection set up process is interpreted as HTTP, faulty proxies may keep interpreting subsequent data on that connection as HTTP data, and return cached results instead of passing it through to the server. [19] Masking the data does not provide any security to eavesdroppers or altering of data by third parties, because the key to unmask the data is present in the packet itself.

The protocol also defines some values that indicate the type of the payload being transported. Besides `control`, `ping` or `pong` packet types, the `text frame` and `binary frame` codes define how the other side must interpret the data. These values are specifically designed for the JavaScript API that either allows to send data as a string, or as a blob of binary data.

2.3.2 WebSocket API

The WebSocket API highlighted here is the one that is exposed in the browser, at client-side. The API is very limited in functionality, and allows no customization of the WebSocket protocol and underlying TCP connection settings, e.g. toggling on / off Nagle's algorithm. That said, the API is simple in usage and clearly formulated. Only very few steps are necessary to start sending data between the client and server. These steps are listed below, as well as a JavaScript code fragment to illustrate the process. Next to sending and receiving data, the web developer can check on the status of the WebSocket.

1. Create WebSocket with the server address
2. Wait for the WebSocket to open
3. Start sending and receiving data

```javascript
var websocket = new WebSocket("ws://server.example.com");

websocket.onopen = function() {
  console.log("WebSocket is open.");
  websocket.send("Hello server");
};

websocket.onmessage = function(event) {
  if (typeof event.data === "string") {
    console.log("Received string message from server: " + event
       .data);
  } else {
    console.log("Received blob message from server.");
  }
};
```

Listing 2.2: WebSocket connection setup example

2.4 WebRTC

WebRTC is a collection of technologies and APIs for the web browser to set up peer-to-peer connections and use them for real-time applications. The APIs include access to a user's webcam and microphone to create video and audio conversation applications on the web. Besides audio and video, channels for regular data can also be established to create a wide range of real-time data-based applications, including multiplayer games and file-sharing. These regular data channels can be configured in reliability and ordering. But it doesn't have to stop at purely web applications. WebRTC could also be used to connect the browser to other kinds of clients, such as native clients implementing WebRTC, but also *SIP* [55] or *Jingle* [36] clients, or even a regular phone. If the appropriate servers and / or gateways are placed in the network, such applications could be built.

WebRTC removes the client-server architectural barrier that has been in place since the beginning of the web. This gives more freedom in creating applications that require more than the client-server model. But even though WebRTC works through peer connections, does not mean that client-server can't be used. A server incorporating WebRTC connectivity options can act as just another peer. This allows for more complex architectures, mixing client-server and peer-to-peer.

To achieve all of these features and functionality, a large stack of technologies and protocols are used to support it on as many browsers and platforms as possible. Currently, WebRTC is still in an experimental phase and is only supported by a limited number of web browsers, including Google Chrome (incl. on Android), Mozilla Firefox and Opera.

The technologies and techniques behind WebRTC are further detailed in the following sections. Much of this information is gathered from a book by Johnston and Burnett. [29]

2.4.1 Protocols and Services

WebRTC uses a large amount of protocols to establish a peer connection and to support that connection. Some protocols have been introduced already in section 2.1.3. These are not further explained, except for their contribution in the relevant areas of WebRTC. One major difference between the protocols from section 2.1.3 and those introduced here, is that these are not transport protocols, but rather session, security, or application-level protocols.

STUN

Session Traversal Utilities for NAT (STUN) is a service used for discovering a user's public address whenever he's behind a home- or company-router that implements *Network Address Translation* (NAT). A server running a STUN service will simply reply to requests with the public address information found in that request's IP header. Sending such a reply back will let the user know which public IP address he has, and which port has been opened on the outer-most router of the user's network. It's important to note that a user may be behind several NAT routers, but only the address and port number of the outer-most router of the network will be known. These STUN request-responses create NAT bindings in the router(s) as they travel through the network. As long as these bindings are available, addressing the outer-most router from outside the network with the correct port address should cascade the data through the network to the correct system.

TURN

Traversal Using Relay around NAT (TURN) is a service that incorporates STUN, and also offers a relay function for media streams. This media relay function is necessary when peers can't connect to each other directly. This can happen when a firewall or router is configured very restrictively in letting certain types of traffic pass, e.g. UDP traffic. Then, a peer can opt to use a server that offers the TURN service which will accept the incoming data streams from the user and relay it to the correct receiver. Using a TURN server essentially goes back to a client-server architecture. The advantage here however, is that a TURN server can be a different device than the web server, which doesn't overload the web server with media streams.

ICE

From the moment users are about to establish a connection with each other, they must know on which address and port number they can reach each other. The *Interactive Connectivity Establishment* (ICE) is a process that will try to discover as many reachability options for the user. ICE tries to gather several pairs of address and port numbers on which the user is reachable for the other peer. First, it queries the own system's NIC for the assigned IP address and a free port number. Next, STUN and TURN servers are queried to detect whether the user is behind a router implementing NAT, and which public IP address and port number are associated with the user. If the user is behind NAT, then querying a STUN or TURN server will add an entry in the NAT table for the user, creating a small 'hole'.

Once the available options are gathered, they are sent to the other user. The remote user also performs these same actions. Upon receiving the gathered address-port pairs from the remote user, ICE will start *hole punching* a connection through NAT to reach the remote user using his address-port information. If a connection is established, it is important to keep probing the other user regularly, to prevent NAT from closing the hole again if no activity is detected. If no direct connection could be established, then a TURN server could be used for relaying the data.

SDP

The *Session Description Protocol* (SDP) is a protocol that defines a format to describe a real-time communication session for a user. Such a session description contains information about the contact information of participants, which type of media sent (audio, video, text), which codecs are used, which protocol is used, ...

A session description in WebRTC is sent between users to announce their type of output and what format will be used. Then the receiver can prepare itself and allocate the necessary resources on the system.

SRTP

The *Secure Real-time Transport Protocol* (SRTP) is a secure variant of RTP. Although the name of the protocol suggests it is a transport protocol, it is considered an application-level protocol that is still dependent on true transport protocols such as TCP or UDP to get the data delivered to the correct application on the receiving end. RTP is a protocol that defines a standardized packet format to deliver audio and video data in a streaming-based manner to its receivers. RTP can be extended with profiles, of which SRTP is one of them. The SRTP profile provides encryption on the audio and video data, as well as message authentication, integrity protection an replay protection.

RTP also has a sister-protocol, RTCP, that provides statistical information about the data arriving at the receiver(s). This allows the sender to adjust its streaming behaviour according to the reported information. To also protect this feedback information, the *Secure RTP Control Protocol* (SRTCP) is defined, providing the same protection as in SRTP.

Because RTP is mainly defined for real-time audio and video, it is used in WebRTC to transport the audio and video streams. Using SRTP instead of RTP protects the data from alteration and eavesdropping by third parties in the network.

DTLS

Transport Layer Security (TLS), known as the successor to *Secure Sockets Layer* (SSL), is a thin layer on top of the transport protocol that provides protection against eavesdropping on the data being sent over a connection. It provides confidentiality and authentication services by using encryption and certificates to secure the line. *Datagram TLS* (DTLS) is a version of

TLS designed to specifically run on UDP. DTLS is set up by WebRTC for every type of communication channel. For audio and video, DTLS is used to generate and manage the security keys for SRTP, while for the regular data channels, DTLS is part of the protocol stack. This makes all data sent by WebRTC secure by default, making it hard for eavesdropping on the conversations going on.

2.4.2 Protocol Stacks

The two main protocol stacks used for sending data are highlighted here. For audio and video, the same protocol stack is used, while for arbitrary data, a different stack is used.

For audio and video transmission, either an *SRTP/TCP* or *SRTP/UDP* stack is used. WebRTC prefers the latter stack due to UDP's properties. But if no connection can be established using UDP because of a restrictive firewall or NAT, then a TCP connection can be tried. SRTP is always used to secure the audio and video content transported.

For data, only one stack is used. Here, two transport protocols are used on top of each other. UDP is used as the base transport layer because almost every operating system with a network stack supports UDP. The second transport protocol is SCTP, running at the top. SCTP is difficult to use as the base transport layer because of its lack of native support in some operating systems. The browser implements SCTP at the application-level for its support of multiplexing several streams of data. This allows a developer to create several data streams and multiplex them over a single WebRTC connection, as well as configuration of reliability and ordering for individual streams of data. In between UDP and SCTP, a DTLS layer secures all of the data transported by SCTP. Figure 2.4 shows the protocol stack used for data channels.

2.4.3 Signaling

An important part of WebRTC is how the session description that contains configuration information for connection establishment, is communicated with the other peer. The communication channel over which this information is exchanged is referred to as the *signaling channel*. The WebRTC specification, however, does not specify what kind of communication channel this should be. This is left for the developer to decide on how he wishes to approach this. This can almost be any form of communication available, and even very primitive ones:

- A dedicated communication channel (WebSocket, HTTP-based connection, ...) to a server bringing several peers together, or

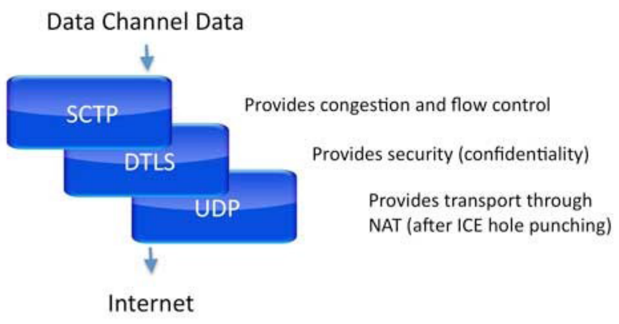

Data Channel Data
SCTP
Provides congestion and flow control
DTLS
Provides security (confidentiality)
UDP
Provides transport through NAT (after ICE hole punching)
Internet

Figure 2.4: Data channel protocol stack
Source: *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web* [29]

- An already existing WebRTC data channel to renegotiate a connection, or
- An e-mail or a text-based chat application to copy-paste the session description, or
- Communicating the session description by phone, or
- Any other form of communication one can think of.

Although sending the session description is a signaling channel's most important task, it also serves to identify and authenticate peers to each other, control the media session in progress, and to resolve glare resolution when both peers try to change the session in progress at the same time. Whenever security and privacy-sensitive data will be exchanged over the WebRTC session, it is the developer's responsibility to make sure the signaling channel is also secured to protect session description data. Intercepting and decoding the data sent over the signaling channel may allow a third party to eavesdrop or perform malicious actions.

2.4.4 The Peer Connection API

Once peers are aware of one another, a peer connection can be set up between them. The connection set up requires several more steps compared to WebSockets, due to the variability in the connection configuration. This also makes the API more extensive compared to the one for WebSockets. The steps one must take depend on who is the caller and who is the callee. They are summed up briefly below, followed by a code example to set up a regular data channel connection as seen from the caller-side.

For the peer initiating the call, assuming the signaling channel is already up and running:

1. Create a peer connection object configured with STUN and TURN server addresses for querying the public address of the system.
2. Locally allocate resources for the media streams the caller wishes to send out. This includes one or more audio, video and data channels.
3. Create a session description offer for the callee, and set this description as the local description.
4. Send the gathered ICE candidates (results from querying STUN and TURN servers) and the local session description to the callee over the signaling channel.
5. Receive ICE candidates from the callee and its answered session description. Set the received session description as the remote session description.
6. Allocate resources for receiving the streams the callee is planning on sending out.
7. Wait for the connections to open up.
8. Start sending and receiving data.

For the peer waiting to receive the call, assuming the signaling channel is already up and running:

1. Wait for a session description offer from the caller and its gathered ICE candidates.
2. Create a peer connection object configured with STUN and TURN server addresses for querying the public address of the system.
3. Locally allocate resources for the media streams the caller wishes to send out. This includes one or more audio, video and data channels.
4. Apply the session description from the caller as the remote description and allocate resources for the incoming media streams.
5. Create a session description answer for the caller, set this description as the local description.
6. Send the gathered ICE candidates and the session description answer to the caller over the signaling channel.
7. Wait for the connections to open up.
8. Start sending and receiving data.

The following piece of JavaScript code is a short example of a client that wishes to setup a reliable data channel with another peer. For the callee, the code is very similar, only that the callee must wait for an incoming session description and that some actions are done in a different order, as described above in the enumerations.

```javascript

// It is assumed here that the signaling channel is available
    and open, e.g. a WebSocket connection
var signalingChannel;

var peerConnection = new RTCPeerConnection(
  { iceServers: [{url : "stun:stunserver.example.com:19302"}]
      },
  { optional: [] }
);

peerConnection.onicecandidate = function(event) {
  var candidate = event.candidate;
  signalingChannel.send(JSON.stringify({
    "type" : "ice",
    "sdp" : {
      'candidate' : candidate.candidate,
      'sdpMid' : candidate.sdpMid,
      'sdpMLineIndex' : candidate.sdpMLineIndex
    }
  }));
};

var sendOffer = function(offer) {
  signalingChannel.send(JSON.stringify({
    "type"  : offer.type,
    "sdp" : offer.sdp
  }));
};

var setLocalDescription = function(desc) {
  peerConnection.setLocalDescription(new RTCSessionDescription(
      desc), sendOffer(desc));
};

var createOffer = function() {
  peerConnection.createOffer(setLocalDescription);
};

var createReliableDataChannel = function() {

  var reliableConfig = {
    "ordered": true,
    "maxRetransmits": 10
  }

  var channel = peerConnection.createDataChannel(label,
     reliableConfig);

  channel.onopen = function() {
    channel.send("Hello peer");
  };

  channel.onmessage = function(event) {
    if (typeof event.data === "string") {
      console.log("Received string message from other peer: " +
           event.data);
    } else {
      console.log("Received blob message from other peer.");
    }
  };

  createOffer();
};

var setRemoteDescription = function(desc) {
  peerConnection.setRemoteDescription(new RTCSessionDescription
     (desc));
};

signalingChannel.onmessage = function(event) {
  var data = JSON.parse(event.data);
  if (data.type === "answer") {
    setRemoteDescription(data);
  } else if (data.type === "ice") {
    var candidate = new RTCIceCandidate(data.sdp.candidate);
    peerConnection.addIceCandidate(candidate);
  }
};

createReliableDataChannel();
```

Listing 2.3: WebRTC data channel connection setup example

2.5 Plug-ins

Browser plug-ins, sometimes also called extensions or add-ons, are containers where small programs loaded by the browser can run in a closed environment. Plug-ins can provide a fixed set of functionality, such as a media player to play videos encoded with a special codec, or a virus scanner. Others provide a entire programming environment that allow to develop an application that will run inside the plug-in container of the browser.

Plug-ins usually perform tasks that are difficult to achieve in regular JavaScript code, such as image manipulation, 3D animations or games. However, with HTML5, some of these tasks become much more manage-able to do with JavaScript, and plug-ins start to see a drop in their usage. For example, *YouTube* used the Adobe Flash Player plug-in to display their videos on their web site, but is now gradually being replaced by HTML5 video. Nonetheless, plug-ins are still a major part of the web, and are worth introducing them here.

To run a plug-in application in the browser, it is usually also required to have a program installed on the client machine that is able to execute the plug-in application next to the browser. This program is often the target of exploits and malicious attacks to bypass the browser's security restrictions.

The plug-ins introduced below are all plug-ins that offer a programming environment and give access to sockets or some other way to send and receive data from a network.

2.5.1 Java Applets

A *Java applet* is small application written in Java that can be run in the browser. In requires the *Java Virtual Machine* to be installed on the client machine to run. Almost all off the Java functionality can be directly used in the applet, with the exception of access to the file system and network.

Access to the file system and network are, however, not completely off-access. The applet must get signed by a trusted certificate organisation that identifies the developer as a trusted entity. After it is signed, the developer can get access to a client's file system and network. This allows a developer to create regular TCP and UDP sockets to send and receive data.

2.5.2 Adobe Flash

Adobe Flash, formerly known as *Macromedia Flash* and *Shockwave Flash*, is probably the most commonly used plug-in on the web. It is used for many web applications, including animations, games and other rich internet applications. It is available for almost any operating system (including mobile ones) and web browser. One notable exception though, is *Apple*'s *iOS*, which actively blocks the Flash player.

Flash applications are created in Adobe's own *ActionScript* language. The Flash plug-in has networking capabilities, but only through a TCP socket, or a higher-level ActionScript construct. So for raw data transfer, one can use the TCP socket. A reference UDP socket is not available for the Flash player. This somewhat limits Flash's applicability to certain real-time applications. The higher-level ActionScript constructs can provide access to real-time streaming of video and audio using their proprietary protocol, but arbitrary data channels are not available.

Flash also does not allow to just connect to any domain. Whenever a connection is set up, it requires that the remote domain has a socket policy server running that notifies the Flash player that it will accept or reject the data.

2.5.3 Microsoft Silverlight

Microsoft Silverlight is a plug-in that is similar to Adobe Flash in terms of features. It aims to provide an easy platform for streaming media, but later also expanded to also include rich media content, animations and graphics. Microsoft officially supports Silverlight for the Windows and *Max OSX* operating systems, and several major web browsers. A free-software implementation of Silverlight, *Moonlight*, also made it available on the *Linux* and *FreeBSD* operating systems. Moonlight development, however, got dropped due to a lack of interest from developers. Support for mobile systems is also rather limited.

Developing for Silverlight is done using Microsoft's *.NET* framework, where several connection options are available, including TCP and UDP sockets, and also HTTP-based connections. But, just like with Flash and Java, several security restrictions are set in place. Silverlight too, requires a policy file to be retrieved from the domain to which data is being sent. UDP connections can be created from which data can be sent, but only multicast traffic can be received. Receiving unicast UDP traffic is not allowed.

2.5.4 Unity Webplayer

The *Unity Webplayer* from *Unity Technologies* is a plug-in designed specifically for games and highly interactive web applications. It is officially supported on the Windows and Mac OSX operating systems. Through the use of third-party support, it is also available on Linux. All major browsers are supported, except for mobile ones. The Unity development toolkit allows to export an application directly for mobile devices however, but the Unity Webplayer is limited to desktop systems only.

Developing a Unity Webplayer application is done in the Unity development toolkit, using one of the supported programming languages. A developer can use *Mono*, an open-source implementation of Microsoft's .NET framework. This means that, just as with Silverlight, regular sockets can be instantiated. In contrast to Silverlight, the Unity Webplayer does not restrict UDP sockets to only receive data from multicast sessions. In this regard, it is the plug-in with the least restrictions to networking, which makes it a good option for real-time data. It does, however, also require a socket policy file to be retrieved from the domain to which packets will be sent.

2.5.5 Custom Plug-ins

Many web browsers offer an external application interface that can be used to communicate with the browser from another application. This interface can be used to create a custom plug-in. Some frameworks exist that are tailored to created a custom plug-in, such as *Firebreath*. The *Qt* framework also has options to be used as a plug-in in the browser.

Creating a custom plug-in can be done when the developer believes the browser does not expose enough functionality. He can then extend it by creating his own. The challenge in developing a custom plug-in is found in supporting it for many the browsers and platforms available. Not every browser exposes the same interface. Nonetheless, some take on this challenge, and create a their own. The browser game Quake Live, was played using a custom developed plug-in. At the time of this writing, the maintenance of this plug-in is discontinued and converted to a stand-alone client application.

Chapter 3

Optimization Strategies

This chapter explores earlier and related research in both game and web development in bringing a better networking experience to clients. In both the fields of web and game development, several strategies exist to counter common problems. Here, a closer look is given to how latency can be reduced, how bandwidth is optimized, and analyse protocol overhead induced by commonly used protocols on the web and in game development. For each topic, similarities can be found in the strategies employed. These are discussed at the end of each topic.

Not every strategy can be clearly put into a specific topic of either latency reduction or bandwidth optimization. Some strategies will achieve both.

3.1 Reducing Latency

Every multiplayer game that is played, or every web page that is requested, has to deal with latency. This is due to the fact that every signal sent over the network is limited by several factors: the signal speed (speed of light), the load of intermediate nodes (routers, switches, proxy servers, ...), connection bandwidth, and processing latency.

Although the speed of a signal cannot be immediately be improved upon, other factors can be. Trivial solutions, however, such as increasing processing power and memory size of intermediate nodes, or increasing the bandwidth of a link are usually not within a developer's reach. Both the web and game development communities have their ways of improving on the issue of latency.

3.1.1 Latency Improvements on the Web

Improving the latency experienced by web users may have a serious impact. Some even talk about *the war on latency.* For commercial web services, especially web shops and search engines, latency can have a tremendous impact on the behaviour of customers. A higher latency can cause less purchases or less search queries, cascading to less revenue [32].

Caching

When a user requests a web resource, it is downloaded from a web server. On the way from the server to the user, the downloaded web page may be stored on intermediate nodes such as proxies or ISP routers. Whenever a request for the same page arrives next at such a node, it may already send the web resource to the user without contacting the original server first. This same system can also be applied in the user's web browser: the web browser saves a local copy of the page, and whenever the user requests the page again, it can be loaded directly from disk instead of the network. Disk access is faster, even on consumer-grade systems. The cache used by browsers is frequently used when the user presses the "back"-button to view a recently viewed web page.

Caching may result in serving the user an outdated web page. To overcome this, several mechanisms exist to invalidate the cached content, e.g. an update is posted by the user, or an expiration date. This forces a request to fetch the latest version of the page at the server again. Thus, caching can reduce latency when it is fetched from a node closer to the user.

This is interesting for web pages containing fixed, static content, and dynamic content fetched through asynchronous means. Here, the static content isn't changed often and can be cached for a very long time at the user or intermediate node. The dynamic content is then always fetched directly from the server. This reduces perceived latency by quickly showing the static content and later filling it up with the dynamic content. Total latency here is not necessarily reduced because the dynamic content may still take considerable time to load.

Content Distribution Networks

*Content Distribution Network*s (CDN) are a set of geographically distributed servers around the world, to provide web content to users in a fast and efficient way. A web developer may host its content on a server (his own or at a web hosting service), generally called the *origin* server. A user may then load the web content from this origin server but, depending on the user's physical location, these loading times may be fast or slow. To overcome this, the web developer may also may host this same content on a CDN. Users issuing a page request for a website that is registered with a CDN can be redirected to one of the CDN servers hosting the page. The server-selection process in redirecting the user usually takes into account the load of the CDN servers and the physical location of the user, determined through the user's IP address. The user's client will fetch the content from the designated server. In a study by Krishnamurthy et al. [33], resource retrieval was found to be significantly faster compared to retrieval from origin servers. They also concluded that the results improved when more servers were used by the CDNs.

CDNs can also counter *distributed denial-of-service* (DDOS) attacks. When one server goes down because of high server load, another server that is not overloaded, is selected to serve the content, resulting in a high availability of the content.

Pre-fetching

A requested web page usually contains some links to other pages, for example links to: the home page, a contact page, frequently asked questions, ... When the web developer can anticipate which links are likely to be clicked next, the web browser can be instructed to *pre-fetch* these pages. These pre-fetched web pages are cached by the browser and can be displayed immediately when the user clicks on one of the pre-fetched links.

Pre-fetching only reduces the perceived latency by the user because it is already present when the user needs it. But the actual retrieval of web resources still experiences the network latency. This technique's performance is thus heavily dependent on the prediction algorithm used by the web developer to correctly select the next links the user might click on. Predictions can be based on patterns found in the clicking behaviour of users. In general, predictive pre-fetching significantly reduces perceived latency [42] [18].

To reduce the latency to zero for the entire website, one could argue to pre-fetch every link referenced in a page. But this causes waste of bandwidth and other resources on the network when the user is not interested in the pre-fetched links. The pre-fetching algorithm should intelligently balance between reducing latency and bandwidth usage.

Next Generation Protocols

HTTP is the foundation of data communication on the web. However, the current version of HTTP (1.1) is an old protocol, and was not designed for the rich internet it is today. For every web resource that is to be downloaded, a separate request is send. Compared to version 1.0 of the HTTP protocol, version 1.1 allows the requests to be pipelined, resulting in lower latencies. Still, HTTP 1.1 suffers from *head-of-line* blocking because the requests are expected to be handled in the order they are received. Due to its wide usage however, HTTP cannot be just disposed of.

To overcome the limitations of HTTP 1.1, the next version of HTTP is being developed: 2.0. Currently, *SPDY* [27] is being considered as the starting point for HTTP 2.0. SPDY is a project started by Google to speed up the internet by reducing latency, hence its name. The latency reduction is partly achieved by eliminating the head-of-line blocking through removing the FIFO restrictions for the incoming requests. SPDY is not designed to replace HTTP 1.1. Rather, it complements it by manipulating its traffic, through modifying how HTTP traffic is send over the transmission medium, e.g. compression of the HTTP headers using gzip.

Google themselves tested SPDY and show a significant improvement over HTTP [27]. However, this gain is only achieved in certain network environments such as wired, high bandwidth or 802.11 WiFi networks. In other environments, e.g. 3G cellular, or web pages with resources scattered over many domains, SPDY performs on par with HTTP, or can actually hurt performance of loading web pages [17] [66] [15].

New Communication Techniques

To use dynamic content on web pages, a web developer had to use polling techniques, such as HTTP Polling, Long Polling and BOSH. These techniques all used regular HTTP connections to request the server for new or updated information. Each time a request was made, a new HTTP connection had to be opened, and it was closed when the server responded with or without new information. Each connection results in a relatively high bandwidth usage and high protocol overhead. Especially when only small amounts of information were needed. Another limitation was that the server could never initiate an update when it had new information available. BOSH provides a mechanism to circumvent this limitation, but still relies on the client to set up the initial connection. Polling techniques have a high protocol overhead and long connection set up time.

To create real-time applications on the web, a faster and persistent connection that allows data to flow freely between client and server is needed. The WebSocket JavaScript API and WebSocket protocol provide exactly that. A client must still initiate the connection, but once it is established it stays open for both to use and is closed by either the client or the server. Compared to polling and its many variants, WebSockets require very little overhead, and are much faster in transmitting data. Much research has been done in comparing WebSockets with polling techniques, concluding that WebSockets are faster and more efficient. WebSockets also reduce CPU and memory usage at the server, reducing processing delays. [51] [3] [34] [47]

Compression

Although compression is mainly used as bandwidth optimization technique, for some compressed web resources, the latency perceived by the user is improved although real latency is not necessarily reduced.

Perceived latency is the waiting time a user is conscious about. When this waiting time becomes too long, a user can become annoyed or frustrated and leave. This can be an issue when a web page contains large web resources that take considerable time to download. Common large web resources include images and video. The most primitive way of displaying them is to only show them when they are fully downloaded and available on the user's system. This, however, does not help in reducing the perceived latency the user experiences because he still has to wait the entire download time. In this case, the perceived latency is equal to the total latency. In order to reduce the perceived latency, one should provide a visual cue to the user to let him know that something is going on.

For images, this can be done in the form of *progressive image rendering*. When the first bytes of the requested image start to arrive, the web browser can already start to visualize this data. Simple progressive *as-received* image rendering will start to display the image data from top to bottom. Here, the user sees that an image is being loaded, but has little information how large the final image will be. *Interlacing* can tackle this last problem [41]. An interlaced encoded image allows to quickly display a poor quality result, but in full image size. As more data keeps flowing in, more details are added to the image until all image data is received, resulting in the original image. However, compression efficiency is decreased when encoding an image with interlacing support. This increases file size, and may result in a higher, total latency due to more data being sent over the network, but perceptive latency is greatly reduced! Figure 3.1 illustrates both visualization methods.

For video, a similar tactic is available through multi-layered approaches. One such implementation is called *Scalable Video Encoding* [56]. The video is encoded as a base layer and several enhancement layers. These layers are also commonly called streams. The base layer contains a low resolution / low detail / low frame-rate version of the original video. When the user requests the video, the base layer is sent first and can be played as soon as the first bytes have arrived. When the connection to the user allows it, more enhancement layers can be requested to upgrade the video to its original quality level in terms of resolution, frame rate and details. For video, it is also common to already display a frame of the video on the web

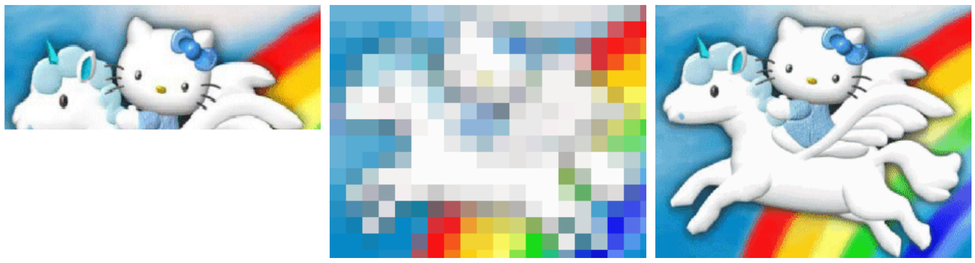


Figure 3.1: Progressive image rendering
Left: as-received. Center: interlaced. Right: final image.
Source: `blog.codinghorror.com`

page location where the video is played. This gives an indication of which video will be played.

3.1.2 Latency Improvements in Games

Latency can make or break a game. For slow-paced games, latency is not directly an issue. But for fast-paced, highly competitive multiplayer games, it can be a game-breaking one. High latencies can cause player frustration and can lead to less people playing the game. When people quit a game due to high latencies, a lower revenue is incurred when the game runs on a subscription based model or has a *micro-transaction store*.

Geographic Server Placement

When an online multiplayer game gets popular among players around the world, only one location for the game servers is not optimal for many players living far away from this location. The signal might take considerable time to get to the server, giving a disadvantage to these players and a possible cause for frustration when the signal has to go through several routers.

Many popular online games overcome this problem by placing multiple game servers across the world. These servers are usually placed in regions with a high concentration of players: the east- and west-coasts of North America, western Europe and China [10].

This geographic server placement also helps in spreading the load over multiple servers. However, because players are connected to different server regions, they often cannot play together beyond these regions. This is frequently observed in MMOs where players from Europe cannot play with players from North America. Some games allow the player to connect to a server region of choice, but usually also then show an indication of the latency to be expected when connecting to that server. This makes the player aware of the potential delay consequences when selecting such a server.

Sharding

A *shard* is an authoritative server, or cluster of servers, that hosts a copy of the virtual game world. This hosting of multiple copies of the world is called *sharding*, and is mostly applied in MMOs that offer a persistent world. In many games, shards are also known as *realms* or *worlds*. Players can connect to such a shard and then interact with other players connected to the same shard. Players on different shards, however, are seldom allowed to play together.

Sharding causes players to be spread across the several copies, resulting in servers that are load balanced, unless many players decide to connect to the same shard. Therefore, shards have a limit on the number of simultaneous players, forcing players to pick less busy servers. This guarantees that servers will have an upper bound on their load.

One notable exception of an MMO not using sharding is *EVE Online*. This game only has one instance of the game running on a server cluster. Players are spread across the virtual world, but some events may cause a heavy load on the server-side system, causing some locations to become unavailable when too many players are present.

Custom Protocols

Each game has its requirements regarding reliability and ordering of transmitting its game data. Some games simply use TCP for all of their communications. Other games implement custom protocols, usually built on top of UDP because it is the least restrictive and has low protocol overhead. Wu et al. [70] show an overview of popular MMOs and their primary communication protocols used. A custom protocol allows to tailor to a specific quality of service: when no reliability of data is needed, then it makes little sense in using an reliable communication channel which might slow down traffic when a packet gets lost and needs to be re-transmitted.

Open Game Protocol [60] is an example of a general game protocol running on top of UDP, providing real-time communication specifically designed for games.

Network Architectures

The simplest network architecture for multiplayer gaming is the client-server architecture. But the simple *single server* architecture will quickly run into scalability issues when increasing the amount of simultaneous players connected. The central server, or the network leading up to the central server, will then become a bottleneck in processing the updates, resulting in increased latencies. To overcome this, other network architectures and topologies can spread the load over multiple servers, or even players.

To keep the client-server architecture scalable when more players join, several of the techniques discussed above (and discussed further below) can be used together, e.g. sharding, geographic server placement and update filtering. Often, this is not enough and the architecture between multiple servers needs to be considered to optimally distribute the load. Clusters of servers can be used and put in a certain topology e.g. a star, full mesh or hierarchical.

ALVIC-NG is a client-server architecture designed to handle large amounts of players simultaneously by dynamically sub-dividing the virtual game world (see section 3.2.2), developed by Quax et al. [49] Each back-end game server takes up responsibility for managing part of the game world. When the server experiences high load, the game region is split up and divided among more servers. Regions can also be merged during calm periods. This is controlled by the region management system. However, clients do not directly connect to one of these game servers directly. Rather, they connect to a proxy server, hiding the underlying connection switching when travelling between regions or when they get split / merged. The proxies servers can also act as a sort of update cache, reducing latency by not querying the game servers for the latest information. Figure 3.2 shows an overview of the ALVIC-NG architecture.

However, complex server architectures can increase latency compared to the simple single-server architecture due to increased server management overhead and architecture abstraction through proxies.

A central server is not always necessary to run a multiplayer game, and peer-to-peer solutions can be viable. In a peer-to-peer setup, players send their game updates directly to each other through a potentially much shorter / faster path. Due to the absence of a central game server, management of the game state is now distributed over the participating peers, or one peer takes full responsibility over the entire game state. The latter case essentially boils down to a client-server setup again where one peer acts as both the server and a client. In the former case however, the distributed game state needs to be cooperatively managed and can create architectural and communication overhead. A peer-to-peer middleware implementation for *Quake II* is demonstrated by Reina et al. [52]

Most pure peer-to-peer multiplayer implementations are used in games where short and temporary tournaments are played, e.g. in shooters, strategy or racing games. This is because online games featuring a persistent

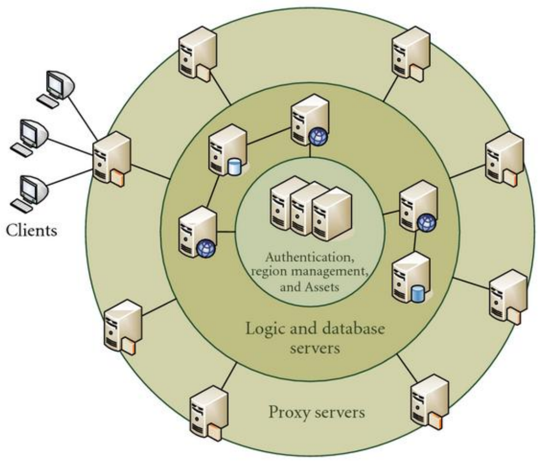

Clients
Authentication, region management, and Assets
Logic and database servers
Proxy servers

Figure 3.2: ALVIC-NG architecture.

world cannot rely on players being online all the time keeping the world available.

Hybrid models exist, client-server and peer-to-peer, that try to provide the best of both worlds. Some players are designated by the server as *super nodes* with which other players can communicate *non-critical* state changing updates. These players are trusted players by the game and have a good quality connection. *State changing* updates by players are however directly communicated with the central server. [28]

Player Response Cues

Some multiplayer online games can tolerate several hundreds of milliseconds of latency and can achieve this without affecting the user's gameplay experience. However, this latency can certainly become noticeable when simply making the player wait for its requested actions to happen. This perceived latency can become a frustration and make the game feel sluggish. Here, cues to the player's actions can help in eliminating this perceived latency by letting the player know its input is registered by the game. This is, for example, frequently observed in RTS or RPG games. In the case of an RTS game, the selected troops for example, will say some phrase when they are requested to move to another location or when an attack is initiated. For RPGs, the player's avatar abilities usually have some activation time with a corresponding build-up animation. Displaying such an animation, or confirmation phrase eliminates perceived latency, assures him that his input is registered, while at the same time creating some headroom for the update to be processed at the game server or other peers.

On the other hand, the responsiveness of a game and perceived latency are also subjective matters. Much research has been done to how latency influences the scores of a player in online game sessions, as well as how the players experience this impairment. People react differently on different values of latency in different types of games. Players also tend to underestimate their tolerance towards latency. [57] [43] [14] [50]

3.1.3 Discussion

Unsurprisingly, some similarities can be found in how the issue of latency is handled by the web and game development communities.

A first common divider is found in reducing the physical distance to the clients. Content distribution networks act as a large scale caching mechanism, placing copies of popular web resources close to the user, while on the other end, game servers are placed in densely populated geographic regions hosting the virtual game worlds. However, although servers are brought closer to the user, there is a big difference in how each maintains their data. CDNs act as mirrors for the original content residing on an origin server. Updates made to the content on the origin server eventually is also reflected on the CDNs. This is not necessarily the case for the game servers spread over the world. Although all game servers serve the same game, different players are connected to them, influencing the game state on that server, i.e. completing events or player economies. This especially noticeable in MMOs with a persistent world.

Another difference between CDNs and game servers is that web content hosted on a CDN is more volatile. A CDN server usually only serves popular web resources for that area. Whenever it starts to receive many requests for a certain resource, it may fetch it from the origin server and host it. This also means that unpopular content will eventually get replaced with more popular content. Game servers on the other hand, have dedicated resources to keep the game online for that geographic region, until the game is shut down.

A second method in closing the physical distance gap is done by using peer-to-peer technology. The web mainly operates through client-server connections, but some applications better lend themselves through direct communication between users, i.e. audio and video conversations. Privacy reasons also play a role in why such applications are best run over peer-to-peer instead of a client-server architecture. A user is never sure the server may record and store a copy of the conversation. It does, however, remain to be seen how quickly WebRTC is adopted by all major browser developers, as well as how well it can adapt to current and new trends in peer-to-peer communications. This can be problematic due to a web browser's adherence to web standards. Peer-to-peer for multiplayer games has been around for a long time. Several architectures have been developed to support large-scale worlds, but many face the issue of game security: player cheating [31] [7] [26]. This is a problem that arises due to managing the game on a machine that is not under the developer's control. Players may tamper with the system, giving them an unfair advantage over other players. This is part of what makes the client-server architecture still a very attractive choice for games.

Every networking application benefits from using an appropriate communications protocol that suites the application's needs. So an important factor to reduce latency is to optimize the protocols used for the application. For multiplayer games running natively on a machine, this isn't a problem. Several protocols exist that can be used as is and that have proven themselves in various scenarios. Or, a custom one can be built on top of an already existing one, which is frequently done with UDP. For applications running in a web browser, the options are limited and more restricted. Here too, web standards play a role, but also security issues. WebSockets are a big improvement over the older polling connections in terms of latency and bandwidth efficiency, but still lack flexibility to customize the underlying connection, e.g. enabling or disabling Nagle's algorithm. The WebRTC datachannel API allows limited customization for handling lost packets and packet ordering. However, using typed arrays in JavaScript (discussed in section 3.2.1), one could build semi-custom protocols on top of WebSockets or WebRTC datachannel by working on byte level.

For both web and game development, reducing the perceived latency by the user is also a very important factor. Giving the user a cue that something is going on helps in bringing a fluid experience. However, how this is handled differs between the two communities. On the web, the encoding format for video and images determines whether such cues can be given or not. Also, such a cue can only be given when a certain amount of information about the web resource has been received, e.g. the size of the image. In games, this happens the other way around, a cue is given when user input is registered, but the actual data still has to be sent. In some cases, reducing the perceived latency, the tolerance for the actual latency may increase. For images on the web, the gradual increase in image quality may distract the user that it is actually taking a longer time to load the image, than waiting for a blank canvas to instantly fill in a slightly less time. The same is true for certain games: giving the user feedback on his input, distracts him from the delay that it may take a bit more time to process his actions on all participating clients. For video however, there is a clear upper-bound when the latency becomes noticeable, namely when the buffer runs out of frames to play.

Table 3.1 gives a quick overview of how each community tackles the problem of latency.

	On the web	**In games**
Reducing physical distance	Caching and CDNs	Servers in populated regions and peer-to-peer
Protocol optimization	WebSockets and WebRTC, limited customization, standardization	Using proven protocols or build custom protocols
Perceived latency	Visual cues after first data is received, depends on encoding	Cue after user action, depends on type of game

Table 3.1: Overview of latency reduction strategies

3.2 Optimizing Bandwidth Utilization

Bandwidth in networking is defined as the amount of data per second a link between two systems can carry. However, although this link can be a limiting factor, intermediate devices can also form a bottleneck when too much data flows over the cables. When sending data over a network, it passes intermediate nodes that buffer and route the data to its destination. A high bandwidth utilization results in packets being queued at the intermediate nodes or maybe even dropped when the queue runs out of memory to buffer the packet. Queueing increases latency due to the node not being able to process packets fast enough. When a packet gets dropped, a re-transmit is triggered by the sender which increases latency, or the information is lost forever, depending on the transmission protocol.

Here too, intermediate nodes and links are not accessible to a developer to alter their specifications to meet the application requirements. Even if one had access, not every change would have the desired and / or expected effect, e.g. delays become more excessive when a router is equipped with an excessive amount of memory. This effect is called *bufferbloating* [16].

Optimizing the bandwidth utilization aims to reduce packet size or the amount of packets being sent, so that packets can be processed faster and reduce their chances of getting dropped. As a side-result, this also achieves some latency improvements by reducing queueing times and reducing the amount retransmissions.

3.2.1 Bandwidth Optimization on the Web

Even though a high bandwidth utilization does not necessarily mean that the user will have a bad web experience, it is a good idea to maximize efficiency of the used bandwidth of a link. Usually, costs are incurred on the amount of bytes that come and leave the web server. This becomes especially important when the web service is popular among many users. Optimizing the usage of a connection reduces the amount of useless data, such as protocol overhead or redundancy.

Compression

Compression reduces the amount of space a certain piece of data needs. When the server can compress the data requested by the client, and the client has the corresponding de-compression algorithm, less bandwidth can be used by sending the compressed web resource to the client. On the web, on the fly compression between client and server is mostly applied on text files such as HTTP pages, CSS and JavaScript using *gzip*. Especially HTML (and other XML-like variants) can be compressed very fast and efficiently due to redundancy in tags. SDPY, mentioned in section 3.1.1, applies gzip compression on the HTTP headers, greatly reducing protocol overhead.

Image and video files are rarely stored in a raw format on the server for display within the browser. They are usually already heavily compressed in one of the supported formats and are transmitted as such.

Another form of compression that can be applied on text files, and more specifically on source code, is *minification*. Minification of a source file removes all unnecessary white space characters and comments. More advanced minification can also parse the file and rename variables and functions to shorter versions or even a single character. Its important to note that, although the source code is altered, its functionality remains intact. It does have a negative effect on readability of the code for human reader. Minification on the web is usually applied on JavaScript scripts [23].

Compression can be executed as lossless or lossy. De-compression of a lossless compressed file will yield that exact same file again. Text files are usually compressed lossless to not alter the text after decompression. Images and videos can also be compressed lossless, e.g. *PNG* for images is lossless. However, when bandwidth is limited, or no exact reconstruction of data is needed, lossy compression can be used for images and video. Lossy compression loses a part of the original data which results in much lower file sizes than the original file or lossless compression. Lossy compressed images and video will have reduced quality and detail compared to the original. Supported lossy image formats by most web browsers are *JPEG* and *GIF*. Google seeks to introduce a new image format for the web: *WebP* [24]. They claim its compression is superior compared to the older JPEG compression with equal quality settings [25]. Mozilla on the other hand, aims to further improve JPEG compression. They developed a new encoder that is compatible with older decoders, and show that their implementation improves upon the old JPEG in every quality metric used in the study [12]. The new Mozilla encoder also outperformed the WebP format in some quality metrics. *WebM* [46] is a new lossy video codec supported by recent web browsers.

To further optimize bandwidth for users with different connection speeds, interlaced images and multi-layer videos can be sent. For a user with a slow connection, only the base-layer is sent which allows to visualize a lower resolution / frame-rate / detail result. When a connection is fast, more layers can be sent to enhance the base layer, improving resolution, frame-rate and details of the image or video.

Aggregation of Resources

A web page may contain many resources, i.e. JavaScript script, CSS files, icons, images, ... For each such resource referenced by the web page, the browser must download each file separately. Each file triggers a request to the server, setting up a new HTTP connection for relatively small portions of data. This causes a high protocol overhead.

To speed up the loading times and to optimize bandwidth utilization, web resources can be aggregated into larger resources. Individual JavaScript scripts can be put together in one large JavaScript file. Icons used by the web page may be put together in one larger set of icons, which the web page then "cuts out" when needed.

Aggregation of web resources optimizes bandwidth utilization and latency by removing protocol overhead for the otherwise many small file requests.

Typed Arrays

The primary data type for sending data through WebSockets, WebRTC and XHR, is to use strings. This is easy, but not transparent when transmitting numeric data. Indeed, the length of the payload depends on the value of the number. For example, when the number `1000` is converted to a string, it will contain four characters, while the number `999.9999` will contain eight characters when converted to a string. With every character encoded as `UTF-16` means the former will have a payload length of 8 bytes, while the latter has a payload length of 16 bytes. Both numbers are, however internally represented as 64 bit floats.

Typed arrays are a data view on a raw buffer of bytes, allowing to assign and interpret parts of the buffer as a certain numeric type, e.g. unsigned integers, 32 bit floats. WebSockets, WebRTC data channels and XHR can be configured to send these type of buffers directly. This way, numeric data is sent transparently with the desired amount of precision. Knowledge of the numbers being transmitted can result in a significant reduction in bandwidth utilization [35] [2].

Caching

Caching, discussed above for reducing latency, also causes a reduction in bandwidth usage. When the user requests a page, and it is already in its own browser cache (and the page hasn't expired yet), it can be just loaded from the user's disk resulting in no bandwidth usage at all! If the user does have to fetch the web page from an external source, it might have been cached at some intermediate node. This then saves bandwidth further down the line to the original content.

Using the example of the web page with static and dynamic content again, bandwidth utilization can be optimized by only requesting the dynamic content, because the static content is rarely updated.

Advanced Protocols and APIs

A web page can have dynamic content which is updated through incoming updates from the server. To receive these updates, the browser must communicate with the server. However, the browser is limited in communication options between client and server. Polling techniques, as described earlier, require a new connection to be set up over HTTP for each update. When these updates are frequent and / or small in size, the HTTP protocol overhead contributes largely to the bandwidth usage. Bandwidth is also wasted when no new information is available, because the client always is the initiator of the update process.

To eliminate this overhead and optimize bandwidth usage, other protocols and APIs can be used instead of polling, e.g. HTML5 WebSockets and WebRTC. Both WebRTC and WebSockets allow to set up a persistent, bidirectional connection. The connection only needs to be set up once for the current session, after which data can flow freely between clients and / or server.

For each message sent over a WebSocket connection, hundreds of bytes of overhead are saved compared to a polling technique. A WebSocket header only adds, at most, 13 bytes of header data. Rakhunde [51] compared WebSockets and HTTP requests in theoretical operation, concluding that WebSockets are far more efficient in every aspect.

A WebSocket connection still uses *ping-and-pong frame*s to keep the connection alive when no data is sent between client and server [20]. For WebRTC, keep alive packets are also sent as to prevent NAT at intermediate routers to close the connection [54]. These keep-alive mechanisms send a small packet in certain time intervals (typically a few seconds). When data is transmitted frequently over a connection, no keep-alive message is needed, because each end knows it recently received some data from the other end. But when the data flow is sparse or non-existent, the keep-alive messages waste bandwidth, although very little.

3.2.2 Bandwidth Optimization in Games

Reducing bandwidth utilization in games has the same remark as for web development: inefficient use of bandwidth can cause high waste of network resources, especially when the game has many players. However, in contrast to web resources, game updates are usually much smaller in size. This causes However, game updates are usually much smaller size compared to the data transmitted on the web. So many of the techniques discussed here focus on reducing the protocol overhead, because is it quickly is the dominating factor for the size of update messages.

Compression

Even if protocol overhead is the main contributor to the used bandwidth, its still interesting to see if the payload can be made smaller. To reduce the size of the payload each data packet carries, the game developer can shrink the size of the payload by compressing its contents. As with compression for web development, lossy or lossless compression can be performed. With lossy compression, part of the data is lost and cannot be reconstructed at the receiver side, e.g. loss of precision of coordinates due to rounding off values. With lossless compression, the other end can completely reconstruct the data as it originally was.

Besides lossy and lossless compression, the data can be compressed internally or externally. With external compression, the data to be compressed is considered in relation to the data previously sent. For example the *Protocol Independent Compression Algorithm* by Van Hook et al. [65] periodically sends a reference state, on which subsequent updates can be based on. The following updates can be based on this reference state, by taking the difference between the current game state and the last reference state that was sent. A state containing these difference values, called a *delta state*, usually has much smaller values in magnitude than the values in a full game state. These lower values could be represented by less bytes, and thus saving bandwidth. An update packet containing a reference state carries an important payload. Such an update should be sent reliably, or the updates derived from that state are worthless.

Internal compression, on the other hand, considers only the current data, and tries to compress the data based on redundancy found within the current data. For example,*Run Length Encoding* is a compression scheme that efficiently compresses long sequences of the same values.

Input versus Game Data

In most multiplayer games, the player only manages one central entity such as his avatar or a racing car. This central entity is usually also the only source that triggers the need to send updates. Updates then usually contain information about this entity's position, actions, ... However, in real-time strategy games, the player has control over a larger set of characters or troops. One could send out updates for each individual unit, but this will quickly result in a large amount of updates when the player has a large army and decides to take action with it.

When controlling a large set of entities, instead of sending updates for each entity, the user's input can be used as the subject for updates. The user's input is encoded and sent over the network where his input is remotely applied. Such an input update then, will most likely contain information of the following form: where has the user clicked, which entities are involved and what action should they perform. Multiplayer games using input as update mechanism must take care however that each input action is played back at each client in exactly the same way, and that games stay in sync. Slight deviations in applying the input may result in great game inconsistencies later on in the game. This is because input deviations are harder to correct later in the game than, for example, a player's character position which can be nudged back. Otherwise, one could better have sent the information about the entities in the first place! A commonly used, high-level protocol to achieve this, is the *lock-step* protocol [61].

Update Aggregation

Game updates can be grouped together and be send as one, larger update. Aggregated updates optimize bandwidth utilization by saving on protocol overhead, compared to sending each update on its own.

The game can enforce that a minimum amount of update data is present before it is sent. This guarantees that no bandwidth is wasted on protocol overhead for small updates. Such an absolute enforcement however, increases latency, and may potentially hold up an update forever, if no new update data becomes available. This minimum payload aggregation is useful for games that generate frequent updates, but can tolerate some latency, i.e. a few hundred milliseconds.

To overcome large latency values when updates become less frequent, a maximum delay timer can be put in place. When the timer runs out, the gathered updates are sent, regardless whether enough data was present to efficiently aggregate or not.

The potentially large delay time is more an issue at client-side, than server-side. At client-side, usually only one central entity (the user's character) is managed that triggers the need to send out an update. A game server, however, is a central meeting- and distribution point for game data. Therefore, the server can aggregate data much faster than clients.

Prediction Algorithms

Another way to reduce the amount of packets sent, is to predict and simulate what other clients are doing based on previously received game updates. Whenever a client receives an update about another client, it predicts its actions based on the information in the update.

Predicting the actions of another client does not necessarily mean these actions are also actually happening at the other client. This implies that inconsistencies can creep into the managed game state on clients. Clients will therefore also run a simulation of themselves to keep track of the deviation between the real client and the remote predictions of this client. Whenever a large deviation is (about to be) detected, a new update is send out to nudge the remote predictions closer to the real state.

A well known navigational prediction algorithm is *Dead Reckoning* or *Ded Reckoning*. Here, clients predict the position, movement and orientation of remote clients through the incoming updates containing information about the current position, velocity, acceleration, rotation... When receiving an update, a client will first converge its prediction to the state described in the update, followed by the prediction process.

Another prediction algorithm used in online games, is *Lag Compensation* [6] and refers to compensating latency of player actions through prediction. The client and game server will both simulate the player actions. Whenever, a player sends an update, the server will validate that update by looking back at previously received updates, checking whether the actions are possible. Meanwhile, the client predicts its actions by assuming the server accepts the update, until it receives the acknowledgement from the server. The Lag Compensation algorithm thus uses prediction while waiting for confirmation, keeping the feeling of a fluent game experience.

Using such an algorithm allows to send updates less frequently, saving bandwidth. But the game must be able to tolerate and compensate for (small) inconsistencies in game state. This compensation will come at the cost of more CPU cycles, depending on the complexity of the convergence and prediction algorithms.

Update Filtering

A player in the virtual world does not always need to have detailed information about actions or events happening in the distance, or even on another map. The update information sent to a client from another host or game server can be filtered based on the player's location in the game world. Another term for this filtering of updates is *interest management.*

One way to filter updates for a client based on its location can be achieved by spatially subdividing the game world in discrete zones [37] [38] [48]. This is called *zoning.* A player in a zone will only receive detailed and / or frequent updates from players or other entities in that zone. Spatial subdivision of the world can be applied explicitly by splitting the game world in maps or levels by, e.g. a loading screen as a boundary between zones. When the game consists of one large environment, or the individual levels are large, spatial subdivision can be applied implicitly through an overlay network of zones such as a rectangular grid, hexagonal grid, *Vonoroi* diagram, hierarchical tree structure, free-form, ... With implicit zones, a client may start to receive more detailed / frequent update information when standing near a boundary between zones, as to not cause any jerkiness in the audio-visual representation when crossing borders. The size of a zone does not need to be fixed. When many entities reside within a zone, its size could shrink to offload entities to nearby zones.

Another way of filtering updates is by considering the players and other entities as a starting point for filtering instead of the game world. The *aura-focus-nimbus* model developed by Benford and Fahlén [5], defines the levels of awareness and interaction with others in the surrounding environment, also called the *area of interest* (AoI). Players and other entities such as interactive objects, are coupled with each their own aura, focus and nimbus. Whenever two auras collide, possible interactions can take place. The focus and nimbus further determine the direction of data flow of the interaction. The focus determines that what the player is aware of, while the nimbus determines what is aware of the player. This means that whenever the player has something in his focus, he is interested in receiving updates from that entity. And whenever something is within the nimbus of the player, that entity is interested in receiving updates from the player. More than one aura-focus-nimbus can be associated with each entity, for example, one set for audio and one for vision. The aura-focus-nimbus model can be made arbitrarily complex by providing more levels of detailed update channels, or complex forms of representing the the aura, focus and nimbus, e.g. a cone for vision and a sphere for audio. The model can also be designed to let the player configure how much information he wishes to receive by, for instance, configuring certain settings in the game's options menu.

Filtering of updates can be done through a publisher-subscriber system. Whenever a player enters a new zone, or auras collide, the player's client can subscribe to certain update channels (publishers). Several subscription mechanisms exist, such as multicast groups on networking level, or subscription management servers on application level. In both cases however, extra bandwidth is used, or computational complexity is increased, to notify the system about (un-)subscribing to certain update channels. On the other hand, bandwidth is saved by not sending updates to the players that aren't subscribed to channels that don't interest them. Generally, bandwidth saved by not sending game updates is far outweighed by the subscribing messages.

Note that, zoning and the AoI models can be used together, i.e. zoning as a coarse-grained filter and the AoI as a more fine-grained update mechanism. For example, Marx showed in his implementation that using AoI can help in managing dynamic bandwidth scalability [39]. In a study by Boulanger et al. [9], various interest management techniques are tested, including region-based and AoI techniques. Their results show that, indeed, the AoI model provides a very fine-grained update filtering for players greatly reducing sent updates, but at the cost of a very high server CPU utilization. Region-based filtering techniques are shown to be more coarse-grained (especially square regions) letting more updates pass, but requiring less CPU time.

3.2.3 Discussion

The parallels between web and game development for bandwidth optimization are more clear than is the case for latency reduction.

Both on the web and in games, the amount of bytes a certain piece of information requires can greatly be reduced through compression. Compression on the web is either performed on the fly when the resource is requested (and possibly cached afterwards), e.g. on HTML files. Or, it has already been done offline and stored on disk so that it can be send immediately, as is done with images and video. Typed arrays can also be considered as form of compressing data that would otherwise be send as a plain string or blob. Compression in games is a continuous process that cannot be performed offline. Updates generated by players are unpredictable and thus cannot be compressed offline, unless some pre-defined or recorded action sequences are used. Sending input data instead of actual game data can also be considered compression, since the same result can be reproduced at the remote ends without sending an update for each individual entity.

With aggregation, bandwidth optimization is achieved by reducing the number of packets sent over the network. And here, it has a somewhat similar story to compression. Web resources are usually aggregated offline, e.g. icons placed in one large image or JavaScript scripts that are concatenated into one larger script. While for games, aggregation happens at run-time. Although aggregation reduces the amount of packets sent, thereby eliminating the protocol overhead for each packet, it has a different effect on latency on the web than it has in games. Aggregating web resources into larger files causes less requests to be sent, speeding up the loading process because many HTTP connections are eliminated. However, aggregation has the tendency to increase the latency because it must wait for enough data to efficiently aggregate. This latency-increasing effect is especially present on the client-side where update events are less frequent than at server-side.

Optimizing the communications protocol can also save bandwidth in both cases, but this is the same story as discussed earlier for protocol optimization for latency. Web browsers must adhere to standards and cannot just offer any protocol for the web developer to choose from, while a game developer can. But SDPY and WebSockets a offer great improvement compared to plain HTTP. For game developers, one must be careful in picking or developing a protocol who's headers are larger than the actual data sent, because typically, a game update will not contain more than a hundred bytes.

A final, but less obvious, similarity is found between caching on the web, and delta states in games. With caching, a copy of the static content is saved on the user's system. Whenever the page is loaded from the cache, dynamic content can still be loaded from the server. Together they form the final, and up-to-date web page. Reference and delta states, discussed in the section about compression, perform a comparable function. The reference state can be considered the static content that doesn't change much (relatively) and is cached at the client. Delta states then can be considered the dynamic content, that are updated frequently and build further upon the reference state to form the final game state.

A quick overview of the discussed similarities is shown in Table 3.2.

	On the web	**In games**
Reducing payload data	At run-time and offline compression, typed arrays	At run-time compression, input data versus actual game data
Reducing packets	Aggregate web resources, reduces latency	Aggregate game updates, can increase latency at client-side
Protocol optimization	SPDY and WebSockets are more compact than plain HTTP	Protocol headers can quickly become large compared to the actual update
Incremental	Static and long-lasting content with frequently updated dynamic content	A reference state with subsequent delta states

Table 3.2: Overview of bandiwdth optimization strategies

3.3 Browser-based Multiplayer Games

With the latency reduction and bandwidth optimization techniques analysed and discussed above, it is interesting to see how these techniques can be combined to created a browser-based multiplayer game.

When a multiplayer browser game is published, it will most likely be distributed over the web like a regular web page that a user can visit. To distribute the game efficiently, CDNs can be used. This also relieves the origin server from many download connections when the games becomes popular. To speed up this download process and save bandwidth at the same time, support for SPDY (HTTP 2.0) can help, especially when many resources need to be downloaded, e.g. 3D meshes and textures.

Many of these resources can be served compressed. Textures can be accepted in common compression formats (.png and .jpeg). Textures could also be aggregated to reduce the amount of requests. 3D meshes can be compressed in a certain format, but this depends on the game engine used that is capable of decoding the file. For audio files, such as sound effects or background music, the file format depends on the browser used. Not every browser supports the same audio codecs [63].

While downloading the game with all its resources in one go is an option, but this will likely cause a long downloading time. To further speed this up, resources could be downloaded in a dynamic manner, based on a player's location in the world and what is or should visible to him. This is similar to the area-of-interest update filtering model.

Once the game (or a playable part of the game) has been downloaded, connections can be made to a game server to start the multiplayer game. WebSockets, WebRTC and certain plug-ins allow to connect to different domains than the domain from which the game was downloaded. Servers can thus be placed in several locations, and the user can connect to one near him to get low latencies. The servers can be used to host the game or to let players find each other to set up peer-to-peer connections.

Depending on the desired network architecture, one or more connection options are available. For client-server, the options are plenty. WebSockets and plug-ins are generally capable of transferring arbitrary data between a client and server. WebRTC, although designed as a peer-to-peer API, can also be used in a client-server architecture by letting the server act as just another peer. For peer-to-peer gameplay however, only one option is immediately available: WebRTC. Although plug-in APIs generally offer more freedom in building a custom connection, the browser or plug-in developer usually restricts how those connections can ultimately be used. For example, a Java applet needs to be certified and signed before it is able to listen on a socket. The Adobe Flash player supports peer-to-peer but only for audio and video, not for arbitrary data. The Unity Webplayer, as well as the Adobe Flash player, require a socket policy server running on the domain to which they wish to send data. This would mean that, for peer-to-peer to be possible from such a plug-in, each peer would need to be running such a policy server next to its browser. Both of them are also restricted from listening on incoming connections, which makes them ill-suited for a peer-to-peer setup. The Adobe Flash player also has the extra restriction that it can only use TCP sockets, whereas the Unity Webplayer can use both TCP and UDP sockets.

Now that connections are set up, game updates can start to flow. Many of the bandwidth reduction techniques that are used for games in general and not restricted to native platforms, can also be used in browser-based multiplayer games: reference and delta states, prediction algorithms, update aggregation, and update filtering. When a lot of numeric data is sent, then typed arrays can also reduce the amount of space required when working with large numbers. If the type of game allows it, then input data can be sent instead of game data.

Chapter 4

Network Setup and Test Cases

Knowing the requirements that different genres of multiplyer games demand from a connection, and how they attempt to optimize its usage, its time to devise a series of tests that will evaluate several browser connection options.

This chapter starts with presenting the different connections that will get tested, followed by how they will get evaluated and which properties will be measured. This chapter ends with the procedure that is followed while testing each connection using a set of parameters.

4.1 Exploring Browser-based Multiplayer Options

As is detailed in the discussions in previous chapters, to build a multiplayer game with a web browser as the target platform, a heavy burden rests upon the shoulders of the communication channels and their associated restrictions. Choosing a communication protocol has implications on the networking architecture which in turn influences performance and scalability. It is therefore of great importance to choose the best communication option for the occasion. This thesis will analyse the following communication options for developing a browser-based multiplayer game:

- WebSockets and secure WebSockets,
- WebRTC data channels, and
- Plug-in sockets

Here, the Unity Webplayer is chosen to represent the plug-in camp due to it being the least restrictive plug-in for networking. A short introduction of the Unity Webplayer plug-in has already been given in section 2.5.4. The plug-in provides the developer with reference TCP and UDP sockets that can connect to any server domain that has a socket policy server running. Other plug-ins only provide a reference TCP socket and limit the usage of the UDP socket. Another option would be to develop a custom plug-in using frameworks such *Qt* or *Firebreath*. But developing a custom plug-in requires a high amount of support and development for all browser - platform combinations, and is also something that is not applied frequently in the game industry.

Because WebRTC has mandatory encryption for its data channels, the Secure WebSocket option is taken into the equation to better compare these browser-native channels. WebSockets only provide a reliable and ordered communication channel, so not much can be configured here. WebRTC on the other hand, allows to configure the reliability and ordering of messages for each individual data channel. So to compare WebRTC with WebSockets and the Unity Webplayer's TCP and UDP channels, it is configured with a 'reliable' and 'unreliable' channel. The reliable one has the same characteristics as a standard TCP connection: reliable and in-order delivery. To compare it with Unity's UDP socket, the unreliable channel of WebRTC is configured to match those characteristics: unreliable and out-of-order delivery. WebRTC also allows to configure the data channels as a mix of the two, for example, in-order but unreliable, or reliable but out-of-order.

The older HTTP-based communication techniques are not further examined here. Many previous research has already been done in comparing them with WebSockets. This research concluded that they perform far from optimal in many aspects compared to WebSockets, which makes them not really relevant for this thesis.

For each of these communication channels, several factors will be analysed to determine which one of them fits the best for certain types of multiplayer games. These factors include:

- Operational network overhead: how much bandwidth does the communication channel require from the network to operate, including side-mechanisms such as acknowledgements and keep-alive packets?

- Operational machine overhead: how much CPU resources does such a channel require from the host machine?

- Protocol behaviour: how does the communication protocol react to packet loss, delay and jitter?

- Browser choice: does the choice in web browser have an effect on the results?

Combining the answers from the above questions ultimately leads to creating a profile for each communication channel. This channel profile can then be compared to the requirements of several online multiplayer games, and can be used as a guideline to select the best connection for the job.

4.1.1 Expectations

Based on the discussions above and background information for each of the communication channels in chapter 2, some rough expectations about their profiles can already be formulated and are shown below.

- WebSockets and WebRTC require some overhead on top of the underlying protocols, it is expected that the plug-in solution will require less bandwidth due to its access to basic socket implementations.
- WebRTC implements the DTLS security layer into its communication channels. It is expected that this will lead to a small extra processing overhead and some extra CPU time compared to WebSockets.
- Whenever a message is sent or received while using the plug-in solution, the communication between browser and plug-in can significantly increase measured latency.
- The plug-in solution runs in its own instance next to the browser. It is expected that communicating over the plug-in solution will require more CPU time than WebSockets and WebRTC.
- Web browsers must comply to communication standards. It is then expected that, in terms of bandwidth consumption, the choice of browser will have little effect for WebSockets and WebRTC.
- Because the plug-in solution is loosely coupled with the browser then, whichever browser is selected, it is expected that there will be no difference in bandwidth consumption for the plug-in.

4.2 Evaluating the Connections

To evaluate these different connection options, a simple client-server setup and a series of test cases are used to monitor their performance. These test cases will mostly consist of measuring the round-trip times of the packets sent out by different connections. In these test cases, the network traffic is also captured for further analysis. More in-depth details about the network setup and the implementation of the these test cases can be found in appendix A.

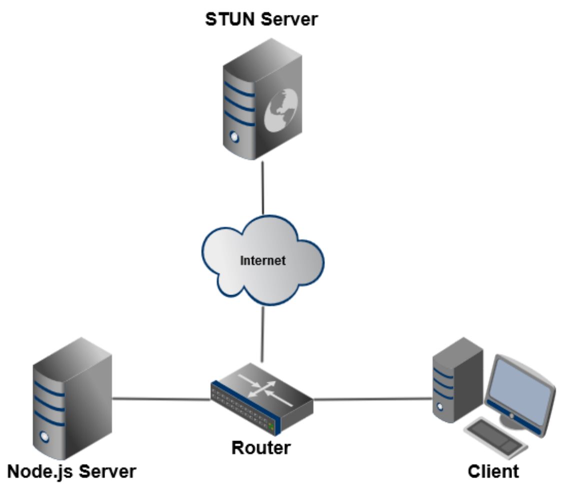


Figure 4.1: Network setup.

4.2.1 Network Setup

In the simple client-server architecture, only three devices are present: the client, the server, and a router that connects the client and server and is a gateway to the Internet. Figure 4.1 shows the simple network architecture. This setup is inspired by research performed by Laine and Säilä. [34] They analysed the possibilities for *XMPP* on the web over WebSockets and HTTP-based techniques.

The server runs an echo application that is able to send and receive packets from all of the different connections. This echo application uses the light-weight *Node.js* server framework to easily bring all the different connections under the same roof on the server-side.

On the client, two popular and readily available browsers are used: *Google Chrome* and *Mozilla Firefox*. Both of the browsers support WebSockets, can 'talk' to each other using a WebRTC peer connection, and allow the Unity Webplayer to be executed.

4.2.2 Test Cases

All connections are subjected to a series of tests that will analyse a specific characteristic. Three broad categories of characteristics are analysed: CPU performance, bandwidth usage, and reaction to varying network conditions. To measure the results of each characteristic of the connection, several parameters are played with between tests. The most important parameters are payload size and packet send rate.

One will notice that some values for these parameters will recur often in the different tests. For the payload size parameter, three values will be encountered: 100 bytes, 1,000 bytes and 10,000 bytes. The first value, 100 bytes, can be considered as the upper limit of the size that a typical game update will carry in a real game. This value will vary per game, or even per update, and will usually be somewhat lower, e.g. 65 or 80 bytes. The value of 100 bytes is chosen as a value that is easy to work with in combination with the larger values, while still low enough to characterize a stream of game updates. The larger values of 1,000 and 10,000 are purely academic values, used to magnify the effects exhibited by the connections, and to make the comparison more clear.

The commonly encountered packet send rate values are: 1, 10, 30 and 60 packets per second. The send rate of only 1 packet per second is not a send rate that will be commonly observed in a real game, but it allows to analyse certain effects of overhead very well. The other send rates more closely resemble real send rates encountered in games. Sending 10 packets per second could correspond to an intensive strategy game, or a role-playing game while in combat. 30 and 60 packets per second is sending a packet almost every frame, which corresponds to the update rate of fast-paced games such as shooters and racing games. All of these send rate values allow to discuss the connections for the different types of games.

CPU Usage

The test cases that focus on measuring the CPU resources used by a connection, will take place in the small client-server network in optimal network conditions. This allows the packets to flow rapidly between the systems, without much competing traffic. One sub-test in this test case will let each connection be exposed to the same network conditions, send packets at the same send rates and generate packets with the same payload size. This makes each connection process the same amount of data in the same amount of time. By keeping all these parameters the same, with the exception of the connection used, one can compare the round-trip times of the connections to get an idea of their load on the CPU.

The second sub-test of this test case lets the client and server play 'ping-pong' with each other. By measuring the time it takes for a connection to send a packet back and forth for a fixed amount of times, then one can directly relate this to the load a connection imposes on the CPU. Here too, all parameters are kept the same, with the exception of the connection used.

Bandwidth Usage

For measuring the bandwidth usage of a connection, packet traces are analysed for each of the connections under different send rates and payload sizes. Part of this analysis is also observing the overhead created by each of the connections. In a first analysis of the packet traces, the overhead directly attached to the actual payload is measured. This gives an impression of the direct overhead a fixed amount of payload has associated with it.

Next, the operation overhead is analysed by also including other packets that generated by the connection, but don't carry any actual game data. This operational overhead can be related to the game data sent, for example, an acknowledgement packet. Or, it can be unrelated traffic such as keep-alive packets to check if the other end is still online. Different payload sizes and send rates are used to determine their effect on the overhead created.

The last analysis in this type of test case is to measure the actual bandwidth used by a connection. This is done by accumulating the amount of bytes sent over a connection over a certain amount of time. Using the results of bandwidth usage to compare the connections can give the developer an intuitive idea of how this overhead translates in a load for the network.

Varying Network Conditions

The last test cases are based on sub-optimal network conditions. Artificial packet loss, delay and delay jitter are introduced to observe a connection's response to these factors, and to get a idea on how they perform in more realistic conditions. The round-trip times are measured here again. They can perfectly reflect how a connection reacts to loss and delay. The first sub-test will focus only on the effects of packets loss. Although the general behaviour for unreliable / out-of-order and reliable / in-order connections are well known, they are performed to detect any noticeable differences between similar connections, e.g. a reliably configured WebRTC connection that retransmits a lost packet faster than its TCP colleagues.

The second sub-test performed is one that combines all conditions at once: packet loss, delay and delay jitter. These results can show a developer the to-be-expected effect of a connection when things eventually go wrong. With the round-trip times and additional traffic traces on both the client and the server, one can follow-up exactly where a packet went, and how the connection decided to react to it.

4.3 Simulation Procedure

In the first and last test cases where round-trip times are measured, a strict procedure is used to setup and test each connection. If not every connection is tested in the same conditions, then some results may turn in favour of a certain connection and may lead to wrong conclusions. The entire simulation procedure is listed in pseudo code at the end of this section.

Before any tests are carried out, the different values for the parameters that will be used are decided upon. These include the earlier mentioned parameters of payload size and packet send rate, but also how long such a test should take, which browser is used and whether the network traffic should be monitored or not. In the test cases where more realistic network parameters are involved, they are configured too at this point. It is important to note that each test scenario only tests one connection at a time, and not several connections simultaneously. Otherwise, some connections my influence the behaviour or results of others.

With a given set of parameters, each test scenario is executed in sequence for each connection. First, the Node.js server application is started with the desired connection ready to accept any incoming connections. Then, the client is signalled that the server is running and it can start the desired web browser. The browser is directed to the web page that the Node.js server is offering and starts setting up the necessary connection. Now, the browser and Node.js start exchanging data packets and the browser will record timestamps for each packet it sends and receives. They will keep exchanging packets until a stop condition is fulfilled, e.g. a fixed amount of packets have been sent back-and-forth, or time has run out. At the end, the browser will send his recorded timestamps to Node.js, which in turn calculates the round-trip times and saves them to a file on disk.

After the results of this test scenario are saved to disk, the Node.js server application and the web browser are completely shut down again. So no previous state, that could have been created by earlier connections, lingers around on the systems. After both have been closed, the next instance of a test scenario is initiated, until all of the connections have gone through all sets of parameters. Each test case is also repeated multiple times to even out irregularities that may still occur in the network or one of the end devices, e.g. the system was performing some garbage collection, or the CPU was handling another system routine.

```
decide on parameter sets

for (number of iterations of this test case) {

  foreach (set of parameters) {

    foreach (connection in connections) {

      start Node.js with current parameters and connection
      start web browser and connect to Node.js

      start sending packets and record timestamps

      while (not done sending) {
        wait
      }

      send timestamps to Node.js
      calculate round-trip times and save to disk

      close web browser
      close Node.js
    }

  }

}
```

Listing 4.1: Simulation procedure pseudo code

Chapter 5

Results and Analysis

With the tests, described in chapter 4, conducted on each of the connection options, their results can now be analysed. This chapter will go over each test case separately and will analyse the results in depth. First, the results of the processing overhead on the CPU are analysed, followed by the protocol overhead on the network and bandwidth usage. The last test results analysed are those from the tests involving packet loss and varying network conditions. This chapter concludes with a summary of the results for each connection.

5.1 Processing Overhead and Delay

The first set of test cases are focussed on measuring the processing overhead for each of the connection types. Here, this is done by measuring the round-trip times for packets that are sent by the client and receiving its echo from the server. The round-trip results reflect, besides its travelling time over the network, how long it takes for a system to process an update. They allow to establish a relative ordering of the connections in terms of their delay due to system processing. Next to the round-trip times, the time the browser actively requires the CPU is monitored. A connection that is more processing-intensive will require more time from the CPU.

Two types of tests are performed here to gather the round-trip times. The first type of test will generate and send packets at a fixed rate, regardless of whether the echo of the previously sent packet has been received yet. The client will do this for a certain period of time. This can be compared to a game loop that sends out an update every n^{th} frame. The second type of test is a 'ping-pong'-like test where the client does wait for the echo to be received before sending the next packet. In this test, the time to ping-pong a fixed amount of messages is measured.

In essence, both these type of tests measure the same thing, but allow to quickly display the results in a different way, as will be shown below.

Each of the experiments performed in this section are conducted under optimal network conditions: high bandwidth, low latency, low jitter and low packet loss. No artificial delays and / or packet loss is introduced at one of the systems.

Each test case in the round-trip simulations is also repeated `10` times per connection to even out possible irregularities that may arise, e.g. the browser or Node.js that was busy doing garbage collection. Although only 10 repetitions are performed per test per connection, each test alone will generate thousands of samples that are spread over time. This should give enough statistical confidence about the data.

5.1.1 Round-trip Times

For the first type of test, the constant send-rate test, two different payload sizes are used to measure the round-trip times: `100` bytes and `1,000` bytes. The first payload size is an approximation of actual game data being sent. The second payload size is expected to enlarge the differences between connections: a larger packet may take more time / processing power to encode and decode. An even larger packet size (`10,000` bytes) was also considered, but communicating such a large payload between browser and the Unity Webplayer would make the plug-in unresponsive, and eventually crash.

The results shown are those for packet sizes of `1,000` bytes. At the end, a comparison of the effects of different packet sizes is shown. The packets are sent at a rate of *30* per second for *60* seconds.

Figure 5.1 displays box plots of the measured round-trip times for all connections in Chrome and Firefox. The figure shows that the observed round-trip times for Unity, both reliable and unreliable and in both browsers, are of a magnitude higher than those of the browser-native connections. The of values for the Unity Webplayer are also much more spread out over a larger range, while the values for the browser-native connections are more concentrated. This suggests that the WebSocket and WebRTC connections offer a much more stable round-trip time than the Unity connections.

Of course, because the Unity Webplayer is designed to run full games at a certain frame-rate instead of just passing through data, these results were to be expected. The Unity Webplayer runs at a certain frame-rate, typically 60Hz. Whenever the browser wishes to send a packet to the server over the Unity Webplayer, the packet is placed in a queue. There, it must wait for Unity's event loop in the main thread to reach the code where it accepts a message from the browser. Only then, the packet is processed and put on the network. This also applies when a packet from the network arrives at the Unity Webplayer. The packet can be read from the network by a separate thread and process it already. But to pass it through to the browser, it must again wait for the event loop in the main thread to read the message buffer and call an external function to the browser. This issue could be resolved if Unity allowed other threads that would run at a higher rate to perform these functions. Unfortunately, this is not allowed by Unity at the time of this writing.

For a packet to achieve a low round-trip time over the Unity Webplayer, it must be 'lucky' to be put in the queue just before the event loop checks the queue. Otherwise, the packet must wait at most one whole cycle of the event loop, which might be several dozens of milliseconds. A very 'unlucky' packet may be held up for a maximum of 33.333 milliseconds! This is assuming the event loop runs at 60Hz. Slower event loops can further increase this value.

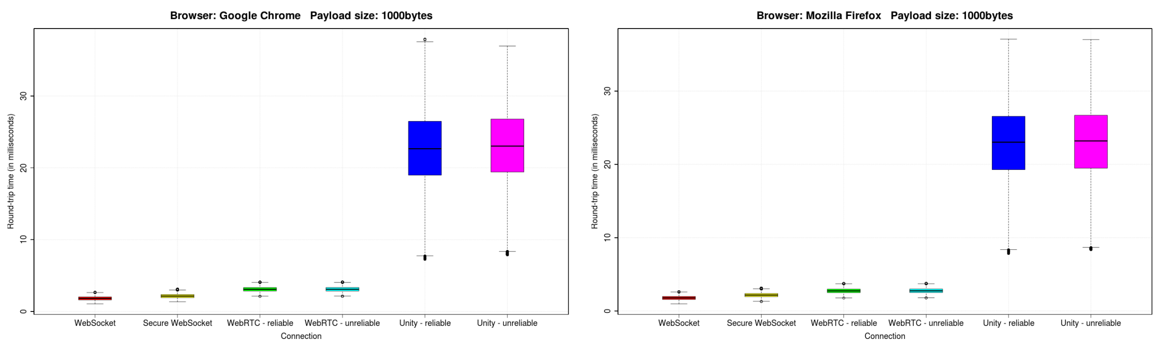


Figure 5.1: Round-trip times for all connections.
Left: Results for Chrome. Right: Results for Firefox.
X-axis: connection name. Y-axis: round-trip time in milliseconds.

Due to the large values of the Unity Webplayer, the results for the browser-native connections are difficult to analyse visually. Therefore, they are displayed again separately in figure 5.2. The figure shows a more clear image of the differences in round-trip times for WebSockets and WebRTC, as well as their stronger concentration on around the median. For both Firefox and Chrome, WebSockets achieve the better round-trip times. This can be due to several factors, including WebRTC being more taxing on the CPU due to the DTLS layer. Indeed, the server also decodes and re-encodes the packet when it is sent back. The larger packet size (results for packet sizes are shown in section 5.2) can also be a factor that may have an influence. However, in the results of packets with a payload size of only `100` bytes, WebRTC performs a little better, but still isn't brought down to the level of WebSockets with a payload of `1,000` bytes.

Another observation is that there is hardly any difference between reliability or ordering in both the WebRTC and Unity Webplayer channels.

Of course, because these tests are performed in optimal network conditions with (nearly) no packet loss, the channels that have reliable and ordered characteristics don't exhibit any delays yet.

A subtle difference is found between browsers when comparing the round-trip times for the WebRTC connections. Firefox performs slightly better in the handling of WebRTC traffic compared to Chrome. Figure 5.3 shows the difference for both the WebRTC connections. This difference could stem from an implementation difference of the WebRTC functionality, or because of the difference in browser architecture. Other connections don't show a significant difference in round-trip times between browsers.

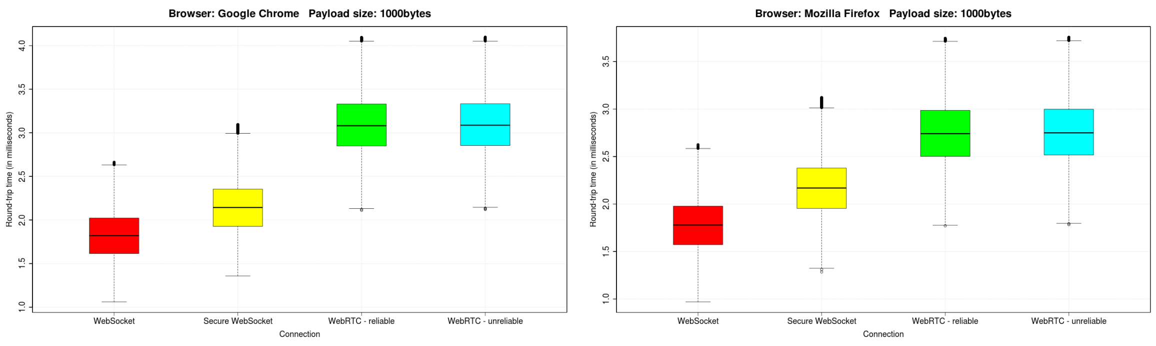


Figure 5.2: Round-trip times for browser-native connections.
Left: results for Chrome. Right: results for Firefox.
X-asix: connection name. Y-axis: round-trip time in milliseconds.

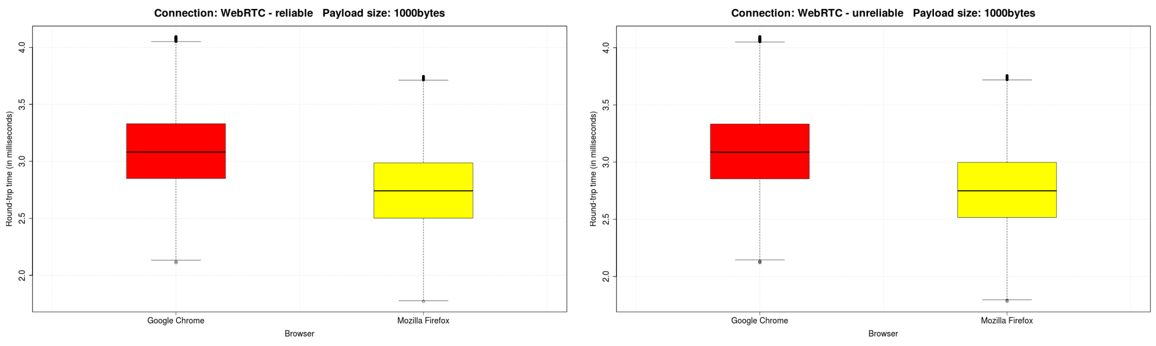


Figure 5.3: Comparison of WebRTC connections between browsers.
Left: WebRTC - reliable. Right: WebRTC - unreliable.
X-asix: connection name. Y-axis: round-trip time in milliseconds.

Because box plots don't show the round-trip measurements over time, figure 5.4 shows a limited time-lapse of the first 200 samples in Chrome. This figure also gives a good overview of the behaviour of the Unity Webplayer. The sawtooth-pattern strengthens the earlier statement about its slow event loop. The timestamps that the Unity Webplayer stores whenever a packet must pass through can also further illustrate this. Figure 5.5 displays the same 200 samples, but the measured round-trip times for the plug-in at JavaScript-side are replaced with the round-trip times measured gathered by the Webplayer itself. Comparing these two figures, it is clear that the Unity Webplayer puts a large delay on a packet when it is sent from the browser's JavaScript-side.

In this last figure, another observation is that the first packet sent over a Unity Webplayer connection is put on hold. The Unity Webplayer will first ask a socket policy file from the server. Only if the server responds with the correct parameters will the Unity Webplayer start sending packets over that connection. It does this for every connection it sets up. So the first packet sent will always experience a greater delay.

The raw results that were used to create the box plot figures can be viewed in tables B.3 and B.4 for Chrome and Firefox respectively. The results for payload sizes of 100 bytes can also be viewed there. They are found in tables B.1 and B.2.

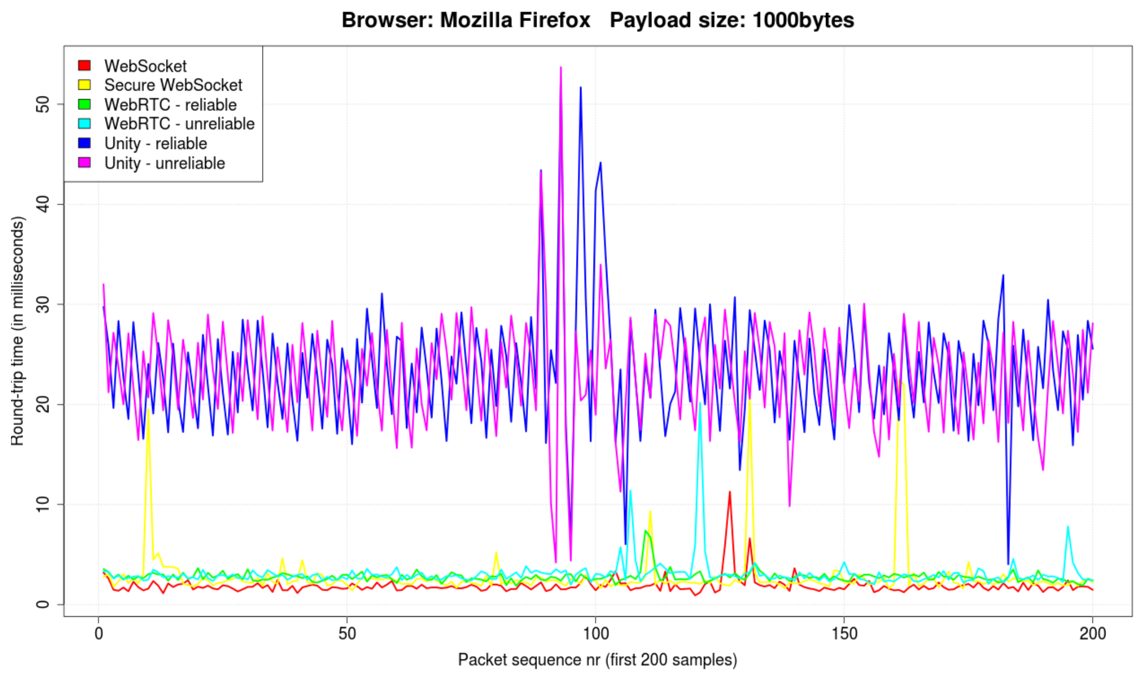


Figure 5.4: Round-trip time-lapse of the first 200 samples in Firefox.

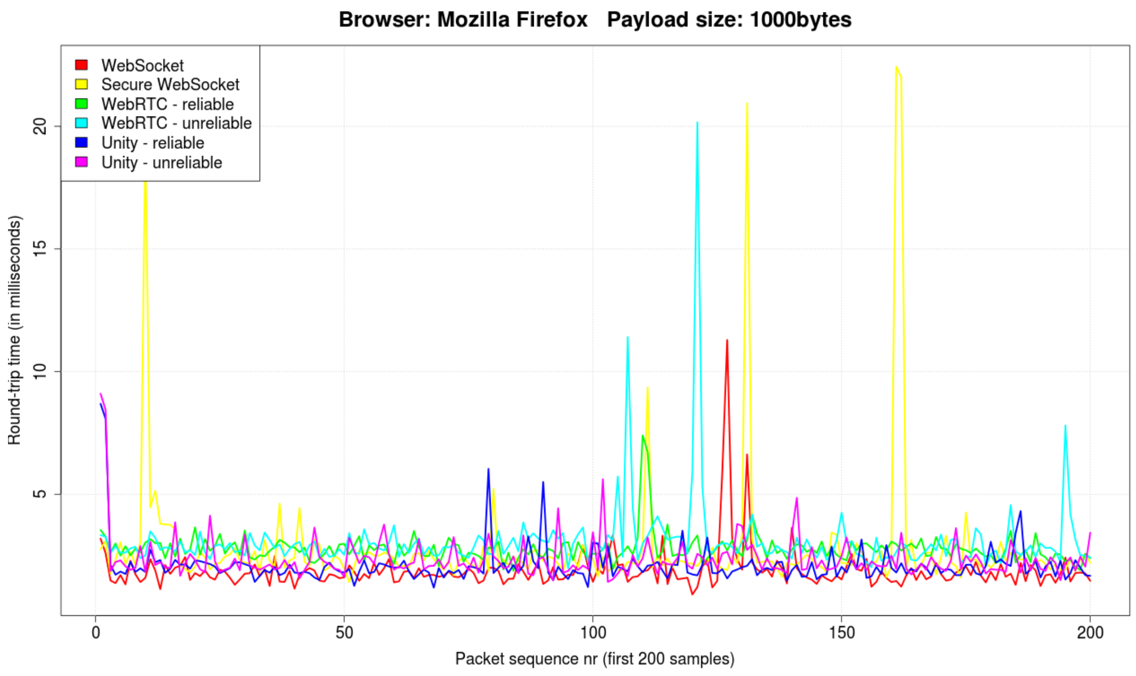


Figure 5.5: Round-trip times measured inside Unity versus browser-native.

As mentioned in the beginning of this section, these results can also be presented in a different form. Displaying the results as the amount of packets that get processed each second gives another intuitive feel for the performance of a certain connection.

In this test, the Unity Webplayer is left out of the results. The test is executed by only sending the next packet when the echo of the previous one has arrived. This puts the Unity Webplayer is a serious disadvantage, and performing this test with the Unity Webplayer would give results that could lead to wrong conclusions. For example, even though it can take up to dozens of milliseconds before the update finally reaches the browser again, does not mean that the CPU was busy processing the update. This would give the impression that the plug-in has a significant CPU overhead. Another mistake would be to say that Unity was only capable of handling one message per frame. On the contrary, it can accept and process multiple messages per frame, each being processed and passed through sequentially in separate function calls.

Temporarily removing the Unity Webplayer from the equation allows to use larger payload sizes than in the previous test. Besides the earlier payload sizes of `100` and `1,000` bytes, a payload size of `10,000` bytes is tested as well to measure the effect of a larger payload size. Figure 5.6 presents the result for a payload size of `1,000` bytes. A comparison of the values for all packet sizes can be found in tables B.5 and B.6 for Chrome and Firefox respectively.

The results, again, speak in favour of the WebSocket connections. WebSockets, even the secure version, can handle roughly `60%` more packets with a payload of `1,000` bytes. The gap becomes smaller for packets with a lower payload because the time a packet is travelling over the network starts to overshadow the actual processing time, but it is still very clear. The results also show a deviation between the two browsers. Firefox seems to be able to process WebRTC packets faster, consistently. For WebSockets, both browsers perform on par with each other.

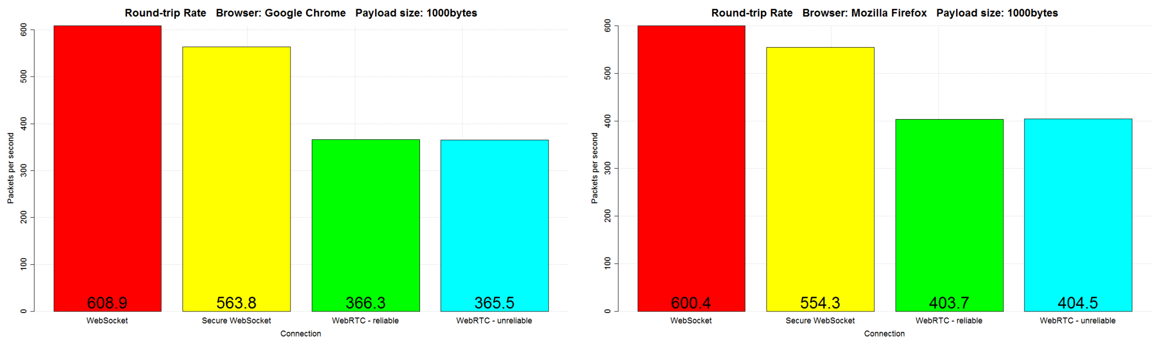


Figure 5.6: Packets processed per second.
Left: Round-trip rate of Chrome. Right: Round-trip rate of Firefox.
X-asix: connection name. Y-axis: packets per second.

5.1.2 CPU Time

During the first type of round-trip test cases, each connection has to process the same amount of packets in a fixed amount of time. This makes the ideal test case for comparing CPU resources consumed by the web browsers while using a certain connection. The CPU resources used by the browser are queried by the client-side management application at the end of a test instance. The management application can ask the operating system how long a process, or a group of processes, has actively used the CPU.

Figure 5.7 and tables B.7 and B.8 display the average CPU times consumed by a browser using a certain connection. Here too, the Unity Webplayer stands out by consuming a large portion of CPU time compared to the browser-native connections. Especially for Chrome, the plug-in consumes a large amount of CPU time. Unity's event loop will keep consuming resources at a constant rate (60Hz), even if no packets need processing. The large difference between Chrome and Firefox could be due to the architectural difference between the two browsers. In Chrome, the tab running the JavaScript code and the plug-in instance of the Unity Webplayer each run in a separate process. This means that, for each message sent and received, they must perform inter-process communication, which might attribute to part of the CPU overhead.

Comparing the browser-native connections again, then the same trend is visible here too: WebSockets consume slightly less CPU time than both WebRTC connections. Here, encoding and decoding of the messages due to the DTLS security layer in WebRTC can cause the increased CPU usage. Another reason might be code optimization. Since WebRTC is still an experimental technology at the time of this writing, its integration in the core of the browser may not be optimal yet. However, on Firefox, the difference between WebRTC and WebSockets is much less compared to those from Chrome. In general, it is hard to say, based on these results, that WebSockets perform significantly better than WebRTC on the CPU usage front, especially compared to the round-trip results above. The browser may be doing other tasks that overshadow the processing of encoding and decoding packets.

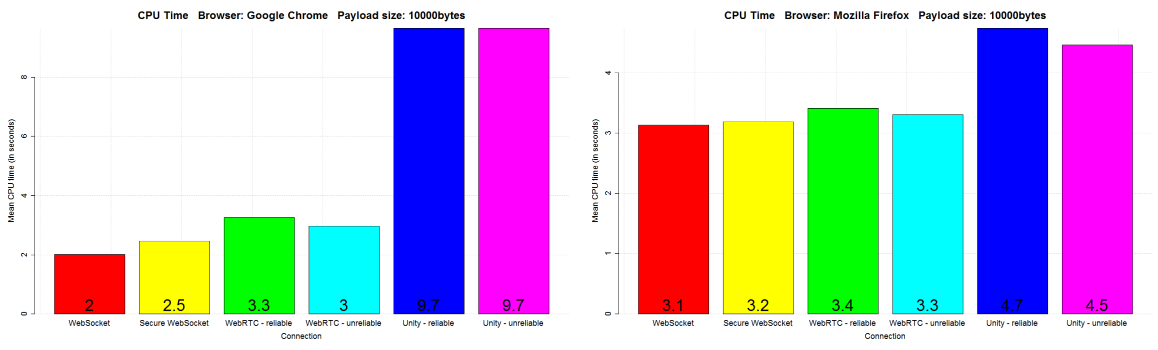


Figure 5.7: Comparison of average CPU resources consumed.
Left: Time consumed by Chrome. Right: Time consumed by Firefox.
X-asix: connection name. Y-axis: CPU time in seconds.

5.2 Protocol Overhead and Bandwidth Usage

This section will analyse the overhead that is introduced by the connections and their protocols when the packet is put on the network. This also includes any overhead created by other mechanisms that support the connection.

First, the overhead that is directly attached to an actual data packet is investigated. This is the overhead that is generated for each packet sent out. Next, the operational overhead is analysed by also including any other packets or data generated by the connection that is not attached to the application data. Lastly, bandwidth usage is measured.

Analysing the overhead created by the connections is done using traffic captures made with Wireshark. And here too, scenarios are run with different packet sizes to see whether payload size has an effect on the behaviour of the protocol. The two payload sizes tested are, again, `100` and `1,000` bytes. To analyse the operational overhead, different packet send-rates are also used to analyse whether this has any absorption effect on some packets, e.g. ACKs that get absorbed into subsequent packets.

Note that, all packets sent over the network also include the *Ethernet* and *IPv4* headers that are mandatory for all packets sent over a network that involves the Internet and many other networks. Their header sizes are also incorporated into the overhead analysis.

5.2.1 Data Packet Size

By knowing how much actual payload data is generated per packet in a test case allows to analyse the overhead introduced per packet by the connection. The network traffic captured by Wireshark allows to quickly view the total packet size of a packet containing game data.

Comparing the packet sizes and analysing the overhead introduced by the protocols in figure 5.8, one can see that the WebRTC connections produce the largest packet overhead of all connections. The overhead for both the reliable and unreliable channel is exactly the same. This is because the SCTP protocol uses the same packet structure for both reliable and unreliable transmission. Only when a packet is detected to be lost will there a difference in how data is handled depending on its configured reliability. More details about this procedure are found in section 2.1.3. Figure 5.9 shows the header overhead expressed as percentages of the total packet size.

For the tests with a payload size of `100` bytes, the overhead even exceeds the actual amount of data: `100` bytes of data versus `131` bytes of overhead. This high overhead does not come as a surprise. The protocol stack for WebRTC data channels is quite large: SCTP / DTLS / UDP. More information about this protocol stack can be found in section 2.4.2. Note that, these result, are only valid for WebRTC data channels. For audio and video communication over WebRTC, a different protocol stack is used, and other parameters are involved such as the audio and video codes that compress the stream.

WebSockets and the Unity Webplayer's reliable connection nearly produce the same results, but WebSockets create a small amount of extra overhead. This is due the thin WebSocket protocol layer on top of the underlying TCP connection. The WebSocket header size is also dependent on the size of the payload. When the payload size exceeds a certain amount (125 bytes), then the WebSocket protocol needs a few extra bytes to enlarge the 'payload length' header field. For a payload of 100 bytes, the overhead produced is 60 bytes, while for a payload of 1,000 bytes, the overhead becomes 62 bytes. For the TCP connection in the Unity Webplayer's reliable connection, the overhead stays put at 54 bytes.

The size of the WebSocket protocol header also depends on whether the data is coming from the client, or from the server. Data sent by the client is masked, which requires 4 extra bytes for the *masking key* inside the WebSocket header. Data coming from the server is not masked and does not need this value. The WebSocket protocol specification even dictates that no data sent by the server should be masked. [21] The WebSocket results shown in the figure 5.8 are masked results. More information about the masking key and the length header fields in the WebSocket protocol are found in section 2.3.1.

Although not displayed in figures 5.8 and 5.9, the Secure WebSocket connection performs differently between browsers. The value displayed is the one observed in the traces of Chrome: 189 bytes. However, the results for Firefox have a slightly higher packets size: 203 bytes. This is due a difference in how Chrome and Firefox encrypt their data over TLS. For Firefox, these packets also stay at a fixed size of 203 bytes. No difference is found between packets coming from client or server, even though the masking key is included in the packets coming from the client. For Chrome, the packets coming from the clients do differ in size than those coming from the server. The packets size of 189 bytes includes the masking key. Those coming from the server are 185 bytes. Decrypting and inspecting the payload using the self-signed certificate, reveals that both carry the same WebSocket information as with the non-secure WebSocket connection. The difference between Chrome and Firefox in the Secure WebSocket connection will be made visually clear in the next couple of tests.

The lowest overhead for a packet is observed for the Unity Webplayer's UDP connection, with an overhead of only 42 bytes, regardless of payload size. This low overhead is achieved due to UDP not having sequence numbers or flags that indicate an acknowledgement or fragmentation of the data, keeping it light-weight.

5.2.2 Total Overhead

The total overhead created by a communication channel can be found by analysing the Wireshark traces and searching for patterns over a period of time. The search is also broadened by performing the test with different

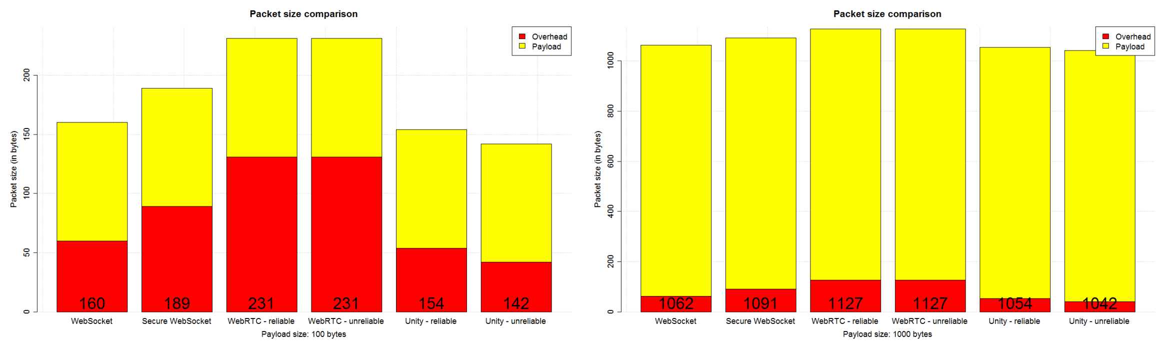


Figure 5.8: Comparison of total packet sizes.
Left: 100 bytes payload. Right: 1,000 bytes payload.
X-asix: connection name. Y-axis: packet size in bytes.

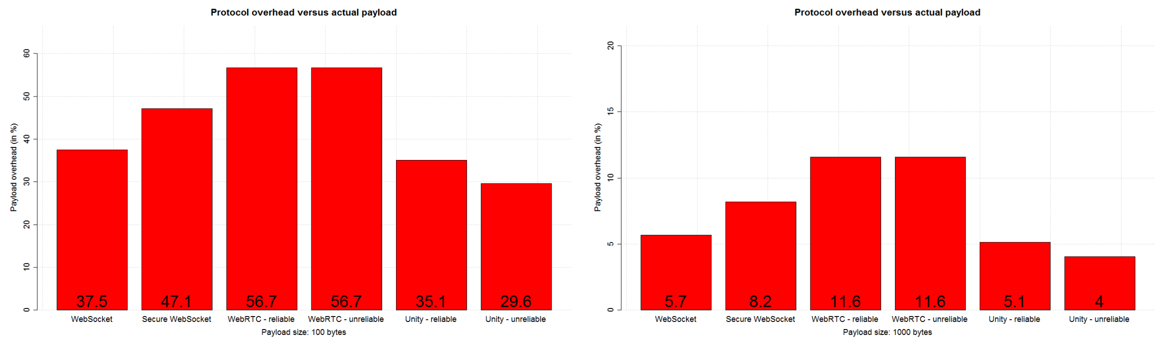


Figure 5.9: Comparison of header overhead percentage to total packet size.

Left: 100 bytes payload. Right: 1,000 bytes payload.
X-asix: connection name. Y-axis: header overhead percentage with respect to total packet size.

sending rates to see whether certain effects, such as packet absorption when sending packets at a higher rate, are observed.

Four send-rates are analysed: 1, 10, 30 and 60 packets per second. The first send rate is merely chosen as a baseline to see every aspect of the connection in action separately. The send-rate of 10 packets per second could be considered as the send-rate experienced during multiplayer strategy games. They are usually limited by the number of events a player can generate per second, and thus can run at a slower rate than a typical rendering rate. A send rate of 30 or 60 packets per second would correspond to sending a packet almost every frame. Such a high send rate is frequently applied in fast paced multiplayer games such as shooters where the timeliness and frequency of updates is important.

In figures 5.10 and 5.11, an overview of the results is shown. The values are also available in table B.10. Table B.9 also gives the raw numbers that are found analysing the packet traces. Each connection is analysed separately in more detail below.

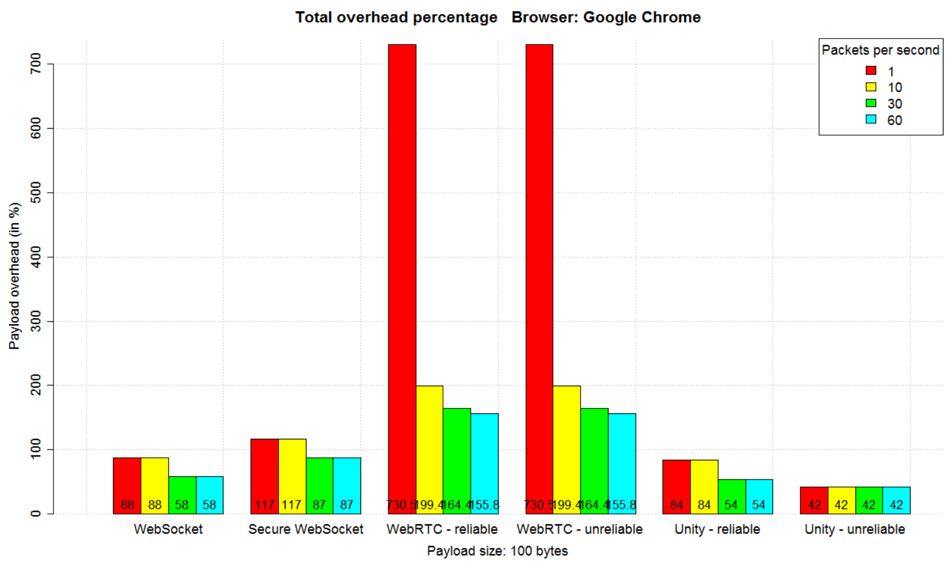


Figure 5.10: Total operational overhead percentage for a payload of 100 bytes in Chrome.

WebSockets

As was mentioned in the analysis above of the WebSocket packet sizes, its header size depends on whether the packet is sent from the client or from the server. An extra operational overhead for WebSockets is found in the acknowledgements sent when sending at a rather slow rate. An ACK packet is sent from the client to the server, ~50 milliseconds after receiving the echo packet back from the server. The ACK packet has a size of 60 bytes. It is, however, not seen any longer when sending at a higher rate: 30 and 60 packets per second absorb this ACK packet in the next packet sent out by the client. In this network environment, one would need to start sending packets at a rate a little higher than 20 packets per second to start this ACK absorption process.

Even though the ACK packet gets absorbed at higher send rates, does not increase the overhead of each packet carrying actual game data. This is because the acknowledgement fields for TCP are mandatory in its header. At these higher send rates, the traffic becomes a pure data stream without any other side-mechanisms further adding to the overhead.

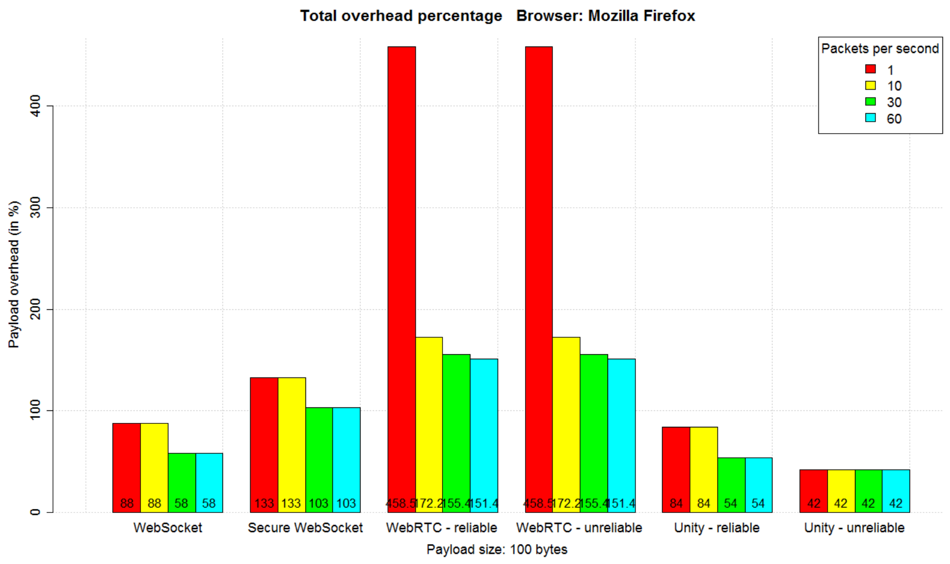


Figure 5.11: Total operational overhead percentage for a payload of 100 bytes in Firefox.

WebRTC

The overhead for a WebRTC packet is already found to be fairly large compared to small payloads. An additional overhead is found in its selective acknowledgement (SACK) mechanism. Whether the stream is configured to be reliable or unreliable, SACKs are always exchanged between end-points. A SACK piggybacking on a packet, in this case, adds a 16 byte overhead. This SACK can grow in size when more streams are multiplexed over the same SCTP connection. A standalone SACK in this test is measured to be 135 bytes. At slow send rates (1 packet per second) this standalone SACK is sent out by the client after ˜200 milliseconds it received its echo back from the server. So in this case, one needs to send a little faster than 5 packets per second to let the SACKs piggyback on the data packets.

The send rates of 10 packets per second and more don't exhibit this standalone SACK any more. They are then carried along with the other data.

One other, very noticeable, mechanism that is present in the WebRTC channels, is the STUN traffic. Once a WebRTC connection has been set up, it requires STUN to check liveliness and consent from to other peer to keep sending data. However, the observed behaviour is browser-dependent. This specification [45] describes that STUN packets should be sent every ˜5 seconds.

When Firefox is used, this traffic is almost absent. It is possible that

Mozilla followed the SIP specification that also uses STUN for the same purposes, and sends them every `25 - 29` seconds. Such an infrequent check might be a too slow rate to detect that the other client is not available any more. Chrome, on the other hand, sends out STUN requests very aggressively. Roughly every `~500` milliseconds. Unfortunately, the Node.js WebRTC module uses the Chrome implementation for WebRTC, which makes it behave identically in this regard.

This implementation difference between browsers for the STUN traffic has a large impact on the operational overhead of WebRTC. Every STUN request-response procedure costs around `260` bytes. Observing this for the traces in Chrome, `1048` bytes are sent every second: two requests from Chrome, and two from the server. In Firefox, the STUN traffic is dominated by the requests from the server, resulting in `504` bytes per second originating from the server, and approximately `10` bytes per second from STUN traffic originating from Firefox (assuming a STUN request is sent every `25` seconds). This difference will also be made visually clear in the analysis of the bandwidth usage.

This currently means that, even for Firefox, WebRTC has a very large operational overhead for small payloads and slow send rates, which is clearly visible from the figures. Increasing the payload and / or send rate spreads this overhead over the packets.

Unity Webplayer

The Unity Webplayer houses two sockets of which the characteristics are well known. The TCP connection of the Unity Webplayer has the same operational overhead that is described earlier for WebSockets. It has the separate ACK packet of `60` bytes that is observed when the send rate is slow, which is sent approximately ~50 milliseconds after the echo has been received. The ACK packet is absorbed when using the send rates of `30` or `60` packets per second.

The plain UDP stream is the most pure data stream one can observe in these packet traces. Each packet sent over the connection has the same size. The only overhead observed is that of the headers that are carried with the payload. This also means that, whichever send rate or packet size is chosen, the packet overhead stays the same for each set of parameters.

5.2.3 Bandwidth Usage

The same Wireshark traces that were used to analyse the total protocol overhead, are here also used for determining the bandwidth utilization of a connection under a certain send rate. The elements that are identified as part of the total overhead are applied in Wireshark filters to distil the bandwidth usage of a connection. The bandwidth usages for the same four send rates, `1, 10, 30` and `60`, are analysed. The bandwidth usage analysed here is for a payload of `100` bytes.

Figure 5.12 shows a comparison of the bandwidth consumed by all of the connections under different send rates, and for each browser. Tables B.11 and B.12 show all of the numeric data for the four send rates. In both graphs, one can see that WebRTC consumes the most bandwidth. This is a consequence of the large overhead that is created by STUN. However, as was mentioned in the analysis of the total protocol overhead, this STUN overhead, can be diminished by sending packets at a higher rate. This effect is observed by looking at the relative height of the WebRTC connections compared to other connections in the two graphs. Still, WebRTC will remain the connection that consumes the most bandwidth, at equal payload size, due to the many headers that are wrapped around the data.

A second observation is the large difference between Chrome and Firefox when they send data over a WebRTC connection at `1` packet per second. This is attributed to the implementation difference of STUN traffic generation between the browsers. The difference in height is purely the absence of Firefox' STUN traffic. If both browsers (and by extension the Node.js WebRTC module) would have implemented this according to the current specification, WebRTC's bandwidth usage would lie closer to the other connections.

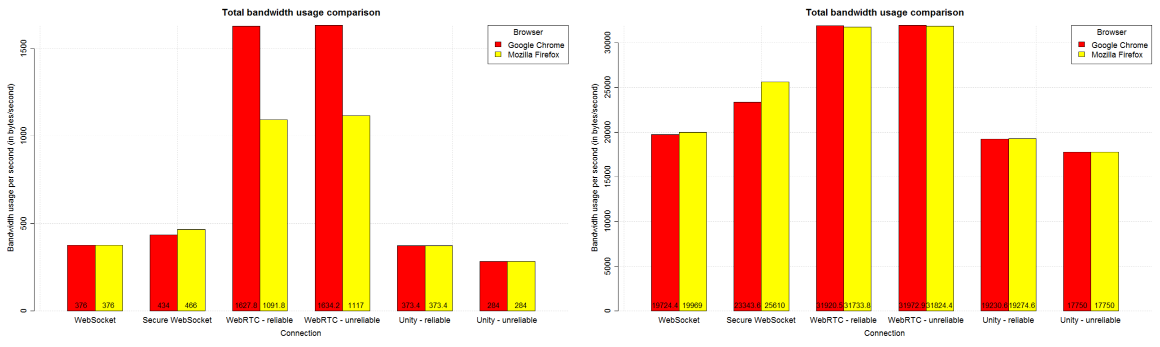


Figure 5.12: Comparison of bandwidth usage between browsers.
Left: 1 packet per second. Right: 60 packets per second.
X-asix: connection name. Y-axis: bandwidth usage in bytes per second.

5.3 Effects of Packet Loss

In all of the previous test cases, ideal network conditions were assumed. In this section, packet loss, delay and jitter are introduced to observe how each connection reacts to these two different situations. Running tests in ideal network conditions helps in analysing certain properties of a connection or underlying protocol, but more often than not, a real-time multiplayer game will be played in a network that will exhibit higher latencies, jitter and packet loss. How the connection will behave when it encounters such situations can have a high impact on the playability of a game.

Two types of test cases are performed. In the first type of test, only the effect of packet loss is monitored to gain insight about how a connection responds to loss and what consequences this has for subsequent packets. In the second type of test cases, a more realistic network environment is emulated by introducing delay, jitter and packet loss all at once. In both cases, the round-trip times are measured again. Just as with the earlier round-trip tests, these too are executed 10 times to gain more confidence about the results.

5.3.1 Packet Loss

In the tests for packet loss, the server is configured to randomly drop 5% of its incoming and outgoing traffic. Initial tests were performed with a fixed, n^{th} packet that would get dropped from the stream. However, this would lead to repeating patterns that were unwanted for analysing data. For example, every WebSocket stream would get a packet dropped, as well as its retransmission, resulting in very high peaks for the WebSocket connections.

Figure 5.13 gives a time-lapse overview of the first 200 round-trip samples captured for each connection at a send rate of 10 packets per second. Dealing with such a relatively high chance of getting a packet dropped, the send rates of 30 and 60 packets per second could not be dealt with by the Unity Webplayer plug-in when using an asynchronous implementation. This is caused by the large amount of threads that are created and are busy waiting for their packet to be put on the network.

The spikes in the figure are caused by the ordered nature of the TCP connections, and due to the ordered configuration of the WebRTC channel. This is the expected behaviour of these channels. Because they need to deliver data in-order, subsequent packets have to wait for their lost predecessor to be retransmitted and be delivered. Another observation is that the retransmission of the WebRTC channel occurs much faster than the TCP-based channels, even though both use a comparable retransmission trigger mechanism: either a timer runs out, or a few consecutive (S)ACKs notify the sender that data is lost.

The Unity Webplayer and WebRTC - unreliable connections don't show any spikes because they just forego retransmitting. The WebRTC unreliable connection does know about data being lost and will notify the sender about it. But the sender will, in its turn, tell the receiver to just forget about it by advancing its transmission sequence number. More information about WebRTC's unreliable transmission can be found in section 2.1.3.

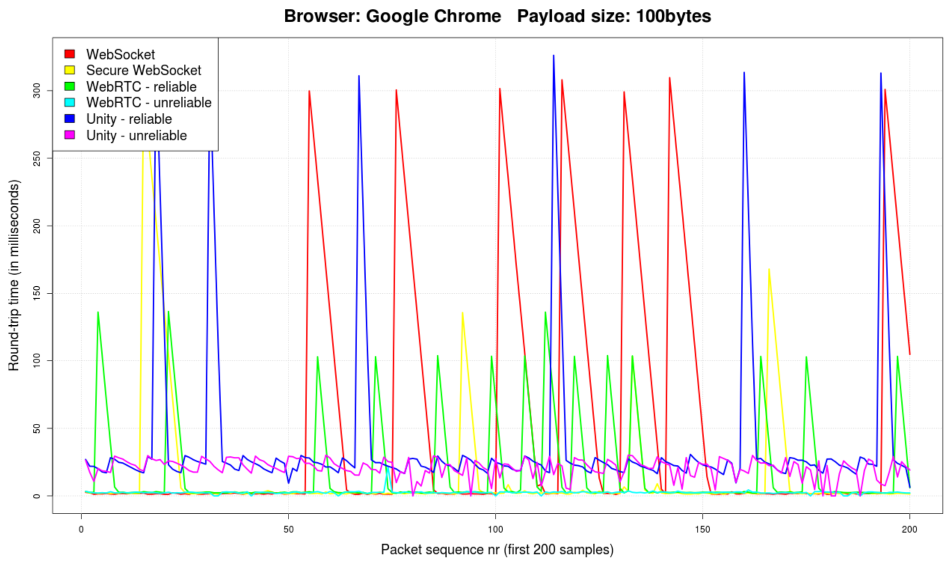


Figure 5.13: The effect of packet loss on each connection.

5.3.2 Wide Area Network Emulation

A rather busy network environment is emulated in this test case. The traffic shaper on the server-side is configured to emulate a network with a one-way delay of ˜50 milliseconds and 5 milliseconds of jitter that is applied from a normal distribution, with a 1% chance of getting dropped. Packets are sent at a rate of 10 per second.

A time-lapse of the first 200 packets is shown in figure 5.14. These results are fairly similar to the results shown with only packet loss, except for the extra jitter and an upward shift due to the added delay. This jitter is extra noticeable on Unity's TCP connection. This is due to the event-loop that introduces some jitter on its own, which is sometimes accumulated with the jitter introduced by the network. On other times, this event-loop jitter cancels out the network jitter. This accumulation is also present for its UDP channel, but it is less noticeable.

In this time-lapse figure, the UDP connection of the Unity Webplayer also seems to 'blend in' with the WebRTC - unreliable connection. This is also illustrated by the box plots that are shown in figure 5.15. The quantile values in tables B.13 and B.14 more or less confirm this observation. They show that Unity's UDP connection is still slightly higher. But the earlier impressions in the section about the round-trip results are smoothed.

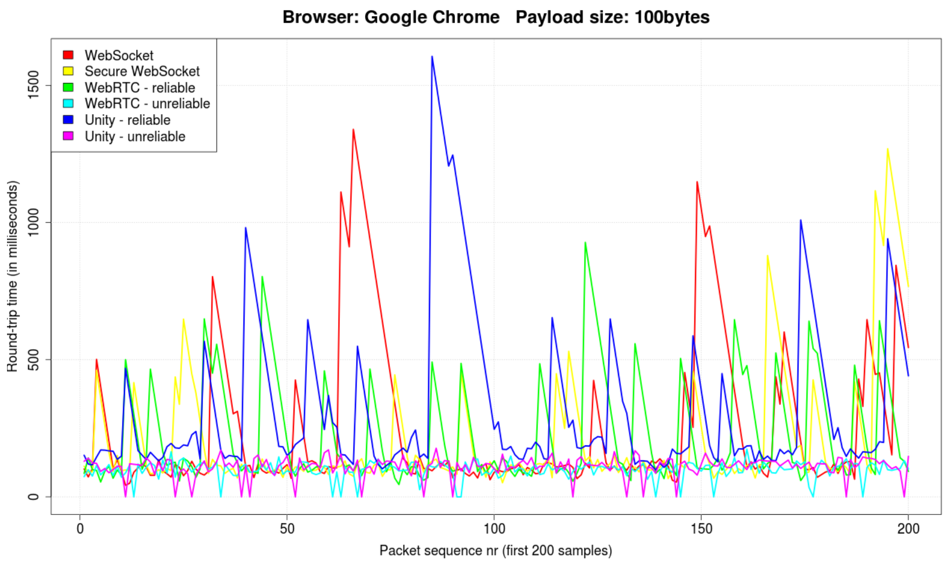


Figure 5.14: Round-trip time-lapse for the first 200 samples in a busy network.

5.4 Summary

This section summarizes the results observed for each connection, before discussing them further in the context of multiplayer games on the web.

5.4.1 WebSockets

The WebSocket connection is the most lightweight connection. It is the least CPU-intensive connection available for the browser and achieves the highest send rate. For typical game updates, the WebSocket connection will only add 2 to 6 bytes of overhead compared to a reference TCP socket, which also makes it a low-profile connection in terms of overhead. When sending packets fast enough (more than 20 packets per second), then the ACK

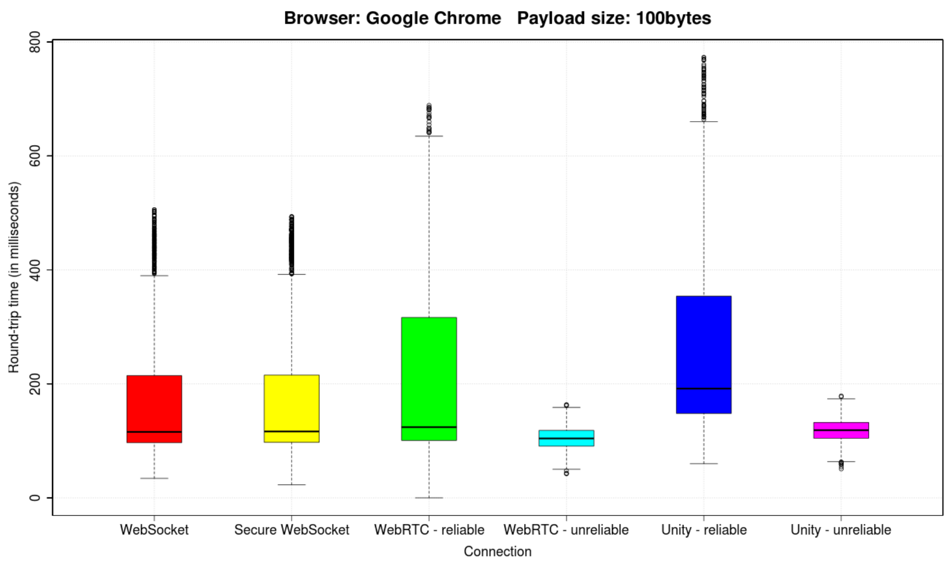


Figure 5.15: Round-trip times for all connections in a busy network.

overhead is also eliminated. One has the option to encrypt the communication by using the Secure WebSocket connection. This adds a little extra overhead on both the network and CPU. The network overhead for the Secure WebSocket connection is dependent on which browser is used. Chrome produces slightly smaller encrypted packets than Firefox. Both WebSocket connections also achieve the lowest delays when used in optimal network conditions, but when packets get dropped, their underlying TCP connection can cause latency spikes due to the ordered and reliable characteristic.

5.4.2 WebRTC

The WebRTC connections are generally less efficient than WebSockets: they add more overhead (header and operational) and are more taxing on the CPU. This is due to many protocol layers and mandatory encryption of the WebRTC data channels. There is no difference between the reliable or unreliable channels in terms of CPU and network overhead, unless packets start to get lost. WebRTC's selective ACK mechanism will always add some overhead, regardless of reliability configuration. The overhead of these SACKs can be minimized by sending packets fast enough. The choice of web browser currently has a large impact on operational overhead. Chrome will produce an excessive amount of STUN traffic, while Firefox is very absent on this aspect. WebRTC does make up for this overhead by providing flexibility in configuring reliability and ordering of the connection, and multiplexing data channels over a single peer connection.

5.4.3 Unity Webplayer

The Unity Webplayer plug-in is the most efficient connection option when it comes to bandwidth usage. It offers a high degree of control over how the data is sent over the connection because it is not restricted by what the web browser exposes to the developer. One can use standard TCP and UDP sockets. The UDP socket has the lowest and most transparent operational overhead. However, this comes at the price of the highest CPU usage due to its slow internal event loop. This high CPU usage is very large for Chrome compared to Firefox. This is due to architectural differences between browsers and how they execute plug-in programs. Communicating data over the Unity Webplayer introduces a high amount of latency and jitter in optimal network conditions, also because of its internal event loop. This effect of latency and jitter, however, is less present when using the connection in a more busy network environment. This is especially true for its UDP connection, which performs similar to WebRTC's unreliable configuration in terms of latency.

Chapter 6

Choosing the Best Connection

Having analysed the results for each connection in the previous chapter, one can discuss their applicability to support multiplayer games for the web browser. One will notice that, compared to a game developer targeting native platforms, the web-game developer is still more restricted in various aspects. The web-game developer will have to make difficult trade-offs.

The different connection options are discussed in their applicability for various types of games, and how they can be used in combination with latency and bandwidth optimization techniques discussed in chapter 3. They are also discussed in the context of the devices on which these games can be played.

6.1 Applying the Connections in Multiplayer Games

The primary topic to discuss is, of course, how well a connection can fit in a certain type of multiplayer game. Some broad categories of games that are frequently played online are discussed in the context of their requirements for game update transmission. The various types of games are broadly categorised in terms of their sensitivity to latency and packet loss. For each such category, one or several connections are recommended to be used for such a type of game. Figure 6.1 shows a possible decision tree that a web-game developer can use to choose a connection for his game.

6.1.1 Shooter and Racing Games

A widely played and very competitive genre of multiplayer games are shooter games, whether they are played in first-person or third-person. Its main requirements are a very low latency and a high update rate. Players usually run around fast and in different directions while shooting multiple times per second. This gives game data a very short relevant time-frame. Although reliability and ordering are always a wanted feature, they can pose a big issue when part of the game updates gets delayed or lost. Reliability and ordering are thus replaced by frequently sending updates to conceal these potentially lost or late updates and to prevent data from being stalled.

Racing games have similar requirements to shooter games: a low latency and frequent updates. The low latency is needed to make the vehicle feel responsive and to timely synchronize the position of the vehicles of all players. Frequent updates are necessary to correct the vehicle's position, direction, speed and acceleration, e.g. to avoid crashing into a tree or oncoming traffic.

Looking at these requirements, one can see that either the Unity Webplayer or an unreliably configured WebRTC data channel matches these requirements. In these scenarios, one would need to make the trade-off between bandwidth and latency. The WebRTC connection is a fast connection, but it produces a very high overhead. Especially for these games, where packets are already very small, the WebRTC overhead will quickly reach values between 200% and 300%. The rest of WebRTC's operational overhead is spread over the many game updates, but this will be largely implementation-dependent, e.g. the frequency of STUN requests. The Unity Webplayer's UDP connection on the other hand, produces very little overhead but introduces latency and jitter that depends on Unity's internal event loop. This jitter however, mostly blends in with the latency and jitter of the network. The increase in latency is still slightly present. This means that, if the developer decides that one can cope with the extra bandwidth, then the WebRTC connection is the preferred choice of communication. But if latency can be minimized by, for example, placing enough servers in good locations, then the latency increase of the Unity Webplayer can be less of an issue, and thus Unity's UDP connection can be used and much network resources are saved.

The bandwidth usage can be further mitigated by applying bandwidth optimization techniques such as dead-reckoning for positional prediction, and delta-updates to reduce the size of updates. Dead-reckoning will reduce the amount of updates sent, and is especially effective when the character or vehicle keeps going in a straight line. Delta-updates will reduce total bandwidth consumption by packing the numbers in less bytes. This does, however, increase the relative overhead because the overhead remains fixed for the amount of data being transported. For the delta-update technique, an additional reliable channel should also be used to periodically send a reference state. This could be any of the three reliable connections available. When a WebRTC connection is chosen, one must also be aware that creating such delta-updates is best done using JavaScript's Typed Arrays. Update filtering is also able to reduce the amount of packets received from other clients. Since these types of games are focussed on what is directly in sight, updates from what is happening behind the player or around the corner are not necessarily needed.

If a client-server architecture is used, then the developer must also choose optimal locations for the servers to minimize latency. When not taking place in a persistent online world, most of these games also allow to be implemented in a peer-to-peer architecture. For peer-to-peer, the only (immediate) solution is to implement the game using a WebRTC connection. But, as will be suggested and discussed later, the Unity Webplayer could still be of use in this scenario.

6.1.2 Action and Role-playing Games

Action and role-playing games, including massively multiplayer games, can have a somewhat slower pace than the shooter games described above. The avatar that the player has under his control can move around, but its exact position is usually of less importance than in fast-paced games. The actions this avatar can execute usually also have some kind of build-up or activation time before any effect is applied on other players. This makes that these games are a bit more tolerant to latency, and that reliable transport protocols are an option.

This means that all possible options are on the table, and much depends on more specific requirements of the game. One need not choose, however, between either a reliable or unreliable channel. Both types can be used, a dual channel approach. If, for example, the game boasts free-movement combat, then maybe the best option is to choose an unreliable channel for just the positional information and look at the arguments given above to make a choice. Then, a reliable channel could be used for the avatar's other actions such as swinging a sword or casting a spell. Otherwise, a reliable channel for all game updates can be chosen.

The choice of a reliable channel is faced with the same trade-off that had to be made for fast-paced games: latency or bandwidth efficiency. The WebSocket connection performs the fastest of all connections and uses the least CPU power, but adds a slightly higher protocol overhead compared to the standard TCP socket in Unity. The developer must also be aware that, if the game is played in an HTTPS web page, then only a Secure WebSocket can be used. The browser does not allow to 'downgrade' the security of the page, even if it's not needed. This would mean that the overhead becomes a bit larger, and shifts the favour, here too, to the Unity Webplayer. If security is desired, then a Secure WebSocket is the better option, unless a custom security mechanism will be developed.

The bandwidth and latency optimization techniques are also of great importance here. Aggregation of updates may be possible, e.g. packing a positional update and ability activation in the same update packet. Applying update filtering by making use of zones in the world can reduce the amount of packets the client receives. The other strategies discussed above can be applied here as well: dead reckoning, delta-updates and good server placement.

Here, the WebRTC connection should not be considered due to its high overhead, unless a peer-to-peer architecture is involved, or one would decide on using the multi-channel approach. Remember that WebRTC can multiplex multiple data channels over one peer connection, which optimizes overhead efficiency and bandwidth usage. Multiplexing more than one channel lowers the relative protocol overhead, and may eventually put it on par with the WebSocket and Unity connections if one manages to send enough data through the channels. The WebRTC data channels are also secured by default.

6.1.3 Puzzle and Strategy Games

Many game genres are somewhat slower-paced than the games described above, but therefore not less competitive. Common slow-paced games are strategy and puzzle games. The update pattern for strategy games and puzzle games, is based on the player's input rather than the entities he controls. This means that updates are only, almost exclusively, needed when the user triggers some event by giving input, e.g. by pressing a button. However, each update should be sent reliably because they contain information explicitly requested by the player. Allowing such an update to get lost may confuse the player and cause frustration. These games also have a relatively high tolerance towards latency, as long as the player is given some feedback about his actions, which improves perceived latency.

Since latency is less of an issue here, one can focus on the bandwidth efficiency and processing power. If the game is to be played in a peer-to-peer architecture, then a (partially) reliable WebRTC connection is the way to go. In a traditional client-server setup, the WebSocket and Unity connections are the two best possible options. However, unlike in the discussion above, where Unity's TCP connection could be used, in this situation, sending updates will be relatively infrequent. This means that Unity will mostly be sitting idle and consuming CPU resources, waiting for a packet to process. Especially for battery-powered devices, this is not desirable. In this case, the standard WebSocket connection is the preferred solution. And

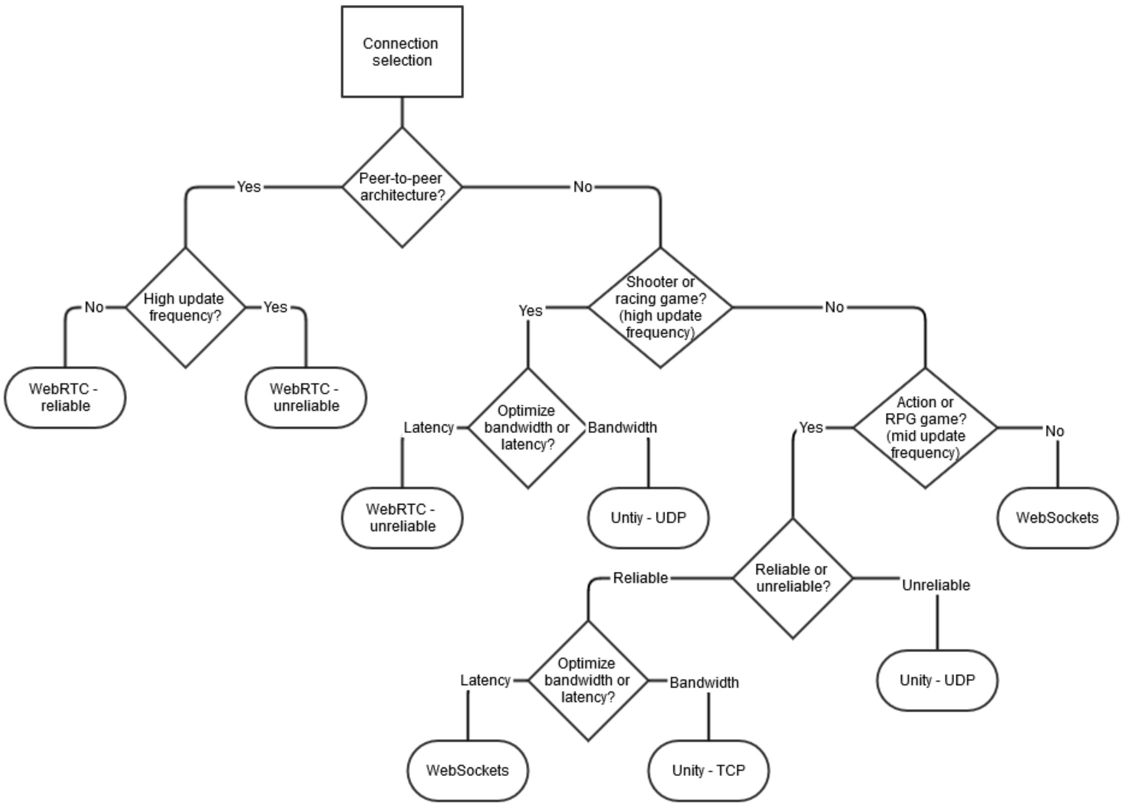


Figure 6.1: Possible decision tree for choosing a connection.

even the Secure WebSocket solution would still be justified. Also, because the frequency of the updates is far lower compared to other game genres and dependent on user input, these extra bytes will only contribute marginally to the total amount of bytes used during a game session.

Optimizing bandwidth and latency for input-driven games is harder because sending input is, on its own, already an optimization. However, update filtering can be applied here too. If, for example playing a real-time strategy game with *fog of war*, receiving detailed updates about players that are not yet visible should not be necessary. Nonetheless, good server placement is important here too.

6.2 Using the Connections on Different Devices

Another important factor to take into account is the target platform of the game and the infrastructure behind the connections to support the game. Indeed, the web browsers on mobile devices may be a potential target platform, and one must keep in mind its limitation when using a connection on such a device. But this discussion is not limited to client devices alone. The server-side also plays an important role in keeping the game online and can also face some limitations.

6.2.1 Laptop and Desktop Systems

Devices such as laptops and desktops are relatively powerful devices and are usually connected to a decent wired or wireless home network. In such cases, most players will hardly care if a connection uses a few extra bytes per connection, or that the connection uses a little extra CPU resources.

6.2.2 Mobile Systems

Mobile devices such as smartphones and tablets have become a large group of devices that also have access to the web. Their browsers too support HTML5 technologies, and some even WebRTC. And although they can connect to Wi-Fi networks at home or work, they are often connected to less stable, mobile networks as well, such as *3G* and *4G* wireless networks. Not only do these networks have a higher latency and packet loss, they also have a higher financial cost associated per byte that is transmitted. Additionally, mobile devices are battery-powered and are usually less powerful on the CPU-front. This makes choosing the right connection option especially important.

Using the Unity plug-in has a higher CPU cost, while WebRTC has a higher bandwidth cost. So when possible, WebSockets would seem to be the obvious choice. However, because the mobile networks exhibit a lot more packet loss, one should carefully analyse the latency limits for the playability of the game and then decide on a connection to go with.

The choice of a browser here may also be of great influence. Firefox uses more CPU resources for the browser-native connections compared to Chrome, but for the Unity plug-in, the opposite is observed. But this is a choice that lies in the hands of the player, not the developer. He can only make the player aware of the implications the choice of browser has on the player's device.

6.2.3 Server Systems

Not only the systems on which the game will be played are important. In cases where a game server is involved, one must also keep in mind that the server will be a central meeting point for all game data. If a connection with a lot of overhead is chosen and many clients are connected, then the server-side of the game may start to become a bottleneck. This is certainly true when many Chrome WebRTC clients would be used to play the game. The server will then have to respond to a lot of STUN requests which adds no value to the game. Additionally, costs associated with bandwidth would also have a negative impact. Keeping the security of the connection in mind is also important. The server must encode and decode each of the packets it sends and receives, adding to the load.

A possible solution to this problem can be found in the ALVIC-NG architecture discussed in section 3.1.2. Proxy servers at the edge of the server-side can be made responsible for decoding and encoding of game data, as well as responding to STUN requests in order to lift this effort from the core server machines. The actual game update data can then travel on a light-weight transport protocol within the server architectural system.

6.3 The Unity Webplayer Plug-in as a Communication Channel

When using the Unity Webplayer plug-in as the preferred choice of communication channel, then one can also discuss whether or not to just develop the entire game for the plug-in. Unity's biggest disadvantage as just a communication channel stems from the additional latency introduced due to its update loop when an update is communicated over the network. But this disadvantage disappears the moment the game would run entirely inside the plug-in. As could be seen from figure 5.5, the latency and jitter disappears and performs equally well compared to the WebSocket connections. Its other disadvantage is that its CPU usage is higher than the other communication channels, but using the update loop to actually render the game as well, optimizes its usage.

However, there are still good reasons why one would opt to use a plug-in as just a communication channel. WebRTC is currently not supported on all popular browsers, e.g. Internet Explorer. Previous versions of the browsers that support WebRTC data channels now, may have only partial support, or none at all. Then a plug-in could still be used as a fall-back mechanism or as a secondary option, as is done in the *socket.io* WebSocket implementation. In this case, the Unity Webplayer's UDP connection can then be used for unreliable and / or unordered communication if the game requires it. The plug-in is supported in a much broader set of browsers and their older versions. In some cases, Unity's TCP connection is still preferred above a WebSocket connection, i.e. when playing on a web page secured by HTTPS. Although it are Unity's capabilities that are tested in this thesis, another plug-in may also provide a standard TCP socket, which may not be limited by a frame rate and may consume less CPU resources. Adobe's

Flash Player and Microsoft's Silverlight, for example, provides such a TCP socket.

This does not, however, solve the problem of a developer that wants to deploy a peer-to-peer architecture. For players with older browsers that would use the Unity Webplayer, a kind of proxy-relay server could be set up. This server then acts as a WebRTC relay and traffic translator to communicate with the WebRTC peers. This is similar to the TURN relay servers used for WebRTC when two peers can't directly connect to each other. The server could also intelligently filter out redundant updates coming from multiple peers containing the same information to save bandwidth. Such servers would also need to be deployed on multiple locations so that players can connect to a server that is located 'nearby' for all peers. Of course, when all or a large majority of the players falls back on the Unity Webplayer, then this boils back down to a client-server architecture, with the difference that the server just performs a relay and filtering function, and no authorization of player actions. Figure 6.2 illustrates the idea.

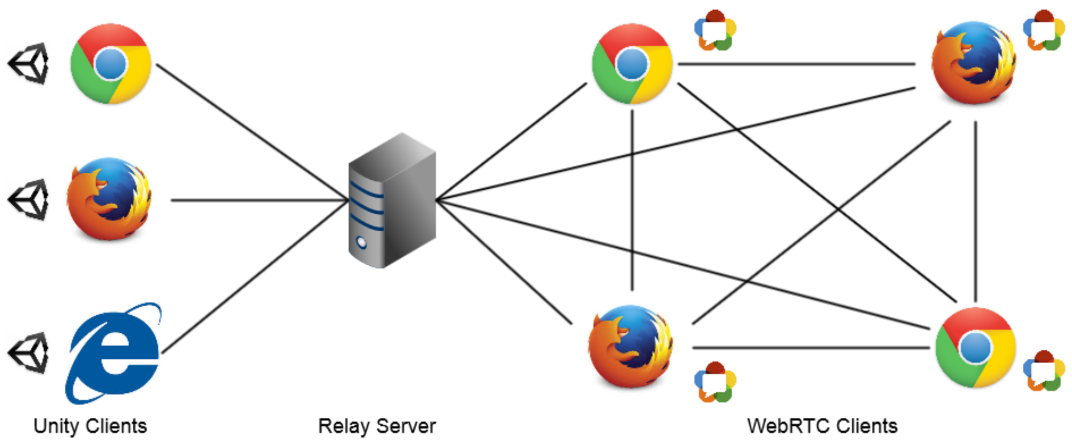


Figure 6.2: Using the Unity Webplayer as a fall-back connection in a full-mesh WebRTC peer-to-peer setup.

Chapter 7

Conclusion

In this thesis, the different communication options that can be used for multiplayer games in the web browser have been analysed and evaluated. WebSockets and its secure variant, Secure WebSockets, the new and experimental WebRTC API, and the Unity Webplayer plug-in have been compared in their capabilities to support different types of games that can be played in multiplayer games.

All but two of the expectations that were formulated in chapter 4 were met.

It was expected that both browsers would perform the same for the browser-native connections because they follow the specifications for such connections. However, there are differences between Chrome and Firefox for the Secure WebSocket and WebRTC connections. They produce a different overhead for each of them.

The expectation that the Unity Webplayer would have a significant latency penalty is only somewhat fulfilled. In optimal network conditions, it does produce a relatively high amount of extra latency and jitter. But in a more realistic network setting, this latency and jitter more or less blends in with the latency and jitter introduced by the network, making it behave very similar to the standard WebRTC and WebSocket connections.

The following list enumerates each connection option with its associated characteristics:

- The WebSocket and Secure WebSocket connections are the fastest connections a browser can use, and additionally, require the least CPU resources. The standard WebSocket only adds little overhead. The secure variant adds some more overhead, but depends on the browser. However. due to their underlying reliable and ordered transport protocol, they are not suited for fast-paced games.

- WebRTC's biggest advantages are its support for peer-to-peer architectures and very flexible API to configure the data channels. This comes at the cost of the highest overhead, and a slightly higher CPU usage compared to WebSockets. A part of this overhead is also browser-dependent, but this may change over time when WebRTC moves on to become a real web standard. WebRTC also has mandatory encryption.

- The Unity Webplayer plug-in has the most bandwidth-efficient communication channels, but is the most CPU intensive connection. Its feature of supporting a standard UDP socket is its biggest advantage. It allows to easily build a custom protocol that can add support for ordered and / or reliable delivery of game data if the game requires it.

With HTML5, a web-game developer already has the tools to create fancy games, but now he can also add a multiplayer aspect to it. However, compared to a game developer for native platforms, more trade-offs have to be made. The most prominent trade-off is that of latency versus bandwidth. One cannot achieve optimal latency values without also adding more overhead, and thus increasing bandwidth usage. When a game is more latency-resistant, that is it doesn't require frequent updates and a reliable connection is used, then the CPU versus bandwidth usage will be the trade-off to ponder about.

A big challenge that still lies ahead for game and browser developers, is the support for all these technologies across all different browsers and platform. With Mozilla Firefox and Google Chrome, and in the mean time Opera as well, supporting the connection options analysed in this thesis, a large target audience can already be addressed. However, not all browsers have support for each technology yet, most noticeably WebRTC. A developer can opt to use fall-back mechanisms for these players, or advise them to use a different browser to play their game.

Chapter 8

Acknowledgements

After writing this thesis entirely from an aloof perspective, it would seem distant and cold if this was the case for this chapter as well. So this is the perfect place to get down and personal. First of all, I would like to thank my promotor, Prof. dr. Wim Lamotte, not only for his patience with my lack of communication, but also for his enthusiasm and passion during his lectures. And most importantly, his detailed feedback on my works in progress. I would also like to thank his team members, Prof. dr. Peter Quax and dr. Jori Liesenborgs for their valuable tips and feedback during those rare moments we met.

I spent a large part of my time at LuGus Studios. They provided me with a good amount of laughs and an enjoyable environment that made this work possible. So a big thank you to the entire LuGus crew and the numerous interns I had the pleasure to work with. However, I would like to mention and thank Robin Marx in particular, for without his insightful feedback, honest critique and strive for perfection, I would probably still be dabbling in non-relevant topics and words.

The last group of people to whom I wish to express my gratitude, and one that is a big cliché, are of course my friends and family. Their continuous support and understanding throughout this period is of great value to me. To my parents and brothers: thank you for providing such a warm and lovely home. To my good friends: thank you for the, sometimes necessary, reminders that there are other things in the world than to work on a thesis. A particular thank you to Keke Kokelenberg, who had too much free time on his hands and corrected some overlooked spelling and grammatical mistakes.

Appendices

Appendix A

Implementation Details

This chapter delves into more details about the simulation setup, the software used, the programs developed specifically for this thesis, and some noteworthy encountered implementation problems.

A.1 Devices Setup

Only three physical devices are directly used in all of the test cases:

- A client machine acting as a packet source,
- A server machine acting as a packet reflector,
- A router connecting the client and server machine, and serving as a gateway to the internet.

They are placed inside a local network, and connected to each other over wired 10/100/1000 Mbps connections. No wireless communication is used inside the network to minimize interference and possible transmission errors. The router is connected to the internet to reach a STUN server for when a WebRTC connection is used. This is a fourth physical device outside the local network, but other than obtaining the public address for WebRTC connections, this STUN server is not used.

This network setup is heavily inspired by the setup used in a performance evaluation of XMPP over web technologies, by Laine and Säilä. [34] A similar setup is used by Dianotti et al, where they test and compare the throughput of SCTP to that of TCP and UDP. [13]

A.2 Server-side Setup

The server-side of the project is stationed in a virtual machine running *Ubuntu 12.04 LTS 32bit*, with 4GB of RAM and 4 dedicated cores running at 2.2GHz. On the server, Node.js is used as the main server application, and performs three functions:

- Serve the web pages that contains client-side code
- Management and coordination of the connected client and current test, and
- Act as a reflector for incoming packets of the communication channel being tested.

When the Node.js server application is started, it will start up several connection modules. Each connection module manages one specific type of connection, functioning on a separate port number. Some modules are always initiated when the Node.js application starts, while others are only initiated upon request. These modules are detailed further down in this section. The following connection modules are available:

- An HTTP web server that serves the client-side simulation scripts and accompanying web page. This connection module is always initiated by the server.
- A WebSocket control server for managing clients and coordinating the current test scenario. It is also used by the clients to send their results to the server, so that it can gather and compile the scenario results. This connection module is always initiated by the server.
- A WebSocket signaling server for exchanging WebRTC signaling information. This module also contains the functionality to set up WebRTC peer connections. It is only started when a WebRTC connection will be used.
- A WebSocket connection server that is used for the actual WebSocket testing. This connection module is only started when the WebSocket connection will be used.
- A Secure WebSocket connection server that is used for the Secure WebSocket testing scenarios. This connection module is only started when the Secure WebSocket connection will be used. This module also requires a certificate to secure and encrypt the connection. The certificate used here is a self-signed *OpenSSL* certificate.
- A connection module with both regular TCP and UDP servers that is used for the plug-in simulation. This connection module is only started when the plug-in connection will be used.

Besides the Node.js server application, four other applications also run on the server system. Firstly, a *socket policy* server application is used that is addressed whenever a plug-in is used at client-side. This socket policy server acknowledges the plug-in at the client that the server will accept the incoming data. The returned socket policy file here allows connections from all domains on all ports. In a production environment, these values should be more restricted to only accept connections from trusted domains. The piece of XML below shows the policy file returned. This socket policy server is provided by installing the Unity development suite.

```
<?xml version='1.0'?>
<cross-domain-policy>
        <allow-access-from domain=""*"" to-ports=""*"" />
</cross-domain-policy>
```

Listing A.1: Socket policy file contents

Secondly, *Wireshark* or rather its console variant, *Tshark*, is used to monitor incoming and outgoing network traffic. For certain test cases, traffic is analysed to measure the effects of latency and packet loss, as well as to analyse the protocol and operational overhead for each connection.

Next, *tc*, a traffic shaping program is started for test cases where the effects of packet loss, delay and delay jitter are examined. This traffic shaping program is used only at server-side, but is applied on both incoming and outgoing traffic. For incoming traffic, an *intermediate functioning block* is created that buffers the traffic and where delay, jitter, and packet loss are executed upon. For outgoing traffic, the normal network device is used.

Finally, a management application runs in the background, and can be considered as the coordinator of the other applications, including the client-side. When starting this management application, various parameters can be configured. Based on these parameters, this application will create a configuration file that describes all the different test scenarios that will be executed. An example of a few test scenario descriptions is given below. For each such a test scenario, it will reset the Node.js server application with the required configuration and start capturing network traffic. It also signals a client-side management application by letting it know when a test scenario has finished executing so that it can clean up.

```
{
  "connectionInfo": {
    "ipAddr": "127.0.0.1",
    "port": "8080"
  },
  "sessions": [
    {
      "simulationType": "interval",
      "browser": "chrome",
      "connection": "websocket",
      "clients": 1,
      "duration": 60,
      "pps": 30,
      "payloadSize": 1000,
      "packetCount": 10000,
      "sessionNr": 1,
      "wireshark": false
    },
    {
      "simulationType": "interval",
      "browser": "chrome",
      "connection": "webrtc - reliable",
      "clients": 1,
      "duration": 60,
      "pps": 30,
      "payloadSize": 1000,
      "packetCount": 10000,
      "sessionNr": 1,
      "wireshark": false
    },
    ...
    {
      "simulationType": "pingpong",
      "browser": "chrome",
      "connection": "webrtc - unreliable",
      "clients": 1,
      "duration": 60,
      "pps": 30,
      "payloadSize": 1000,
      "packetCount": 10000,
      "sessionNr": 1,
      "wireshark": false
    },

  ...
  ]
}
```

Listing A.2: Test case configuration examples

A.2.1 WebSocket Module

WebSockets are not a standard part of Node.js. Therefore, a module that implements WebSocket functionality is imported and coupled to the server application. Several modules that offer a WebSocket implementation exist, e.g. *ws* [58], *socket.io* [4] and *WebSocket-Node* [69].

In the current server application, the first listed implementation is chosen: ws. At the time writing it is actively maintained and offers a reference WebSocket interface as it is available in the web browser, as well as conforming to the latest WebSocket standards. WebSocket-Node is also a reference implementation but is not actively maintained any longer, and also performs slightly worse according to benchmarks. [68] Socket.io is a popular WebSocket module, but is not considered due to its different API interface. It also adds another layer on top of the WebSocket header to support 'rooms'. This can have an influence on measurements of packet size as well as on the processing delay. Socket.io also features many fallback mechanisms such as a plug-in socket and an HTTP-based technique. This could lead to an unwanted increase of load on clients.

The ws module for Node.js is also used to create the Secure WebSocket connection module. Using Secure WebSockets requires certificates to secure the connection. These certificates are self-signed by using OpenSSL.

Using the ws module allows to very easily set up a WebSocket server and open a standard WebSocket connection with the client. The following piece of JavaScript code illustrates how the WebSocket server is used in the server application.

```
var WebSocketServer = require("ws").Server;
var wsServer = new WebSocketServer({"port": 9003});

wsServer.on("connection", function (ws) {

  ws.on("message", function (data) {
    // Process incoming message
  });
  ws.on("close", function (code, message) {
    // Process connection closing
  });
  ws.on("error", function (error) {
    // Process error
  });

});

console.log("WebSocket server started listening on port 9003");
```

Listing A.3: WebSocket server set up

A.2.2 WebRTC Module

WebRTC is also not natively supported by Node.js and requires a module to be imported. The options for WebRTC implementations are limited compared to WebSockets. The only available module for Node.js that has support for all features of WebRTC is *node-webrtc* [30]. On its own, node-webrtc only implements an interface for Node.js that equals the interface for WebRTC in the web browser. However, during installation of the module, it fetches and compiles the complete WebRTC source code that is used in

Google Chrome to build the WebRTC library. Afterwards, it is coupled to the interface for Node.js, which allows Node.js to be used as a regular WebRTC peer. See section 2.4.4 for a code example on how a peer connection is initiated.

In the server application, a separate standard WebSocket channel is used to support the WebRTC connection and acts as the signaling channel. This WebSocket uses the same ws Node.js module that is introduced above. In a production environment, this signaling channel should be a secured one. It carries sensitive information that is used to encrypt the WebRTC data channels. Sending this data unsecured may allow a third party to eavesdrop on the communication line. Once the WebRTC connection setup has been completed, the signaling channel is closed because it is not further needed in the current implementation.

A.2.3 Native Module

For communication with the plug-in at client-side, regular TCP and UDP sockets are used at server-side. A TCP server as well as an UDP socket are readily available in Node.js, so there is no need in importing separate modules.

```
var dgram = require("dgram");
var net = require("net");

udpSocket = dgram.createSocket("udp4");
udpSocket.bind(9007, "127.0.0.1");
udpSocket.on("message", function (message, rinfo) {
  // Process incoming message
});

console.log("Native UDP server started listening on port 9007")
    ;

var tcpServer = net.createServer( function(socket) {

  socket.on("data", function (message) {
    // Process incoming message
  });
  socket.on("close", function(hadError) {
    // Process connection closing
  });
});

tcpServer.listen(9007, "127.0.0.1", function(){
  console.log("Native TCP server started listening on port 9007
      ");
});
```

Listing A.4: Native server set up

A.3 Client-side Setup

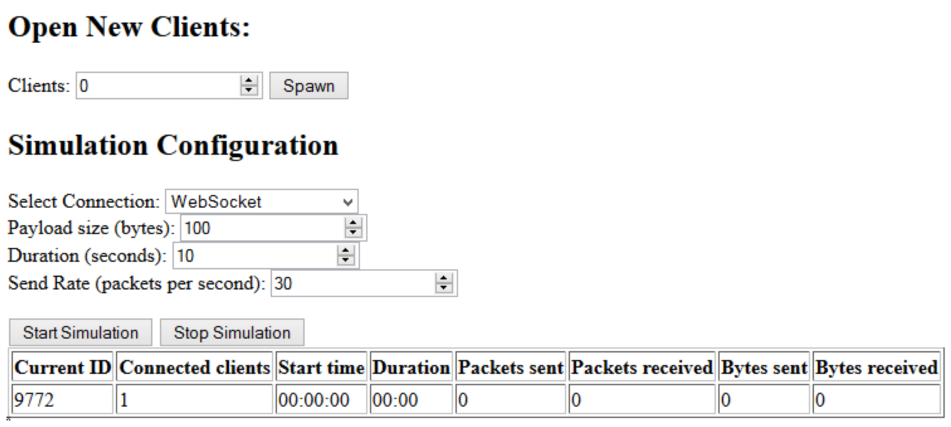
At client-side, all test cases are executed on a machine running *Windows 8.1 64bit*, with 8Gb of RAM and 4 dedicated cores running at 3.4Ghz. As the client-side simulation application, readily available browsers are used: Google Chrome (version 36) and Mozilla Firefox (version 31). Both these browsers support the latest version of the WebSocket protocol, have interoperable support for WebRTC, and can run the Unity Webplayer plug-in. Both browsers are also popular and have a large market share. The browser's main function during a test case is to send and receive packets at a certain sending rate and collect timing information about them. The Opera browser could've been taken into the equation as well but, at the start of this thesis, it did not support WebRTC data channels yet.

At the client, some side-applications are used as well. Tshark is used to monitor incoming and outgoing traffic during certain test cases. A client-side management application runs in the background that is connected to the server-side management application through a simple TCP connection. The client-side management application will receive signals from the server whenever a simulation starts and ends. When starting a test case, the management application will open up the web browser and immediately directs it to the web server. If a stop signal is received from the server, it will close the browser. During a test case, it also monitors the CPU and memory usage of the web browser.

A.3.1 Web Application

The web application is a single web page that is downloaded from the server every time a tab is opened during a test case. The web page has a very basic structure and limited user interface. Figure A.1 shows the user interface. The user interface allows to set some parameters to rapidly create and execute a certain test case. Only one tab needs to be configured with the desired simulation parameters. Once the 'Start Simulation' button is pressed, the configuration is compiled and sent to the server, which will then spread the configuration over the other connected clients. A table is also present in which some quick results can be viewed for that client. This user interface is hardly used during the execution of the test cases, it is for quickly testing a set of parameters only.

The bulk of the work is done in a few JavaScript scripts behind the scene. When the plug-in solution is used during a simulation, then a `C#` script is used that drives the plug-in. The important scripts are detailed below.



Open New Clients:

Clients: 0 Spawn

Simulation Configuration

Select Connection: WebSocket
Payload size (bytes): 100
Duration (seconds): 10
Send Rate (packets per second): 30

Current ID	Connected clients	Start time	Duration	Packets sent	Packets received	Bytes sent	Bytes received
9772	1	00:00:00	00:00	0	0	0	0

Figure A.1: Client-side web application

Switch.js

The *switch.js* script is in essence a manager for the different types of connections that can be used by the client. It manages, at most, five types of connections: a WebSocket control channel, a WebSocket and Secure WebSocket data channel, a WebRTC peer connection, and the Unity Webplayer. A control channel is always created by the script, but the other connection types are optional. It features a very simple interface to quickly switch from one connection to an other. The interface is shown in the code piece below.

Whenever a message has to be sent, it can be given to the script, and it will take care that the correct connection is used. The interface, however, only allows to send data, not receiving data. Upon initialization, connections are configured with a configuration object that contains a reference to a callback function for when data is actually received on that connection.

```
Network.Switch = Class.extend({
  init: function(configuration) {
    // Constructor
    // Creates a list of connections based on the provided
       configuration with callback references for receiving
       data
  },
  setActiveConnection: function(id, label) {
    // Switches to another connection with the provided ID, e.g
       . "websocket", "securewebsocket", "webrtc", or "unity"
    // The label parameter is used when a connection supports
        multiple channels, e.g. "reliable", or "unreliable"
  },
  send: function(message) {
    // Send a message over the currently active connection
  },
  sendControl: function(message) {
    // Send a message over the control connection
  },
  open: function() {
    // Returns a boolean indicating whether all connections are
         open for communication
  }
});
```

Listing A.5: Switch.js script object interface

Simulation.js

The *simulation.js* script is the main script used by each client running inside the browser. Its main responsibilities are the initialization of other scripts and objects such as switch.js, generating and processing of control and data packets, as well as keeping track of the round-trip times.

When a client in the web browser requests the web application page, the server will first modify the web page to set some boolean values that indicate which connections are to be initialized by the client. When the simulation.js script is downloaded and executed, it will use these values to create a configuration object for the switch.js script, which will, in turn, only set up the requested connections. This allows to limit the client and server to only use the necessary connections and avoid clutter of the memory space.

After initializing the switch.js script, it will create a WebWorker that is used as a timer mechanism for generating packets at regular intervals. Why this WebWorker is spawned, and no ordinary timeout mechanism is used, is explained in section A.4.

When the script receives a signal from the switch.js script that all connections have been opened, the simulation.js script will contact the server over the control channel to request an ID and gather knowledge about other connected clients. Hereafter, the script will go dormant, and only wake whenever an event of interest happens, i.e. a button is pressed, a simulation is started, a message is received, etc.

Whenever a test case is started, the script will start generating packets in regular time intervals, depending on the send rate of the scenario parameters. Before the first packet is sent, however, the script will create a packet skeleton structure that is used afterwards for each packet that is sent. This skeleton structure contains the client ID to identify the source of the packet, a sequence number, and a randomly generated payload that stays fixed during the rest of the simulation.

Next, packets are generated using the skeleton structure and can be sent immediately. Timestamps are generated using the `performance.now()` function and stored for measuring the round-trip time when the reply is received.

When the simulation ends, the timestamps are compiled in a control packet and sent to the server. The server will save the results to disk for later analysis.

```
var payload = "";
var randomSelection = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

// Generate the random payload - fixed portion of the payload
    is 34 bytes
var randomSectionLength = Math.max(1, desiredPayloadSize - 34);
for (var i = 0; i < totalPayloadLength; ++i) {
  payload = payload.concat(randomSelection.charAt(Math.floor(
      Math.random() * randomSelection.length)));
}

// Skeleton object for the message
var skeletonPacket = {
  id: currentID,
  nr: "00000",
  data: payload
};
```

Listing A.6: Packet skeleton structure

```
// Send message
var message = JSON.stringify(data);
timestamps.push([performance.now(), null]);
networkSwitch.send(message);

...

// Receive message
var timestamp = performance.now();
var data = JSON.parse(message);
var index = parseInt(data.nr);
timestamps[index][1] = timestamp;
```

Listing A.7: Sending and receiving packets with timestamping

NativeClient.cs

Whenever the plug-in is used as a communication channel, then the *Native-Client.cs* script inside the Unity Webplayer is used. It is the only script inside the plug-in, and manages a reference TCP and UDP socket over which it can send and receive data. It also stores a set of timestamps for each packet that is passed through.

On initialization, the NativeClient script will wait for the browser to communicate configuration details, e.g. server address information, preferred communication channel (TCP or UDP), on message callback function, ... Once this information has been received, the script will create the desired connections and start listening for incoming messages from the browser and network.

Whenever the browser wishes to send a message over the plug-in, the `SendData()` function of the plug-in is called from the browser-side along with the data to be sent. The NativeClient script will then extract the client's ID and the packet's sequence number (these values are part of the data to be sent in this implementation) to couple a timestamp to this data. The data is then put on the network by using the preferred connection. Sending data is done in an asynchronous way. This is to prevent other messages that need processing from being held up until this packet is finally put on the network.

Receiving data from the network is also done asynchronously. When the NativeClient script has been configured by the browser, it will start listening on the socket for incoming data in a separate thread. When data is received on the socket, a callback function is executed that will process the incoming data, and store a timestamp when a complete message has been received (remember that the TCP socket uses stream-based communication). This timestamp marks the arrival of the message in the plug-in. The callback function will temporarily store the received data in a buffer.

Whenever Unity's event loop arrives at its `Update()` function in the main thread, it will process the complete messages stored in the buffer and pass them through to the browser. This pass-through of messages can only be done in the Unity Webplayer's main thread because it does not allow other threads to use these functions. When a message is passed to the browser, a last timestamp for that message is stored, which marks the time the packet is passed on to the browser.

Because a standard TCP socket is used inside the Unity Webplayer, messages need to be delimited in some way due to TCP's stream-oriented nature. Luckily, the messages sent in this implementation are in a JSON format and have a strict data structure. This makes it easy to find the end of the message.

```
public class NativeClient : MonoBehaviour
{
```

```

private List<string> messageBuffer;
private UdpClient udpSocket;
private TcpClient tcpSocket;

protected void Update()
{
  if (messageBuffer.Count > 0)
  {
    // Extract messages from the buffer

    ...

    // Send to the browser
    Application.ExternalCall(onMessageCallback, message);
  }
}

public void Init(string nothing)
{
  // Initialize TCP and UDP socket and other objects
  udpSocket = new UdpClient();
  tcpSocket = new TcpClient();

  ...

  // Start listening for incoming packets from the network
  udpSocket.BeginReceive(new AsyncCallback(OnMessageUDP),
      null);
  tcpSocket.GetStream().BeginRead(TCPBuffer, 0, 1024, new
      AsyncCallback(OnMessageTCP), null);
}

public void SetServerAddress(string address)
{
  // Configures the plug-in with the server address
}

public void SetServerPort(string port)
{
  // Configures the plug-in with the server port
}

public void SetActiveChannel(string type)
{
  // Configures the plug-in to use either the TCP or UDP
      connection
}

public void OnMessage(string functionName)
{
  // Configures the plug-in with the name of the callback
      function in the browser
}
```

```

  public void SendData(string message)
  {
    byte[] data = Encoding.UTF8.GetBytes(message);

    switch (channel)
    {
      case Channel.RELIABLE:

        tcpSocket.GetStream().BeginWrite(data, 0, data.Length,
            new AsyncCallback(EndTCPSend), tcpSocket.GetStream
            ());
        break;

      case Channel.UNRELIABLE:

        udpSocket.BeginSend(data, data.Length, new
            AsyncCallback(EndUDPSend), udpSocket);
        break;
    }
  }

  private void EndTCPSend(IAsyncResult result)
  {
    tcpSocket.GetStream().EndWrite(result);
  }

  private void EndUDPSend(IAsyncResult result)
  {
    udpSocket.EndSend(result);
  }

  private void OnMessageTCP(IAsyncResult result)
  {
    // Process incoming data on the TCP socket
    // and put the data in a buffer
  }

  private void OnMessageUDP(IAsyncResult result)
  {
    // Process incoming data on the UDP socket
    // and put the data in a buffer
  }
}
```

Listing A.8: NativeClient.cs

A.4 Encountered Implementation Problems

Making the web application behave identically in each web browser is not always a straightforward process. Some noteworthy problems were encountered during implementation that are either general browser behaviour, or a browser-specific thing.

Limited Callbacks in Inactive Tabs

A problem encountered in both Google Chrome and Mozilla Firefox, is that inactive tabs are limited in the amount of callbacks per second for the `setInterval()` function. This function will call a provided function in regular time intervals. The function in the web application is mainly used to regulate the packet send rate for each tab in the web browser. However, when a tab is not selected as the current active tab, then it is limited to calling the callback function only once a second. This is implemented in web browsers as an optimization feature to prevent inactive tabs from using too much CPU time. This also means a client, represented by a tab in the browser, would be limited to only sending one packet per second, severely limiting simulation scenarios.

Firefox allows to modify the value that imposes this limit. By modifying the `dom.min_background_timeout_value` value from `1000` to `1` (values are in milliseconds) in the Firefox-specific `about:config` configuration page, this limit is lowered from once per second, to once per millisecond. Chrome, however, does not allow to adjust this setting, which begs the need for another solution.

Using an HTML5 WebWorker, this problem is circumvented. Although inactive tabs are only limited to call back a function once a second, WebWorkers are not. Thus, in the final implementation, each client running in a tab will spawn a WebWorker upon initialization. This WebWorker will run the `setInterval()` function and send a message to the main thread whenever a certain function should be called. The piece of JavaScript code below shows the WebWorker code that is used to circumvent the callback limit in both Firefox and Chrome.

```javascript
var timerID;

self.addEventListener("message", function (event) {

  switch (event.data.code) {
    case "start":

      timerID = setInterval(function () {
        self.postMessage("send");
      }, event.data.sendInterval);

      break;

    case "stop":
        clearInterval(timerID);
        break;
  }

}, false);
```
Listing A.9: Callback limit omission using a WebWorker

Different Browser Architectures

Each time a test scenario is run, the client-side management application will monitor the resources consumed by the browser. At first glance, Google Chrome seemed to perform consistently and significantly better than Mozilla Firefox. However, this was not taking into account the difference in architecture between browsers.

Firefox is a single process, multi-threaded browser. For each tab opened in Firefox, at least one thread is spawned that manages this tab. A WebWorker is also handled in a separate thread. But since everything is happening in one process, it is easy to monitor Firefox's resource usage. Chrome, on the other hand, is a multi-process, multi-threaded browser. This means that Chrome's resource usage is spread across several processes. Chrome uses one master process and several child processes. Whenever Chrome is started, the master process will create a process for the visible window, a management process, and a separate process for each tab opened. This was not accounted for in the initial implementation.

Initially, only the resource usage for the Chrome master process was monitored. Because this process is not performing the actual tasks that are of interest, skewed results were obtained. By monitoring all processes spawned by the Chrome master process, one can make a better and more fair comparison between browsers.

Timing Inaccuracies

Many test scenarios depend on accurate measuring of a packet's send time and arrival time. Using an inaccurate measuring tool for this timing in low latency environments can give wrong impressions about the used connection and can lead to incorrect conclusions.

Initially, the JavaScript `Date.now()` function was used to measure the round-trip time of a packet. However, `Date.now()` only has millisecond-accuracy, and its returned values are not always the real amount of elapsed milliseconds. Using this function can result in graphs showing high fluctuations in round-trip times. [53]

The `performance.now()` function in JavaScript is a much more accurate measurement tool than the previous function, giving sub-millisecond accuracy. On Firefox, this function works without any further modifications to the settings of the browser. However, on Chrome for Windows, `performance.now()` rounds its returned value to the nearest millisecond. This would sometimes give very 'flat' results, e.g. many sequential round-trip times were exactly equal to 2 milliseconds, or sometimes even round-trip times of 0 milliseconds. To resolve this, Chrome needs to be started with the `--enable-high-resolution-time` CLI flag present in the start-up arguments.

Appendix B

Result Details

This chapter displays the raw results that were obtained during several test sessions on the different connections. Many of these tables also contain the results on which many of the figures in chapter 5 are based on.

B.1 Processing Overhead and Delay

	Q1-(1.5*IQR)	**Q1**	**Q2**	**Q3**	**Q3+(1.5*IQR)**
WebSocket	0.946	1.239	1.444	1.645	2.254
Secure WebSocket	0.923	1.500	1.715	1.920	2.550
WebRTC - reliable	1.522	2.208	2.436	2.667	3.354
WebRTC - unreliable	1.554	2.218	2.446	2.678	3.365
Unity - reliable	6.547	18.314	22.171	26.164	37.879
Unity - unreliable	6.772	18.510	22.379	26.340	37.870

Table B.1: Quantile round-trip values for Chrome.
Payload size of 100 bytes. Expressed in milliseconds.

	Q1-(1.5*IQR)	**Q1**	**Q2**	**Q3**	**Q3+(1.5*IQR)**
WebSocket	0.862	1.208	1.412	1.611	2.209
Secure WebSocket	0.900	1.524	1.732	1.947	2.581
WebRTC - reliable	1.359	2.042	2.270	2.499	3.182
WebRTC - unreliable	1.373	2.052	2.277	2.508	3.191
Unity - reliable	6.932	18.516	22.487	26.243	37.020
Unity - unreliable	7.420	18.753	22.629	26.311	37.455

Table B.2: Quantile round-trip values for Firefox.
Payload size of 100 bytes. Expressed in milliseconds.

	Q1-(1.5*IQR)	**Q1**	**Q2**	**Q3**	**Q3+(1.5*IQR)**
WebSocket	1.061	1.621	1.831	2.038	2.663
Secure WebSocket	1.359	1.941	2.171	2.403	3.095
WebRTC - reliable	2.111	2.854	3.091	3.350	4.094
WebRTC - unreliable	2.122	2.861	3.099	3.356	4.097
Unity - reliable	7.274	18.769	22.534	26.437	37.913
Unity - unreliable	7.891	19.198	22.895	26.742	36.949

Table B.3: Quantile round-trip values for Chrome.
Payload size of 1,000 bytes. Expressed in milliseconds.

	Q1-(1.5*IQR)	**Q1**	**Q2**	**Q3**	**Q3+(1.5*IQR)**
WebSocket	0.970	1.579	1.791	1.998	2.625
Secure WebSocket	1.285	1.971	2.199	2.432	3.122
WebRTC - reliable	1.771	2.507	2.750	3.003	3.746
WebRTC - unreliable	1.783	2.523	2.759	3.017	3.757
Unity - reliable	7.848	19.062	22.900	26.539	37.086
Unity - unreliable	8.375	19.353	23.104	26.673	37.032

Table B.4: Quantile round-trip values for Firefox.
Payload size of 1,000 bytes. Expressed in milliseconds.

	100 bytes	**1,000 bytes**	**10,000 bytes**
WebSocket	639.3	608.9	216.2
Secure WebSocket	609.1	563.8	207
WebRTC - reliable	440.8	366.3	88.9
WebRTC - unreliable	429.4	365.5	92.4

Table B.5: Packets per second comparison for Chrome.
Expressed in packets per second.

	100 bytes	**1,000 bytes**	**10,000 bytes**
WebSocket	644.3	600.4	217.6
Secure WebSocket	599.1	554.3	195.1
WebRTC - reliable	520.8	403.7	119.9
WebRTC - unreliable	516.3	404.5	120.2

Table B.6: Packets per second comparison for Firefox.
Expressed in packets per second.

	Google Chrome	**Mozilla Firefox**
WebSocket	2,081.25	3,139.062
Secure WebSocket	2,335.938	3,145.312
WebRTC - reliable	2,945.312	3,262.5
WebRTC - unreliable	2,968.75	3,354.688
Unity - reliable	8,195.312	4,475
Unity - unreliable	8,873.438	4,514.062

Table B.7: Avarage CPU time comparison.
Payload size of 100 bytes. Expressed in milliseconds.

	Google Chrome	**Mozilla Firefox**
WebSocket	2,010.938	3,137.500
Secure WebSocket	2,471.875	3,189.062
WebRTC - reliable	3,251.562	3,410.938
WebRTC - unreliable	2,962.500	3,304.688
Unity - reliable	9,660.938	4,743.750
Unity - unreliable	9,664.062	4,465.625

Table B.8: Avarage CPU time comparison.
Payload size of 1,000 bytes. Expressed in milliseconds.

B.2 Protocol Overhead and Bandwidth Usage

	Header	**ACK**	**STUN**
WebSockets 1pps	60(C) / 56(S)	60	NA
WebSockets 10pps	60(C) / 56(S)	60	NA
WebSockets 30pps	60(C) / 56(S)	NA	NA
WebSockets 60pps	60(C) / 56(S)	NA	NA
Secure WebSockets 1pps	89(C) / 85(S)	60	NA
Secure WebSockets 10pps	89(C) / 85(S)	60	NA
Secure WebSockets 30pps	89(C) / 85(S)	NA	NA
Secure WebSockets 60pps	89(C) / 85(S)	NA	NA
WebRTC - reliable 1pps	131 (147 incl. SACK)	135	1,048(C) / 514(F)
WebRTC - reliable 10pps	147 incl. SACK	NA	104.8(C) / 51.4(F)
WebRTC - reliable 30pps	147 incl. SACK	NA	34.9(C) / 17.1(F)
WebRTC - reliable 60pps	147 incl. SACK	NA	17.5(C) / 8.5(F)
WebRTC - unreliable 1pps	131 (147 incl. SACK)	135	1,048(C) / 514(F)
WebRTC - unreliable 10pps	147 incl. SACK	NA	104.8(C) / 51.4(F)
WebRTC - unreliable 30pps	147 incl. SACK	NA	34.9(C) / 17.1(F)
WebRTC - unreliable 60pps	147 incl. SACK	NA	17.5(C) / 8.5(F)
Unity - reliable 1pps	54	60	NA
Unity - reliable 10pps	54	60	NA
Unity - reliable 30pps	54	NA	NA
Unity - reliable 60pps	54	NA	NA
Unity - unreliable 1pps	42	NA	NA
Unity - unreliable 10pps	42	NA	NA
Unity - unreliable 30pps	42	NA	NA
Unity - unreliable 60pps	42	NA	NA

Table B.9: Total connection overhead analysis.
All values are expressed in bytes. The *Header* column shows the amount of bytes used by the header directly attached to the data. For WebSockets, the header size depends on whether the packet is sent from the client *(C)* or from the server *(S)*. For WebRTC, a SACK can be incorporated in the same packet as the data. The *ACK* column shows the amount of bytes that are registered for a separate (S)ACK packet that is sent for a request. The *STUN* column shows the amount of overhead the STUN mechanism produces per packet, per second and which browser is used: Chrome (C) or Firefox (F).

	100 bytes	**1,000 bytes**
WebSockets 1pps	88%	8.8%
WebSockets 10pps	88%	8.8%
WebSockets 30pps	58%	5.8%
WebSockets 60pps	58%	5.8%
Secure WebSockets 1pps	117%(C) / 133%(F)	11.7%(C) / 13.3%(F)
Secure WebSockets 10pps	117%(C) / 133%(F)	11.7%(C) / 13.3%(F)
Secure WebSockets 30pps	87%(C) / 103%(F)	8.7%(C) / 10.3%(F)
Secure WebSockets 60pps	87%(C) / 103%(F)	8.7%(C) / 10.3%(F)
WebRTC - reliable 1pps	730.5%(C) / 458.5%(F)	73%(C) / 45.9%(F)
WebRTC - reliable 10pps	199.4%(C) / 172.2%(F)	19.9%(C) / 17.2%(F)
WebRTC - reliable 30pps	164.4%(C) / 155.4%(F)	16.4%(C) / 15.5%(F)
WebRTC - reliable 60pps	155.8%(C) / 151.4%(F)	15.6%(C) / 15.1%(F)
WebRTC - ureliable 1pps	730.5%(C) / 458.5%(F)	73%(C) / 45.9%(F)
WebRTC - ureliable 10pps	199.4%(C) / 172.2%(F)	19.9%(C) / 17.2%(F)
WebRTC - ureliable 30pps	164.4%(C) / 155.4%(F)	16.4%(C) / 15.5%(F)
WebRTC - ureliable 60pps	155.8%(C) / 151.4%(F)	15.6%(C) / 15.1%(F)
Unity - reliable 1pps	84%	8.4%
Unity - reliable 10pps	84%	8.4%
Unity - reliable 30pps	54%	5.4%
Unity - reliable 60pps	54%	5.4%
Unity - unreliable 1pps	42%	4.2%
Unity - unreliable 10pps	42%	4.2%
Unity - unreliable 30pps	42%	4.2%
Unity - unreliable 60pps	4.2%	4.2%

Table B.10: Total connection overhead percentage with respect to the payload size.
Expressed in percentage to actual data. Some values depend on which browser is used: Chrome (C) or Firefox (F).

	1 pps	**10 pps**	**30 pps**	**60 pps**
WebSocket	376.0	3,760.0	9,574.8	1,9724.4
Secure WebSocket	434.0	4,340.0	11,338.2	23,343.6
WebRTC - reliable	1627.8	6,084.3	15,749.6	31,920.5
WebRTC - unreliable	1634.2	6,130.3	16,132.5	31,972.9
Unity - reliable	373.4	3,685.4	9,363.0	19,230.6
Unity - unreliable	284.0	2,840.0	8,611.2	17,750.0

Table B.11: Bandwidth usage comparison for Chrome.
Payload size of 100 bytes. Expressed in bytes per second.

	1 pps	10 pps	30 pps	60 pps
WebSocket	376.0	3,760.0	9,645.0	19,969.0
Secure WebSocket	466.0	4,660.0	12,337.2	25,610.0
WebRTC - reliable	1,091.8	5,481.1	15,596.3	31,733.8
WebRTC - unreliable	1,117.0	5,506.3	15,575.7	31,824.4
Unity - reliable	373.4	3685.4	9,368.4	19,274.6
Unity - unreliable	284.0	2,840.0	8,605.2	17,750.0

Table B.12: Bandwidth usage comparison for Firefox.
Payload size of 100 bytes. Expressed in bytes per second.

B.3 Effects of Packet Loss

	Q1-(1.5*IQR)	Q1	Q2	Q3	Q3+(1.5*IQR)
WebSocket	34.16	98.36	119.22	262.31	505.70
Secure WebSocket	23.03	98.49	119.43	256.83	494.24
WebRTC - reliable	0.00	101.68	126.04	338.21	692.47
WebRTC - unreliable	42.54	84.81	101.74	116.68	164.46
Unity - reliable	60.09	150.13	196.65	399.76	773.04
Unity - unreliable	50.79	98.74	116.16	130.88	178.96

Table B.13: Quantile round-trip values for Chrome in a busy network.
Payload size of 100 bytes. Expressed in milliseconds.

	Q1-(1.5*IQR)	Q1	Q2	Q3	Q3+(1.5*IQR)
WebSocket	44.75	99.32	121.53	316.67	640.59
Secure WebSocket	35.27	98.88	123.57	313.51	634.78
WebRTC - reliable	1.71	99.14	119.25	310.90	627.20
WebRTC - unreliable	42.28	86.33	102.26	116.19	160.89
Unity - reliable	59.99	145.28	190.05	376.56	718.30
Unity - unreliable	47.71	95.58	114.52	130.23	177.58

Table B.14: Quantile round-trip values for Firefox in a busy network.
Payload size of 100 bytes. Expressed in milliseconds.

Appendix C

Nederlandse Samenvatting

C.1 Introductie

De web browser wordt steeds vaker gebruikt om andere dingen op te doen dan waarvoor het oorspronkelijk ontworpen werd: informatie opzoeken. Nu wordt het onder andere gebruikt om ook videos op te bekijken, muziek te luisteren, en zelfs bedrijfs-gerelateerde taken. Het spelen van games kon dan ook niet uitblijven. Echter, multiplayer games blijven verrassend genoeg uit.

Met de HTML5 standaard worden een aantal interessante technologiën toegevoegd aan de browser zelf. Met WebGL kunnen 3D graphics gegenereerd en weergegeven worden. WebWorkers zorgen voor een primitieve vorm van multi-threading, en WebSockets laten toe om gemakkelijk data over het netwerk te sturen tussen client en server.

Voorheen was het al mogelijk om games te spelen in de web browser door middel van plug-ins. Deze breiden de functionaliteit uit van de browser met een afgezonderde omgeving waarin applicaties kunnen worden uitgevoerd. Één plug-in is specifiek gericht op het spelen van games in de web browser: Unity Webplayer.

WebRTC is ook een belangrijk, nieuw communicatie kanaal voor de web browser. WebRTC laat toe om peer-to-peer kanalen op te zetten met een andere browser. Het is een project opgezet door Google in samenwerking met Mozilla. Op dit moment van schrijven verkeert het nog in een experimentele fase, maar wordt al wel ondersteund door verscheidene browsers, waaronder Google Chrome, Mozilla Firefox en Opera.

C.1.1 Doel van deze Thesis

Omdat multiplayer games in de web browser nog uitblijven, is het interessant om te kijken welke mogelijkheden er zijn voor een game ontwikkelaar om game data te verzenden in een multiplayer game, en hoe deze mogelijkheden zich tegenover elkaar verhouden. Een game ontwikkelaar heeft niet de mogelijkheid om de technieken die beschikbaar zijn op een native platform één-op-één over te nemen naar een game in de browser. Zijn mogelijkheden zijn beperkt tot:

- WebSockets, en zijn beveiligde variant, Secure WebSockets,
- WebRTC data kanalen, en
- Een plug-in gebruiken die netwerktoegang biedt.

Als plug-in wordt gebruik gemaakt van de Unity Webplayer. Deze biedt de mogelijkheid om ook TCP en UDP kanalen op te zetten, en laat toe om deze ook goed te gebruiken. Andere plug-ins leggen strengere restricties op op het gebruiken van zulke kanalen.

Omdat de Unity Webplayer zowel TCP als UDP kanalen aanbiedt, worden twee gelijkaardige kanalen geconfigureerd voor WebRTC. Een betrouwbaar kanaal zorgt er voor dat data gegarandeerd zal aankomen bij de ontvanger, en ook in de juiste volgorde. Deze is gebasseerd op het TCP kanaal van Unity. Het onbetrouwebare kanaal lijkt dan weer op het UDP kanaal, en geeft geen garanties van aflevering of volgorde. WebSockets bieden geen mogelijkheid tot configuratie, en worden dus gebruikt zoals ze zijn.

Een oudere techniek, HTTP-gebasseerd data verzenden en ontvangen, welke zeer inefficiënt werd bevonden in eerdere onderzoeken ten opzichte van WebSockets, wordt niet opgenomen in dit onderzoek. WebSockets presteren op elk mogelijk vlak significant beter dan deze oude techniek.

De web browsers die gebruikt worden in deze thesis ondersteunen al deze mogelijkheden om data te versturen. Deze browsers zijn Mozilla Firefox en Google Chrome.

De doelstellingen van deze thesis worden dan als volgt geformuleerd:

- Een overzicht geven van de verschillende verbindsopties en een analyse van hun eigenschappen,
- Het evaluëren van deze eigenschappen in de context van multiplayer games,
- Nagaan of er een verschil is bij het gebruiken van verschillende web browsers, en
- Het aanbevelen van deze verbindingen tot bepaalde genres van multiplayer games.

C.1.2 Aanpak

De verschillende verbindingen worden eerst geanalyseerd op basis van hun onderliggende transport protocollen voor het netwerk. Dit geeft al een eerste idee van wat verwacht mag worden in sommige testen. Vervolgens worden ze onderworpen aan verschillende testen waarvan de resultaten gebruikt worden om een profiel van de verbinding op te stellen. Nadien worden de verbindingen besproken in hun bruikbaarheid voor verschillende genres van multiplayer games.

C.2 Eigenschappen van een Verbinding

Elke verbinding heeft een aantal eigenschappen die worden bepaald door de technologiën die ze gebruiken om hun data te kunnen transporteren over een netwerk.

C.2.1 WebSockets

WebSockets maken gebruik van TCP, en voegen daar nog een dun laagje aan toe. Een WebSocket verbinding zal garanderen, door TCP, dat elk verzonden bericht aan zal komen bij de correcte ontvanger, en dat de volgorde van de verzending wordt gerespecteerd. Het dun laagje van WebSockets zelf maakt het onderscheiden tussen de verschillende berichten gemakkelijker, wat niet standaard aanwezig is in TCP.

Secure WebSockets maken ook gebruik van TCP, maar beveiligen de data met nog een encryptielaag.

WebSockets kunnen enkel gebruikt worden om verbindingen tussen client en server op te stellen.

C.2.2 WebRTC

WebRTC maakt gebruik van een grotere set van protocollen. Onderaan wordt UDP gebruikt om de data te transporteren op het netwerk. Deze is in principe onbetrouwbaar: het zal niet garanderen dat de data effectief aankomt bij de ontvanger, en garandeerd ook geen afleverings volgorde. Daar boven wordt ook een beveiligingslaag gebruikt zoals bij Secure WebSockets. Ten slotte wordt SCTP gebruikt, welke toelaat om de tekortkomingen van UDP te configureren op maat. Zo kan een WebRTC kanaal geconfigureerd worden om wel of niet te garanderen dat de data zal aankomen, en wel of niet de data in de correcte volgorde afgeleverd zal worden.

WebRTC heeft de mogelijkheid om verbindingen tussen andere browsers, of andere WebRTC gebasseerde applicaties op te stellen. Dit is een unieke eigenschap van WebRTC ten opzichte van de andere verbindings mogelijkheden.

C.2.3 Unity Webplayer

De Unity Webplayer is een plug-in die toelaat om applicaties die ontwikkeld zijn in de Unity ontwikkelaars omgeving te draaien in de web browser. Ten opzichte van andere plug-ins, laat de Unity Webplayer toe om zowel reguliere TCP en UDP sockets te gebruiken. De Unity Webplayer kan echter niet zomaar met eender welk systeem verbinding maken. Het vraagt om een toelating van het andere systeem door middel van het opvragen van een bepaald bestand. Indien dit niet aanwezig is, dan kan de Unity Webplayer geen verbinding maken. Dit maakt het dus enkel geschikt voor client-server opstellingen.

C.3 Optimalisatie Technieken

Voor games zijn de factoren latency en gebruik van bandbreedte van belang om een optimale game ervaring te bieden. Zowel in de wereld van game development als in de wereld van web development zijn er verschillende strategiën die deze factoren optimaliseren.

Deze technieken kunnen gecombineerd worden om een goede en totale ervaring aan te bieden op de web browser als game platform.

C.3.1 Optimalisatie van Latency

Voor het web worden vaak de volgende technieken gebruikt om de latency te beperken:

- Caching: het tijdelijk opslaan van veel gevraagde web pagina's en andere bronnen op verschillende systemen in het netwerk of Internet.
- Content Distribution Networks: een doorgeslagen vorm van caching, waar op grote schaal vele web bronnen worden bewaard om een request snel te kunnen beantwoorden.
- Pre-fetching: trachten te voorspellen welke web pagina de gebruiker als volgende zal aanklikken en deze al op voorhand inladen.
- Nieuwe protocollen: Het huidige web protocol, HTTP 1.1, is al een tijdje in gebruik. Nieuwe protocollen worden ontworpen om de tekortkomingen van het huidige protocol te verbeteren. Hierbij hoort ook het verminderen van de latency.
- Compressie: sommige bestanden en bronnen kunnen sneller aangereikt worden als ze minder ruimte innamen. Compressie kan bestanden verkleinen op verschillende manieren om zo sneller een resultaat weer

te kunnen geven. Hierbij wordt voornamelijk gebruik gemaakt van de latency zoals die wordt ervaren door de gebruiker.

Ook in multiplayer games bestaan verschillende technieken om de latency in te perken:

- Optimale server plaatsing: meerdere servers worden verspreid over de hele wereld geplaatst zodat spelers met een server dichtbij kunnen verbinden waardoor het signaal niet te lang door het netwerk hoeft te reizen.
- Sharding: meerdere kopiëen van dezelfde virtuele wereld aanbieden en hosten op verschillende servers. Dit zorgt voor een spreiding van de load zodat een server niet overbelast raakt.
- Custom protocols: een game ontwikkelaar voor een native platform kan gebruik maken van alles wat dat platform te bieden heeft. In sommige gevallen kan hij er voor kiezen om een eigen protocol te ontwikkelen dat beter aansluit bij de game om zo de latency te verminderen.
- Netwerk architectuur: de netwerk architectuur heeft een grote invloed op welke weg de data doorheen het netwerk zal afleggen. In een client-server architectuur moet alle data eerst langs de server. Hier kan de server, of het netwerk bij de server, een bottleneck vormen. In peer-to-peer architecturen kan data direct tussen spelers gaan, wat de latency verbeterd. Er kunnen ook hybride architecturen opgesteld worden.
- Input response: sommige games kunnen wat meer tegen latency dan andere. De games die beter tegen latency kunnen maken veelal gebruik van feedback op de speler zijn input om hem gerust te stellen dat de input geregistreerd is. Dit verminderd echter alleen de ervaren latency, niet de latency die de update ondervindt.

C.3.2 Gebruik van Bandbreedte Inperken

De volgende technieken worden toegepast op het web om het bandbreedte gebruik te verminderen:

- Compressie: zoals eerder vermeld, kunnen sommige web resources kleiner gemaakt worden. Voor sommige bronnen kan dit resulteren in een enorme verbetering. Denk hierbij aan video en afbeeldingen. Maar ook tekst bestanden, zoals JavaScript code en HTML pagina's kunnen met de juiste technieken in bestandsgrootte enorm krimpen.
- Aggreatie van bronnen: als een web pagina veel verwijzingen heeft naar bvb. afbeeldingen en iconen, dan wordt elke afbeelding en icoontje

apart gedownload. Met aggregatie worden deze bronnen samengevoegd tot één groot bestand, waardoor minder bandbreedte verloren gaat aan onnodige overhead.

- Typed arrays: in JavaScript is het de gewoonte om data als tekst door te sturen over het netwerk. Dit kan zeer inefficiënt zijn. Typed arrays bieden een buffer aan die op byte-niveau aangepast kan worden. Dit kan zeer goed gebruikt worden voor numerische data.

- Caching: ook hier kan bandbreedte mee bespaard worden. Hoewel er even veel data wordt opgevraagd als de data niet gecached zou zijn, wordt er wel bandbreedte uitgespaard op het deel van het netwerk dat zich achter de cache service bevindt. De data moet niet meer van de originele bron komen.

- Geavanceerde protocols en APIs: dynamische web pagina's kunnen data opvragen en het resultaat weergeven zonder dat hiervoor de pagina voor ververst dient te worden. Nieuwere protocols, zoals WebSockets en WebRTC zorgen er voor dat dit veel efficiënter kan gebeuren dan voorheen.

De meeste games verbruiken al relatief weinig bandbreedte ten opzichte van ander data verkeer zoals audio en video conversaties, maar toch zijn er verscheidene technieken om dit nog verder te optimaliseren:

- Compressie: ook bij games kan compressie gebruikt worden. Hierbij kan voornamelijk numerische data met minder bytes worden weergegeven door rekening te houden met wat er in de vorige update verzonden is geweest. Hiermee kunnen delta-updates verstuurd worden die het verschil vormen tussen een eerdere referentie staat, en de huidige game staat.

- Input versus Game data: in sommige games is het beter om de input te versturen dan alle veranderingen die gebeuren in het spel. Voornamelijk bij strategie games, die veel eenheden kunnen bevatten, zal de input doorsturen in plaats van de acties per eenheid, een besparing in bandbreedte opleveren.

- Update aggregatie: om de overhead per update te verminderen kunnen updates tijdelijk opgehouden worden alvorens ze verzonden worden. Er wordt dan even gewacht op een eventuele volgende update die dan samen verzonden kunnen worden.

- Predicitie algoritmen: sommige games laten toe om de positie te voorspellen aan de hand van de data in de update. Zolang de positie van de voorspelling niet te veel afwijkt van de werkelijke positie dient er geen update uitgezonden te worden. Dit laat wel echter kleine inconsistenties tussen spelers toe in het spel, maar spaart wel updates uit.

- Update filtering: een speler heeft niet altijd even veel gedetailleerde informatie nodig van dingen die zich achter hem, of in de verte afspelen. Dan kan er bandbreedte bespaard worden door zo'n updates gewoonweg niet te verzenden naar de spelers die ze niet nodig hebben.

C.4 Setup en Testen

Elke verbinding zal onderworpen worden aan een reeks van testen die elk een bepaald type van verbruik zullen meten. De twee voornaamste parameters waarmee gespeeld wordt zijn de grootte van de data die verzonden wordt, en de snelheid waarmee de pakketten verzonden worden. Door deze twee parameters te laten variëren kan een relatieve vergelijking opgesteld worden van de verschillende connecties. Deze testen worden uitgevoerd in een eenvoudige client-server netwerk opstelling.

Deze testen zijn voornamelijk gebasseerd op de round-trip tijden van de pakketten. Een pakket wordt gegenereerd door de client en verzonden naar de server. De server zal ditzelfde pakket gewoonweg terug naar de zender sturen. De client zal de tijdsstippen bijhouden waarop het pakket werd verzonden en ontvangen.

C.4.1 CPU Gebruik

De eerste test legt zich toe het CPU verbruik van een bepaalde verbinding te meten. Door pakketten aan een constante snelheid te genereren voor een bepaalde zijn met een bepaalde grootte, zal elke verbinding even veel pakketten uitsturen. Dit laat gemakkelijk toe om het verschil in CPU gebruik te vergelijken.

In deze test blijken de WebSocket en WebRTC kanalen goed te scoren. Zij vertonen een laag verbruik van de CPU. WebSockets hebben hier nog een klein voordeel ten opzichte van WebRTC. Ook de Secure WebSocket kanalen presteren beter dan de WebRTC kanalen, ondanks ze beide van encryptie zijn voorzien. De Unity Webplayer scoort hier het slechtste omwille van zijn update loop die continu zijn werk doet, ondanks er geen pakketten zijn om te verwerken. Uit deze tests blijkt ook dat de Unity Webplayer dankzij deze update loop ook extra latency en latency jitter toevoegd.

C.4.2 Bandbreedte Gebruik

In de voorgaande reeks van testen werden door elke verbinding even veel pakketten verstuurd. Dit laat ook toe om de verzonden pakketten te analyseren in hun overhead, zowel directe overhead die direct gekoppeld is aan de data die verzonden wordt, maar ook operationele overhead die wordt geproduceerd door de verbinding. Dit kunnen bijvoorbeeld bevestigings pakketten zijn om aan te geven dat een bepaald pakket goed is aangekomen.

Waar de Unity Webplayer slecht scoorde in de vorige test, scoort hij hier het beste. Door zijn minimalistische kanalen wordt er weinig overhead geproduceerd. WebSockets presteren hier goed tweede door middel van het dunne laagje dat het WebSocket protocol toevoegd aan de onderliggende TCP connectie. WebRTC daarentegen, creëert een enorme hoeveelhied aan overhead. Dit is deels te danken aan de vele lagen die gebruikt worden om data te transporteren, maar ook aan de implementatie van WebRTC. De grootste overhead is te vinden bij Google Chrome. Door een afwijkende implementatie genereerd deze overdadig veel extra traffiek die snel de nuttige hoeveelheid data overschaduwt.

C.4.3 Sub-optimale Netwerkomstandigheden

De vorige testen werden uitgevoerd in optimale netwerkomstandigheden waarbij er geen artificiële packet loss en latency geïntroduceerd werden. Hier wordt wél gebruik gemaakt van packet loss, latency en latency jitter om het gedrag waar te nemen van de verschillende verbindingen op deze parameters.

Hier vallen voornamelijk de betrouwbare connecties op. Deze zorgen ervoor dat de data gegarandeerd wordt afgeleverd aan de ontvanger en in de juiste volgorde. Deze verbindingen vertonen grote pieken wanneer ze worden geconfronteerd met data die verloren is gegaan. De onbetrouwbare kanalen zijn geconfigureerd om geen garanties te bieden. Deze vertonen dan ook een relatief vlakke lijn wanneer de round-trip tijden worden afgedrukt. Hier is opvallend dat de eerder waargenomen latency en jitter van de Unity Webplayer zijn update loop, op gaat in de latency en jitter van het netwerk. Hierdoor verschillen het onbetrouwebare WebRTC kanaal en Unity Webplayer UDP kanaal nauwelijks van elkaar.

C.5 Conclusie

Game ontwikkelaars voor de web browser hebben nu de mogelijkheid om ook een multiplayer aspect toe te voegen aan hun game. Voor elk genre van games is er een oplossing beschikbaar die multiplayer mogelijk maken. Echter, ten opzichte van een ontwikkelaar voor een native platform dient hij nog wel een belangrijke trade-off the maken. Deze trade-off bestaat vooral in het kiezen tussen optimaal gebruik van de bandbreedte, of optimale latency waarden. Er is op dit moment van schrijven geen ideale of optimale oplossing beschikbaar.

Games die gebruik willen maken van een peer-to-peer architectuur kunnen niet onder WebRTC uit. Maar voor client-server architecturen zijn er veel meer opties:

- Voor games die een hoge update rate hebben zoals shooter games en racing games, zijn vooral de onbetrouwbare kanalen geschikt. De onbetrouwebare configuratie van WebRTC en de UDP connectie van de Unity Webplayer bieden deze oplossing. Het WebRTC kanaal biedt de beste latency waarden, terwijl de Unity UDP connectie het efficiënste gebruik maakt van de bandbreedte.

- Action en role-playing games hebben een iets lagere update rate, en hebben eigenlijk de optie om uit alle mogelijke verbindingen te kiezen. Maar hier ook dezelfde trade-off. WebSockets hebben een wat hogere band-breedte nodig dan de Unity TCP connectie. Maar deze laatste verhoogt de latency. Het betrouwbare WebRTC kanaal kan hier niet echt dienst doen omdat de WebSocket verbinding dit beter aanpakt.

- Tragere games zoals strategie en puzzel games worden meer aangedreven door de input van de spelers. Deze zal dus relatief weinig updates genereren, en de games zijn over het algemeen redelijk resistent tegen latency. Hierdoor kan best een betrouwbare verbindingsoptie gekozen worden. Ondanks dat de Unity Webplayer hier een betere bandbreedte efficiëncy zou halen, is het echter aangeraden om hier te opteren voor de WebSocket verbinding. Deze zal minder energie verbruiken, terwijl het bandbreedte gebruik slechts marginaal toeneemt.

Bibliography

[1] AjaxPatterns. Http streaming. http://ajaxpatterns.org/HTTP_Streaming#Solution, Februari 2014.

[2] Alessandro Alinone. Optimizing Multiplayer 3D Game Synchronization Over the Web. http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/optimizing-multiplayer-3d-game-synchronization-over-the-web-r3446, December 2013.

[3] Amir Almasi and Yohanes Kuma. Evaluation of WebSocket Communication in Enterprise Architecture. 2013.

[4] Automattic. socket.io. https://github.com/Automattic/socket.io, 2014.

[5] Steve Benford and Lennart Fahln. A spatial model of interaction in large virtual environments. In Giorgio de Michelis, Carla Simone, and Kjeld Schmidt, editors, *Proceedings of the Third European Conference on Computer-Supported Cooperative Work 1317 September 1993, Milan, Italy ECSCW 93*, pages 109–124. Springer Netherlands, 1993.

[6] Yahn W. Bernier. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization, 2001.

[7] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 389–400, New York, NY, USA, 2008. ACM.

[8] Bluestop. SctpDrv: an SCTP driver for Microsoft Windows. http://www.bluestop.org/SctpDrv/, May 2012.

[9] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA, 2006. ACM.

[10] Wu chang Feng and Wu chi Feng. On the Geographic Distribution of On-line Game Servers and Players. May 2003.

[11] Clay.io. HTML5 Game Engines. `http://html5gameengine.com/`.

[12] Mozilla Corporation. Lossy Compressed Image Formats Study. `http://people.mozilla.org/~josh/lossy_compressed_image_study_july_2014/`, July 2014.

[13] Alberto Dainotti, Salvatore Loreto, Antonio Pescap, and Giorgio Ventre. Sctp performance evaluation over heterogeneous networks. *Concurrency and Computation: Practice and Experience*, 19(8):1207–1218, 2007.

[14] Matthias Dick, Oliver Wellnitz, and Lars Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '05, pages 1–7, New York, NY, USA, 2005. ACM.

[15] Y. Elkhatib, G. Tyson, and M. Welzl. Can spdy really make the web faster? In *Networking Conference, 2014 IFIP*, pages 1–9, June 2014.

[16] Wikipedia The Free Encyclopedia. Bufferbloat. `https://en.wikipedia.org/wiki/Bufferbloat`, May 2014.

[17] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and K. K. Ramakrishnan. Towards a spdy'ier mobile web? In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 303–314, New York, NY, USA, 2013. ACM.

[18] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, pages 178–187, New York, NY, USA, 1999. ACM.

[19] I. Fette and A. Melnikov. The WebSocket Protocol. `http://tools.ietf.org/html/rfc6455#section-10.3`, December 2011.

[20] I. Fette and A. Melnikov. The WebSocket Protocol. `http://tools.ietf.org/html/rfc6455#section-5.5.2`, December 2011.

[21] I. Fette and A. Melnikov. The WebSocket Protocol. http://tools.ietf.org/html/rfc6455#section-5.1, December 2011.

[22] The Apache Software Foundation. Apache core features - keepalive-timeout directive. https://httpd.apache.org/docs/2.2/mod/core.html#keepalivetimeout.

[23] Inc. Google. Optimizing encoding and transfer size of text-based assets. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer#minification-preprocessing--context-specific-optimiz
April 2014.

[24] Inc. Google. WebP. https://developers.google.com/speed/webp/?csw=1, March 2014.

[25] Inc. Google. WebP Compression Study. https://developers.google.com/speed/webp/docs/webp_study, March 2014.

[26] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: A peer-to-peer approach to scalable multi-player online games. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, pages 116–120, New York, NY, USA, 2004. ACM.

[27] Google Inc. SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[28] Jared Jardine and Daniel Zappala. A Hybrid Architecture for Massively Multiplayer Online Games. http://netgames2008.cs.wpi.edu/slides/jardine.pdf, 2008.

[29] Alan B. Johnston and Daniel C. Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web.* Digital Codex LLC, USA, 2012.

[30] Alan K. node-webrtc. https://github.com/js-platform/node-webrtc, March 2014.

[31] B. Knutsson, H. Lu, Wei Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, pages –107, March 2004.

[32] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and RandalM. Henne. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1):140–181, 2009.

[33] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the use and performance of content distribution networks. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 169–182, New York, NY, USA, 2001. ACM.

[34] Markku Laine and Kalle Säilä. Performance evaluation of xmpp on the web. Technical report, Aalto University Technical Report, 2012.

[35] Eric Li. Optimizing WebSockets Bandwidth. `http://buildnewgames.com/optimizing-websockets-bandwidth/`, September 2012.

[36] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. XEP-0166: Jingle. `http://xmpp.org/extensions/xep-0166.html`, December 2009.

[37] M.R. Macedomia, M.J. Zyda, D.R. Pratt, D.P. Brutzman, and P. Barham. Exploiting reality with multicast groups: a network architecture for large-scale virtual environments. In *Virtual Reality Annual International Symposium, 1995. Proceedings.*, pages 2–10, Mar 1995.

[38] Michael R Macedonia. *A network software architecture for large scale virtual environments*. PhD thesis, Monterey, California. Naval Postgraduate School, 1995.

[39] Robin Marx, Peter Quax, and Wim Lamotte. Dynamic Bandwidth Scalability in Large Scale Networked Virtual Environments, 2011.

[40] Michael F. Nowlan and Bryan Ford. Unordered Delivery in TLS-Encrypted TCP Connections. 2009.

[41] Nuwen.net. Introduction to PNG. `http://nuwen.net/png.html`.

[42] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[43] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '02, pages 23–29, New York, NY, USA, 2002. ACM.

[44] Ian Paterson, Dave Smith, Peter Saint-Andre, Jack Moffitt, Lance Stout, and Winfried Tilanus. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). `http://www.xmpp.org/extensions/xep-0124.html#technique`, April 2014.

[45] M. Perumal, D. Wing, R. Ravindranath, T. Reddy, and M. Thomson. STUN Usage for Consent Freshness. `http://tools.ietf.org/html/`

draft-ietf-rtcweb-stun-consent-freshness-05#section-4.1, October 2009.

[46] The WebM Project. WebM: an open web media project. http://www.webmproject.org/, 2014.

[47] D.G. Puranik, D.C. Feiock, and J.H. Hill. Real-time monitoring using ajax and websockets. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 110–118, April 2013.

[48] James Purbrick and Chris Greenhalgh. Extending locales: Awareness management in massive-3. *2014 IEEE Virtual Reality (VR)*, 0:287, 2000.

[49] Peter Quax, Bart Cornelissen, Jeroen Dierckx, Gert Vansichem, and Wim Lamotte. Alvic-ng: state management and immersive communication for massively multiplayer online games and communities. *Multimedia Tools and Applications*, 45(1-3):109–131, 2009.

[50] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, pages 152–156, New York, NY, USA, 2004. ACM.

[51] Shruto M. Rakhunde. Real Time Data Communication over Full Duplex Network Using Websocket. *IOSR Journal of Computer Science*, 2014.

[52] Giuseppe Reina, Ernst Biersack, and Christophe Diot. Quiver: A middleware for distributed gaming. In *Proceedings of the 22Nd International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '12, pages 27–32, New York, NY, USA, 2012. ACM.

[53] John Resig. Accuracy of JavaScript Time. http://ejohn.org/blog/accuracy-of-javascript-time/, November 2008.

[54] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. https://tools.ietf.org/html/rfc5245#page-65, April 2010.

[55] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. http://tools.ietf.org/html/rfc3261, June 2002.

[56] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the h.264/avc standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(9):1103–1120, Sept 2007.

[57] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in warcraft iii. In *Proceedings of the 2Nd Workshop on Network and System Support for Games*, NetGames '03, pages 3–14, New York, NY, USA, 2003. ACM.

[58] Einar Otto Stangvik. ws. `https://github.com/einaros/ws`, 2014.

[59] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. `http://www.ietf.org/rfc/rfc3758.txt`, May 2004.

[60] Timo Stripf and Tobias Oetzel. Open Game Protocol. `http://www.open-game-protocol.org/index.php`.

[61] Wikipedia the Free Encyclopedia. Lockstep protocol. `http://en.wikipedia.org/wiki/Lockstep_protocol`, April 2011.

[62] Wikipedia the Free Encyclopedia. Comet. `https://en.wikipedia.org/wiki/Comet_%28programming%29`, August 2014.

[63] Wikipedia the Free Encyclopedia. HTML5 Audio. `http://en.wikipedia.org/wiki/HTML5_Audio`, July 2014.

[64] Wikipedia the Free Encyclopedia. Sliding window protocol. `http://en.wikipedia.org/wiki/Sliding_window_protocol`, May 2014.

[65] Daniel J Van Hook, James O Calvin, and Duncan C Miller. A protocol independent compression algorithm (pica). *Advanced Distributed Simulation Project Memorandum 20PM–ADS–005, MIT Lincoln Laboratories, Lexington, Massachusetts*, 1994.

[66] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is spdy. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 387–399. USENIX Association, 2014.

[67] Second Life Wiki. Viewer Architecture. `http://wiki.secondlife.com/wiki/Viewer_Architecture`, March 2011.

[68] Worlize. WebSocket Client Benchmark. `http://worlize.github.io/WebSocket-Node/benchmarks/`.

[69] Worlize. WebSocket-Node. `https://github.com/Worlize/WebSocket-Node`, 2013.

[70] Chen-Chi Wu, Kuan-Ta Chen, Chih-Ming Chen, Polly Huang, and Chin-Laung Lei. On the challenge and design of transport protocols for mmorpgs. *Multimedia Tools and Applications*, 45(1-3):7–32, 2009.