WEB 3D GAMING OVER HTML5 AND WEB-BASED COMMUNICATION

by

ZACHARO-EVANGELIA ANDRIOTI

Master in Informatics and Multimedia, Department of Informatics Engineering, School of Applied Technology, Technological Educational Institute of Crete, 2015

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF INFORMATICS ENGINEERING
SCHOOL OF APPLIED TECHNOLOGY
TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

2015

Approved by:

Major Professor
Dr. Spyros Panagiotakis
Copyright

ZACHARO-EVANGELIA ANDRIOTI

2015
ABSTRACT

The objective of this thesis is to conduct a study on state-of-the-art technologies that can support networking for Web 3D gaming, evaluate them and finally introduce a new perspective in web 3D gaming architectures that is based on the Web Real-Time Communication (WebRTC) combined with the X3DOM technology. Using peer-to-peer connections between browsers for web 3D gaming, we bridge two major fields: Multimedia and Graphics and introduce the concept of WebRTC over X3DOM.

In particular, we implemented a collaborative web game that is presented in a 3D environment where the users are provided with a rich 3D graphics environment using web browsers without any need for third party software or plugin installation. The basic characteristics of our implementation include the following provisions: video and audio call; exchange of text messages in real-time; selection from a menu and addition in the virtual world of 3D objects (houses in our application); bidirectional interaction with the virtual world, since each user can change the position of every object (houses) in it.

In addition, we conducted experiments on data transfer over WebSockets and WebRTC and developed a benchmark for each technology respectively to assure its applicability on network gaming.

Overall, this thesis makes us acquainted with the fields of web 3D gaming and web-based communications.

Key words: HTML5, WebRTC, WebSockets, JavaScript, X3DOM, node.js, socket.io, web 3D gaming, web-based communication
# Table of Contents

List of Figures .................................................................................................................. 5
List of Tables ...................................................................................................................... 7
Abbreviation Index .......................................................................................................... 8
Acknowledgments ............................................................................................................ 10
Introduction ..................................................................................................................... 11

Chapter 1: Web Gaming Review ..................................................................................... 14
  1.1 History ................................................................................................................... 14
  1.2 Basic Architecture ............................................................................................... 15
  1.3 Distributed Hash Table ....................................................................................... 19
  1.4 Pastry Architecture ............................................................................................. 20
  1.5 Related Work ........................................................................................................ 20

Chapter 2: Background Technologies & Tools ............................................................... 22
  2.1 WebSockets ........................................................................................................ 22
  2.2 WebRTC ............................................................................................................... 27
  2.3 X3DOM ................................................................................................................ 47
  2.4 HTML5 ................................................................................................................ 49
  2.5 JavaScript - CSS .................................................................................................. 49
  2.6 Tools .................................................................................................................... 50
    2.6.1 Node.js ......................................................................................................... 50
    2.6.2 Socket.io ...................................................................................................... 51

Chapter 3: Benchmarking WebSockets & WebRTC concerning data transfer .......... 53
  3.1 WebSockets Benchmark Code Structure ............................................................... 55
  3.2 WebRTC Benchmark Code Structure ................................................................ 57
  3.3 WebSockets Measurements ............................................................................... 58
  3.4 WebRTC Measurements ..................................................................................... 61
  3.5 WebRTC vs. WebSockets .................................................................................. 63
  3.6 Conclusions ......................................................................................................... 64

Chapter 4: Implementation ............................................................................................ 65
  4.1 Running the application ....................................................................................... 77

Chapter 5: Conclusions & Future Work ...................................................................... 82
References ....................................................................................................................... 85
LIST OF FIGURES

Figure 1.1: Client-Server Architecture.................................................................16
Figure 1.2: Peer-to-Peer Architecture.................................................................17
Figure 1.3: MultiServer Architecture.................................................................19
Figure 2.1.1: WebSocket Connection.................................................................23
Figure 2.2.1: WebRTC Browser Model..............................................................28
Figure 2.2.2: Generic WebRTC architecture......................................................29
Figure 2.2.3: JSEP architecture........................................................................30
Figure 2.2.4: MediaStream architecture.............................................................32
Figure 2.2.5: Basic Diagram...............................................................................37
Figure 2.2.6: WebRTC Protocol Layers..............................................................38
Figure 2.2.7: STUN Network Architecture.........................................................40
Figure 2.2.8: TURN Network Architecture.......................................................41
Figure 2.2.9: Peer-to-peer connection...............................................................42
Figure 2.2.10: Mesh topology............................................................................42
Figure 2.2.11: Star topology...............................................................................42
Figure 2.2.12: MCU topology............................................................................43
Figure 3.1.1: WebSocket Benchmark Code Structure.........................................56
Figure 3.2.2: WebRTC Benchmark Code Structure...........................................57
Figure 3.3.1: Compare average time in relation to Data Size between Client-Server &
  Server-Client.................................................................................................58
Figure 3.3.2: Client-to-Client data transfer versus e2e time..............................59
Figure 3.3.3: Client-server, server-client & server processing............................60
Figure 3.4.1: E2E transfer time in WebRTC.......................................................61
Figure 3.4.2: Calculations for the Negotiation time and Data size Transfer in
  WebRTC........................................................................................................62
Figure 3.4.3: Data size vs time transfer: WebRTC vs WebSockets....................63
Figure 4.1: Pictorial view of Implementation.....................................................66
Figure 4.2: Implementation Steps.....................................................................67
Figure 4.3: EViE-m 3D virtual world.................................................................69
Figure 4.4: Video & Audio Call Application......................................................70
Figure 4.5: Chat application exchanging text messages.....................................72
Figure 4.6: 3D houses located by the user in the virtual world.........................75
Figure 4.7: Implementation as shown with the use of Google Chrome Browser ...........76
Figure 4.1.1: Activating Apache Server .......................................................77
Figure 4.1.2: Run node.js server .................................................................77
Figure 4.1.3: Open client.xhtml page ..........................................................77
Figure 4.1.4: Use of video chat .................................................................79
Figure 4.1.5: Selecting HouseA ...............................................................80
Figure 4.1.6: All three Houses placed in the virtual world and showed from
   a different angle .................................................................81
Figure 4.1.7: Exchanging text messages ..................................................81
LIST OF TABLES

Table 1.1: Basic Architectures for MMOGs and characteristics..........................16
Table 1.2: Types of P2P Networks ......................................................................18
Table 2.1.1: Code Sample from RFC6455- Handshake client side......................22
Table 2.1.2: Code Sample from RFC6455- Handshake server side....................23
Table 2.1.3: Opcode initial values.....................................................................24
Table 2.1.4: WebSocket Event Handlers.............................................................24
Table 2.1.5: Sample code of WebSocket Event Handlers.....................................25
Table 2.2.1: Code Sample for Signaling..............................................................31
Table 2.2.2: Sample code for video stream.......................................................33
Table 2.2.3: Sample code for RTCPeerConnection...........................................34
Table 2.2.4: RTCDataChannel API Attributes...................................................36
Table 2.2.5: RTCDataChannel API Event Handlers.............................................36
Table 2.2.6: RTCDataChannel API Methods.....................................................36
Table 2.2.7: Comparing TCP, UDP, SCTP protocols.........................................36
Table 2.3.1: Code Sample of an X3D Scene and the creation of a red box...........48
Table 2.6.1.1: Hello World example.................................................................50
Table 2.6.2.2: Using Node.js and socket.io – Server Side.................................52
Table 2.6.2.3: Using Node.js and socket.io – Client Side....................................52
Table 4.1: CODE – server.js.............................................................................68
Table 4.2: X3DOM libraries..............................................................................68
Table 4.3: Socket.io library...............................................................................69
Table 4.4: Start Video() function.....................................................................71
Table 4.5: Commands used in DataChannel library..........................................72
Table 4.6: Defining id and translation for each house .......................................73
Table 4.7: Code for: Channel opened and HouseA information..........................74
Table 4.8: Code for: When a house is located in the X3D scene and it is moveable...74
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BMOG</td>
<td>Browser Multiplayer Online Game</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Styling Sheet</td>
</tr>
<tr>
<td>DHT</td>
<td>Distributed Hash Table</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DTLS</td>
<td>Datagram Transport Layer Security</td>
</tr>
<tr>
<td>EViE-m</td>
<td>Educational Virtual Environment – mathematics</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTML5</td>
<td>HyperText Markup Language 5</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>ICE</td>
<td>Interactive Connectivity Establishment</td>
</tr>
<tr>
<td>iSAC</td>
<td>internet Speech Audio Codec</td>
</tr>
<tr>
<td>iLBC</td>
<td>internet Low Bitrate Codec</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>JS</td>
<td>JavaScript</td>
</tr>
<tr>
<td>JSEP</td>
<td>JavaScript Session Establishment Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MCU</td>
<td>Multipoint Control Unit</td>
</tr>
<tr>
<td>MOG</td>
<td>Multiplayer Online Game</td>
</tr>
<tr>
<td>MMOG</td>
<td>Massively Multiplayer Online Game</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>Node.js</td>
<td>Free open-source platform built on Chrome's JavaScript runtime</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>RTC</td>
<td>Real-Time Communication</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-Time Protocol</td>
</tr>
<tr>
<td>SCTP</td>
<td>Stream Control Transmission Protocol</td>
</tr>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>Socket.io</td>
<td>Open-source real-time engine used to provide real-time bidirectional event-based communication</td>
</tr>
<tr>
<td>SRTP</td>
<td>Secure Real Time Protocol</td>
</tr>
</tbody>
</table>
STUN: Session Traversal Utilities for NAT
TURN: Traversal Using Relays around NAT
TCP: Transmission Control Protocol
UDP: User Datagram Protocol
VoD: Video on Demand
VoIP: Voice over IP
VP8: Video compression format
W3C: World Wide Web Consortium
WebGL: Web Graphics Library
WebRTC: Web Real-Time Communication
WebSocket: Protocol that supports bi-directional communication over a single TCP connection
X3D: Extensible 3D
X3DOM: open-source framework for 3D graphics on the Web
XMPP: Extensible Messaging and Presence Protocol
ACKNOWLEDGEMENTS

I would like to thank my supervisor Assistant Professor Dr. Spyros Panagiotakis and the Head of Multimedia Content Laboratory Associate Professor Dr. Athanasios Malamos for their trust and faith in me for this thesis and giving me the opportunity to enhance my knowledge with state-of-the-art technologies in the fields of web gaming, 3D graphics and web-based communications.

Furthermore, I want to thank my friends and fellow students Kostas Kapetanakis and Andreas Stamoulias for their help and guidance. I would also like to thank Helen Vonorta and Malvina Steiakaki for their help. Also, I want to thank all the people in Multimedia Content Laboratory for their friendship and support!!

Last but not least, I would like to thank my family for everything they have done for me!! Thank you for your love and support and for being by my side all this time!! :)

March 2015,
Zacharo-Evangelia Andrioti
INTRODUCTION

Nowadays, web 3D gaming is gaining a big percentage in the field of online gaming. A web game consists of several parts many of which need to be updated several times during the game or have to be exchanged between the players. For instance, a game consists of graphics, images, CSS, scripts, databases, etc. As a result, the communication part is extremely important in order to handle the large amount of data. In recent years, the most well-known communication architectures for web games are client-server and peer-to-peer connections. Client-Server connection is a well-known architecture where the client asks information from the server. It provides simplicity, reliability and no scalability. On the other hand, in Peer-to-Peer (P2P) architecture each peer acts as a client and as a server. It has good performance, low cost, supports anonymity, and enables minimum bandwidth.

Lately, HTML5 has introduced two solutions for establishing the communication part of a web game. The first solution is one of the new characteristics that HTML5 provides; which is the WebSockets. WebSockets provide developers to create real-time applications by using socket connections between the browser and the server. In simple words, client and server can exchange data because it offers a persistent connection. The second solution is the WebRTC Data Channel API which supports browser-to-browser communication. DataChannel API supports peer-to-peer connection which enables RTCPeerConnection API in order to settle the connection. As a result, there is less latency and no server to intervene. In addition, DataChannel API uses Stream Control Transmission Protocol (SCTP) which is a transport protocol and supports out-of-order delivery and retransmits configuration. [1]

Furthermore, 3D graphics can be supported in a web game. The technologies that can be used in a web game are WebGL and X3D. WebGL is a free cross-platform API which is used to create 3D graphics in a web browser. It is based on OpenGL ES 2.0. Additionally, it runs on HTML5 canvas element which as a result it also provides full integration with Document Object Model (DOM) interface. WebGL is supported by most browsers such as Google Chrome, Mozilla Firefox, Opera, and Apple Safari. [2]

On the other hand, X3D is a royalty-free and open standard file format and run-time architecture that uses XML in order to represent and communicate 3D scenes and objects. X3D is the continuance of the Virtual Reality Modeling Language (VRML) and is confirmed by the International Organization for Standardization (ISO) standard. X3D
supports 2D and 3D graphics, animation, user interaction, physical simulation and real-time communication and etc. [3]

In the present thesis work, instead of using WebGL and X3D in order to use 3D graphics, we use another technology which combines JavaScript and WebGL. This is X3DOM which is a JavaScript library and combines WebGL and X3D technologies. As a result, our goal is to combine X3DOM with WebRTC DataChannel API allowing the development of virtual worlds where it can support interaction and co-operation between them.

Web 3D gaming that supports real-time communication between users is part of a new era of online gaming. In the last five years, there are samples of web games development based on real-time communication and 3D graphics. Specifically, it is observed that WebRTC technology is used in combination with JavaScript, WebGL and HTML5. For instance, Cube Slam [4], [5] is a web game created by Google Chrome Experiment developed with WebRTC technology. Cube Slam uses all the WebRTC APIs which are: the MediaStream, the RTCPeerConnection and RTCDataChannel. Furthermore, WebGL and CSS 3D technologies have been used and Web Audio API is used for the sound. Cube Slam is the closest concept in comparison with our web game application of our thesis. The main difference is that instead of using WebGL and CSS3 technologies for the 3D graphics, our concept is to use X3DOM technology.

Another web 3D game is the BananaBread. BananaBread is a 3D first-person shooter web game which was developed in 2012 [6]. It is a C++ 3D game engine and it is run on the web using JavaScript, HTML5, and WebGL. Moreover, the latest demo supports WebRTC technology for multiple players. BananaBread web game can run on web browsers without the use of plugins.

Apart from web games, another recent development is PeerCDN which is a Peer-to-Peer (P2P) Content Delivery Network for the web browser [7]. PeerCDN is written in JavaScript and provides the opportunity to transfer files quickly, reliably, improving response time, it is secure and minimizing the bandwidth and server costs. In addition, it enables WebRTC DataChannel API so as to install peer-to-peer connection.

In this thesis, we will focus on the communication part of a web game. Our contribution is to present a new perspective in web 3D gaming architecture that is based on real-time communication over X3DOM technology. The above development is presented in a 3D demo collaborative environment online game.
The goal of this thesis is to conduct a study on state-of-the-art technologies that can support Web 3D games networking. Our scope is to study, test and finally propose the most appropriate architecture for the support of browser-to-browser communication required in case of web gaming.

In our implementation, a virtual world is loaded in our web page and a user has the opportunity to select and insert 3d objects in the virtual world. When user1 is connected with user2, both users have the opportunity to transfer via RTCDataChannel API 3D objects while simultaneously interaction is allowed between them so as any change that occurs by each user in their virtual world to be transferred to the virtual world of the other user.

In order to transfer 3d objects via RTCDataChannel API, we created a new channel and using inline method we were able to import our X3D models into the X3DOM context. In order to include X3D files; X3DOM provides an Inline node. By using inline in the x3d context we are able to load external x3d files [8]. Once we load the X3D object in the virtual world we have the opportunity to change position of the X3D object using an UI event. [9]

Furthermore, for our implementation purposes, we used the virtual world of an edutainment game platform, named EViE-m [12], so as to be used as a demo web3D collaborative environment.

Additionally, we conducted a benchmark between WebSockets [10] and DataChannel API of WebRTC [11] so as to decide if peer-to-peer communication (WebRTC) is more effective than client-server communication (websockets) for data transfer between web game peers. In particular, we attempt to measure the end-to-end latency involved in the exchange of messages of various data sizes between two peers using the above technologies respectively.

The thesis is organized as follows: In Chapter 1 a review is presented for Web Gaming. In Chapter 2 technologies background is provided for WebSockets, WebRTC, and X3DOM. Additionally, we introduce the tools that we used for the implementation which are JavaScript, HTML5, CSS, node.js and socket.io. Finally, we present related work concerning the above technologies. Furthermore, we present a survey on Peer-to-Peer (P2P) architectures for multiplayer online games. In Chapter 3, a benchmark is presented along with our results and conclusions. In Chapter 4 our implementation is presented based on WebRTC over X3DOM and a presentation running our application. Finally, in Chapter 5 we provide our conclusions and future work.
CHAPTER 1: WEB GAMING REVIEW

In this chapter, we will present a survey on web gaming. At first, we will present the history of online gaming and where we stand today. In addition, we will mainly focus on the two different architectures that Massively Multiplayer Online Games (MMOGs) support which are Client-Server and Peer-to-Peer architecture. We will refer to their structure, pros and cons and their differences. Finally, we will reveal possible prospects of developing a web 3D game based on one of the architectures above and combined real-time communication with 3D graphics.

1.1 History

Online games were introduced to the public as a succession of the video games that were initially developed for computers. An online game is a video game which can be played over the internet. Online gaming provided its own influence regarding the development of the internet. At first, online games supported text-based environments and continuously ended supporting complicated graphics and 3D virtual worlds. Additionally, it offered the choice not only to single-players but also to multiplayers to play together at the same time. [13]

A short review of the development of online gaming platforms has as follows: In 1978, Multi-User Domain (MUD) a multiplayer real-time virtual world which was most commonly used as text-based was presented. In 1990s, First-person shooter games were introduced. We observe the beginning of TCP/IP protocols being used on the internet. Moreover, the support of multiple players on online games begins to gain more and more ground. Online games at that time are: DOOM, Halo, Call of Duty: Modern Warfare 3, and more. In the late 1990s, Real-time strategy games appeared. At that time, most of the real-time strategy games support multiplayers from all around the world to play online together. Most known game is the Age of Empire. Continuously, Multiplayer Online Game (MOG) supports the possibility that many players can play an online game over the internet. This leads to the creation of Browser MOG (BMOG). [14]

Browser game (BMOG) could be developed by HTML, CSS, JavaScript, PHP, MySQL and etc. in order to be run in a web browser. A Browser game can support either single-player or multiplayers. In order to support a browser game for multiplayers, a web
server must be used. Furthermore, browser plug-ins such as Flash, Java, Shockwave, Silverlight, etc. was used for the development of graphics. WebGL [15] was also used for browser games for implementing 3D graphics without the use of plug-ins. The main feature that BMOG offered was portability. It was the beginning of a new era in browser online gaming because different devices such as mobiles and tablets could be used.

In the beginning of 2000 and then we watch the revolution of online gaming with the creation of several platforms such as PlayStation from Sony, Nintendo, and Xbox Live from Microsoft. Ultimately, the last few years Massively Multiplayer Online Games (MMOGs) are presented which continue to progress in the field of multiplayer online gaming. [13]

MMOGs are one of the most well-known technologies worldwide where it allows many players - even thousands - to play simultaneously. According to Statista [16], the most popular worldwide online game was World of Warcraft which on 2010 reached close to 12 million subscribers around the world.

1.2 Basic Architectures

Multiplayer Online Game (MOG) and Massively Multiplayer Online Game (MMOG) both of them have two basic architectures. The first one is the Client-Server architecture and the second is the Peer-to-Peer (P2P) architecture. There is a third architecture for MMOG which is the MultiServer architecture. We will describe all three architectures but for the present thesis our focus will be for the first two architectures.

MMOGs create traffic network and load processing. On the other hand, MMOGs must assert scalability, consistency, fast response time, and security. The most well-known architecture of MMOGs is client-server architecture. Thus, P2P architecture can offer features that emphasize the infrastructure of online games such as using multiple servers and reducing network bandwidth and computational power. P2P architecture can provide high scalability, low cost and good performance. Finally, another platform which shows to be promising is cloud-based gaming which also benefits from P2P architecture such as when it comes to 3D audio and video streaming. [17]

According to Ref. [17], there are three primary different MMOG architectures: The client-server architecture, the peer-to-peer architecture and the MultiServer architecture. In Table 1.1 we introduce the characteristics of the main architectures for MMOGs.
### Table 1.1: Basic Architectures for MMOGs and characteristics

<table>
<thead>
<tr>
<th>BASIC ARCHITECTURES FOR MMOGs</th>
<th>CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-Server Architecture</td>
<td>Well-known, simplicity, reliable, no scalability</td>
</tr>
<tr>
<td>Peer-to-Peer Architecture</td>
<td>High scalability, lower cost, good performance</td>
</tr>
<tr>
<td>MultiServer Architecture</td>
<td>Scalability, cost, complexity</td>
</tr>
</tbody>
</table>

The **Client-Server architecture** (shown in Figure 1.1) is well-known, successful and reliable. The clients request data from the server. Examples of this architecture are HTTP, FTP, and web services. The drawbacks are that it cannot achieve scalability, needs administration and some of the resources stay unused. [17]

![Figure 1.1: Client-Server Architecture](image)

The **Peer-to-Peer (P2P)** (shown in Figure 1.2) is distributed network architecture that has the capability to share resources among peers, provide direct communication between the peers of a network and the peers have a double role: they act as a client and as a server as well. P2P architecture provides and consumes data and there is no centralized data source. P2P provides higher scalability and lower cost. The challenges are to reduce bandwidth and to protect the system from cheating. [17]
Figure 1.2: Peer-to-Peer Architecture

It is worth mentioning some of P2P network features. As we mentioned above, each peer can be used as a client and as a server and are autonomous. Additionally, P2P network is dynamic where the peers can leave or access the network at any time and can communicate directly with each other; there is no central database, no peer has a view of the whole system, and each peer has access to all the resources meaning data and services. [18]

The benefits of using P2P network is because of the scalability, reliability, anonymity, privacy, decentralization/autonomy, robustness, minimum use of bandwidth, privacy/anonymity and resource sharing. P2P can be used in various applications such as multiplayer games, file sharing, content delivery, collaborative applications, and distributed computation. [18]

There are 3 types of networks P2P which are the following:

- Unstructured
- Structured
- Hybrid

The first type is Unstructured P2P networks. Unstructured defines a network that is not formed; it has no design of a particular structure of a network; it exhibits complete decentralization. Instead, the nodes are formed in a random way which instantly forms connections to each other. For instance, Unstructured P2P networks are Gnutella and Gossip protocols. The benefits are that it is easy to build; there is no administration, robustness, and low routing delay. The disadvantages are that it has no structure, uses a lot of memory and the only way to find specific data is by flooding the whole network so as to have more possibilities to find peers that can share the specific data. Another disadvantage is that the
consequence of flooding the network is the high traffic that arises which additionally provides no guarantee that the search of specific data will be a success. [19], [17]

The second type is *Structured P2P networks*. Structured P2P network is defined as a network that has a particular structure. As a result the search in the network is efficient. The most well-known type of Structured P2P network is the Distributed Hash Table (DHT). The peers in order to search for data in the network they use hash tables. A hash table has two variables the *key* and the *value*. Each pair (key, value) are saved in the DHT and any node in the network can retrieve the value having the given key. Chord, Pastry, Kademlia, P-Grid, etc. are protocols that belong to this type of network. The advantages are that it is less robust, supports large scale workloads, efficient routing. The disadvantage is that it has maintenance cost and high cost of discovering resources. [19], [17]

The third type is *Hybrid P2P models*. It is defined as a combination between P2P and client-server network. The most used hybrid model is the model that consists of a central server and provides help for locating peers. Hybrid models have a better performance due to the combination of structure and unstructured networks that it uses. [19] An example of Hybrid system is the Kazaa model.

In Table 1.2, we present a complete table with information that concerns the types of P2P networks, the protocols that P2P networks support and a short report of the pros and cons of each type of P2P networks.

<table>
<thead>
<tr>
<th>TYPES OF P2P NETWORKS</th>
<th>PROTOCOLS</th>
<th>ADVANTAGES</th>
<th>DISADVANTAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstructured</td>
<td>Gnutella, Gossip</td>
<td>Easy to build, no administration, robustness, low</td>
<td>No structure, uses a lot of memory, floods the whole network to find specific data, high traffic, no guarantee</td>
</tr>
<tr>
<td></td>
<td></td>
<td>routing delay</td>
<td></td>
</tr>
<tr>
<td>Structured</td>
<td>DHT, Chord, Pastry, Kademlia, P-Grid</td>
<td>Efficient, robust, supports large scale workloads</td>
<td>Maintenance cost, high cost of discovering resources</td>
</tr>
<tr>
<td>Hybrid</td>
<td>Kazaa model</td>
<td>Better performance</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.2: Types of P2P Networks
According to Yahyavi et al. [17], the **MultiServer architecture** (shown in Figure 1.3) can be separated into two categories. In particular, a server can either maintain a “shard” which is an instance of a game world or there is one game world which is divided into different zones maintained by different servers. As we observe in Table 1.1, multiserver architecture provides scalability and embodies several advantages of the client-server architecture. It can support a big number of players at the same time but its disadvantages are mainly the cost and the complexity of the architecture.

![MultiServer Architecture](image)

**Figure 1.3: MultiServer Architecture**

### 1.3 Distributed Hash Tables

Distributed Hash Tables (DHT) provides a way to search and locate nodes that store items (for instance a file) in a P2P network. In DHT, each node and key is assigned with an identifier using SHA-1 hash function. Each node and each key identifier is selected by hashing the node’s IP address or by hashing the key respectively. [20] DHT provides scalability, self-organizing, and robustness. The disadvantages are located in searching and security fields. [20]

A basic example of DHT is Chord P2P protocol [21]. Chord is a scalable lookup P2P protocol. Its function is that being a key provider, it can map the key to a node. Chord belongs to structure decentralized category of P2P networks. By using consistence hashing, it helps balancing the load. Additionally, each node uses a routing table so as to perform a lookup. Every node communicates with other nodes to get information for the lookup. Chord keeps the information that has been gathered in the routing table while nodes can leave or join the system. The information that the system keeps is calculated to be of the order of \(O(\log N)\) where \(N\) is the nodes. [21]
1.4 Pastry architecture

Pastry [22] is defined as a scalable distributed object location and routing application concerning a big area of P2P applications. Pastry can be used in several P2P applications such as data sharing, data storage, and group communication and naming. Pastry can be characterized as decentralized, scalable, efficient and self-organizing.

Each node in the Pastry network has a unique identifier called nodeId. When a message and a key is introduced, a Pastry node routes the message to the node with the nodeId that is arithmetically nearest to the key, among all currently active Pastry nodes. Every single Pastry node knows its neighbors at any time and has notifications if a new node appears, fails or recovers. Additionally, Pastry considers network locality which means that it searches a way to reduce the distance messages travel.

1.5 Related Work

There are many distributed multiplayer online games architectures that have been presented over the last years. We will introduce several architectures which are based on peer-to-peer architecture using Distributed Hash Tables (DHT), hierarchy of peers, etc.

Colyseus [23] is a distributed architecture for interactive multiplayer games. Colyseus is based on two architectures: Chord [21] architecture using Distributed Hash Tables (DHT) and Mercury [24] range-based DHT architecture. The use of the above architectures provides good performance, low-latency, better scalability and load balance and the ability of supporting many players in a game.

Over the years, authors introduced the use of P2P overlays so as to support multiplayer games on the web. SimMud [25] is a game implemented based on Pastry architecture where it uses Distributed Hash Tables in order to distribute game states.

In the game, the virtual world is divided into regions and each region has its own ID. Game regions are mapped by using Pastry key space. A node ID becomes the coordinator of the region if it is the closest to the region ID. The positions of each player are constantly up-to-dated by the coordinator. SimMud main characteristics are self-organization, flexibility and has lower deployment cost.
Hampel et al. [26] introduce a game architecture that combines Massively Multiplayer Online Games with a P2P network. Specifically, Pastry [22] technology is used in combination with PAST [27] and Scribe [28] technologies as well. Pastry technology is DHT-based overlay network that provides robustness, high scalability which leads to good performance. In order to achieve high availability with replications of objects, PAST technology is the right choice. Finally, Scribe technology is used for distributing events.

Rieche et al. [29], present a P2P-based infrastructure that supports MMOG. Specifically, the authors use CAN [30] technology approach which is a structured P2P system. The world game is divided in disjunctive zones and distributed on different nodes in the P2P network.

A middleware system called Real-Time Framework (RTF) is introduced by Glinka et al. [31] for developing scalable multiplayer online games. RTF can be used for distributing the game state between live servers. Additionally, it supports parallel state update computations and socket-based communication.

Jardine and Zappala [32] introduce a Hybrid architecture for MMOGs. Hybrid keeps centralized control of state and minimizes server bandwidth. This architecture combines client-server and P2P distribution systems.
CHAPTER 2: BACKGROUND TECHNOLOGIES & TOOLS

In this chapter, information will be provided for the technologies that are being used in this thesis. Firstly, WebSockets and WebRTC are introduced. Continuously, HTML5, WebGL, JavaScript and X3DOM are presented. Additionally, tools that are used in this thesis are described, namely the Node.js platform and the Socket.io real-time engine. Ultimately, related work for multiplayer online game is provided.

2.1 WebSockets

WebSockets is a web technology that provides bi-directional, persistent communication channels over Transmission Control Protocol (TCP) connections. It is a next generation client-to-server and server-to-client communication technology for web applications which operates over a single session. Specifically, WebSockets are developed so as to be used in web browsers and web servers. [33] WebSocket protocol is standardized by the IETF as RFC6455. Additionally, the WebSocket API is standardized by the W3C. The connection is established using port 80 and port 443 by default. Several browsers support WebSocket Protocol including Google Chrome, Mozilla Firefox, Safari, Opera, and Internet Explorer. [34]

The WebSocket Protocol is based on TCP protocol. The relationship between the WebSocket Protocol and the HTTP is the handshake. According to the RFC6455 standard [9], the WebSocket Protocol constitutes of two parts, the handshake and the data transfer. The handshake from the client side is presented as follows:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBzdWJtZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Table 2.1.1: Code Sample from RFC6455- Handshake client side (See Ref [34])
We can observe that the handshake gathers the following header-fields information: Firstly, it is an HTTP Upgrade request followed by: the GET method which helps to recognize the endpoint of the WebSocket connection; the Host field which concerns the WebSocket server that is used; the Origin which specifies where the application comes from; the Sec-WebSocket that has three different variables and specifies the key that concerns the security, the protocol that indicates the subprotocols and the version.

The handshake from the server is presented as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRBK+xOo=
Sec-WebSocket-Protocol: chat
```

<table>
<thead>
<tr>
<th>Table 2.1.2: Code Sample from RFC6455- Handshake server side (See Ref [34])</th>
</tr>
</thead>
</table>

We observe that from the server side we have the following information: At first, we have the HTTP Status-Line, the Upgrade and the Connection header fields complement the HTTP Upgrade. The Sec-WebSocket verifies and secures the WebSocket connection. Figure 2.1.1 represents the architecture of a WebSocket connection.

```
Request Handshake
(HTTP socket connection)
```

```
Accept Handshake (HTTP socket connection)
```

```
Send & Receive Data
```

```
Close socket connection
```

**Figure 2.1.1: WebSocket Connection**

At first, the connection between the client and the server must be established. Specifically, the first thing is for the client and the server to send their handshakes. If the handshake between them is a success then both client and server can start exchanging data or else messages. Messages can be transferred from the client to the server and vice-versa.
because as mentioned above WebSockets support two-way communication. Ultimately, when the messages are being delivered then the socket connection can be closed.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Meaning</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continuation Frame</td>
<td>RFC6455</td>
</tr>
<tr>
<td>1</td>
<td>Text Frame</td>
<td>RFC6455</td>
</tr>
<tr>
<td>2</td>
<td>Binary Frame</td>
<td>RFC6455</td>
</tr>
<tr>
<td>8</td>
<td>Connection Close Frame</td>
<td>RFC6455</td>
</tr>
<tr>
<td>9</td>
<td>Ping Frame</td>
<td>RFC6455</td>
</tr>
<tr>
<td>10</td>
<td>Pong Frame</td>
<td>RFC6455</td>
</tr>
</tbody>
</table>

Table 2.1.3: Opcode initial values (See Ref [34])

In Table 2.1.3, we have summarized the opcode initial values. Messages can consist of one or more frames. Consequently, each frame has a joined type. Some types are of the following form: textual data, binary data, and control frames (signaling).

According to base frame protocol, data are transmitted using a sequence of frames. Every frame type is defined with an opcode, a payload length, and payload data. Opcode is defined as the type of payload and has 4 bits. On the contrary, the number of bits for payload length and payload data varies.

When the WebSocket connection is established between client and server then messages can be transferred between them. Send() method can be used to send data and onmessage() event handler can be used so as to receive data. Data can be a string, a BLOB, or an ArrayBuffer object.

Additionally, we have WebSocket Event Handlers as follows in Table 2.1.4:

<table>
<thead>
<tr>
<th>Event type</th>
<th>Event Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Socket.onopen</td>
</tr>
<tr>
<td>message</td>
<td>Socket.onmessage</td>
</tr>
<tr>
<td>error</td>
<td>Socket.onerror</td>
</tr>
<tr>
<td>close</td>
<td>Socket.onclose</td>
</tr>
</tbody>
</table>

Table 2.1.4: WebSocket Event Handlers (See Ref. [34])
Open event is when connection is established. Message event is when the client receives data from the server. Error event is when an error appears in communication and close event is when the connection is closed.

In Table 2.1.5 we present a sample code of the event handlers that WebSocket technology employs. We observe that a socket is open locally using 8080 port and echo protocol is defined. We introduce the event handlers that are used in WebSockets which are the following: open, onmessage, onerror and onclose.

```javascript
var socket = new WebSocket('ws://127.0.0.1:8080', 'echo-protocol');

socket.onopen = function () {
    socket.send('Hello World');
};

socket.onmessage = function(text) {
    var receive_msg = text.data;
    console.log("Received message " + receive_msg);
};

socket.onerror = function (e) {
    console.log("Error:" + e);
};

socket.onclose = function () {
    console.log("Connection closed");
};
```

Table 2.1.5: Sample code of WebSocket Event Handlers (See Refs. [10], [35])

It is worth presenting recent developments and implementations using WebSockets technology. Below, a sample is collected and briefly reviewed.

Chen and Xu [36], presented a framework implementation for browser-based multiplayer online game combining HTML5, WebGL and WebSockets and several other web technologies. They were able to study and observe its performance and feasibility and concluded that using the above technologies for browser-based Multiplayer Online Games can uphold the interactivity of small groups of users without any problems. Wessels et al. [37] introduced a remote visualization system with the use of WebSockets and exploited new technologies for Web browsers so that anyone can have access to remote visualization. Komm is a real-time group communication software system which is developed and proposed by Zhangling and Mao. [38] The architecture of Komm is reliant on WebSocket.

25
In their paper, Zhangling and Mao compared Komm with old technologies such as Comet and Ajax, and concluded that Komm exhibits high usability and performance and low resource consumption.

Furthermore, Gutwin et al. [39] presented a study showing results from testing the performance of three different web-based networks where one of the tested web-based network uses WebSockets. Hämäläinen [40], referred to HTML5 technology and specifically to the use of WebSockets technology which provides full-duplex communication channels. Liu and Sun [41], also the use of WebSockets in web real-time communication. In their paper, they compare and resolve the differences between WebSocket Protocol and HTTP protocol. They show that WebSocket protocol can reduce both network traffic and latency.

Hai et al. [42] referred to IMS (IP Multimedia Subsystem) and tried to address the compatibility issue between the User Equipment and the network of the service provider. The authors studied two approaches. The one is the application download and the other is the usage of HTML5/WebSocket approach.

Kapetanakis et al. [43] presented a 3D web application using X3DOM, WebSockets, and HTML5 technologies. Based on these technologies they present architecture for interactive 3D streaming over the web.
2.2 WebRTC

Introduction

Over the years the Web has managed to overcome several obstacles from the past developing and introducing implementations that can enable synchronous communications over the web. One of the challenges that the developers had to overcome was the ability to transfer video, audio and data over the web without using plugins, such as Flash, Java Applets, etc. To this end, the IETF (Internet Engineering Task Force) and the W3C (World Wide Web Consortium) worked together and developed the WebRTC (Web Real-Time Communication) technology. In this alliance, the IETF specifies the networking protocols and the W3C the JavaScript APIs including access to local media devices such as microphones and web cameras. [44] The JavaScript APIs of WebRTC are compatible with any browser that supports WebRTC, so users can establish communication for audio, video and generic data.

WebRTC is an open project supported by Google, Mozilla and Opera. The goal of WebRTC is to enable real-time communication among browsers in a peer-to-peer communication utilizing JavaScript and HTML5 without the use of plugins. For instance, applications such as video calls, video-conferencing, text chat, online games, etc. are some of the applications that can be supported with this real-time communication technology. A general overview of WebRTC for audio, video and data transfer between browsers and other applications is given in [45], [46].

WebRTC is characterized mainly by its architecture where it includes session management and signaling, peer connection, audio and video engines and data transport. In this section a brief review of the WebRTC architecture will be presented including the necessary technical detail related to APIs, topologies and codecs. Part of this review has been included in a book chapter on “Towards ubiquitous and adaptive web-based multimedia communications via the cloud” that is to be published in 2015 by IGI Global. [47]
The WebRTC Architecture

According to RTCWEB overview specification [44], IETF and W3C are responsible for RTCWEB/WEBRTC technology. IETF specifies the protocol specification and W3C works on the JavaScript API specification which includes the use of media devices such as microphones and web cameras. As a result, together, IETF and W3C offer an environment where JavaScript API is used and can be compatible by any browser that supports WebRTC technology. Moreover, the user can establish communication for audio, video and generic data.

The WebRTC Browser architecture is based on two core interfaces as Figure 2.2.1 depicts. The first interface refers to the Protocols that browsers apply in order to have a direct peer-to-peer connection between them without servers to intervene. The second interface refers to the APIs that browsers provide to JavaScript applications so they benefit by the underlying WebRTC infrastructure.

![Figure 2.2.1: WebRTC Browser Model](image)

The trapezoid model of the generic WebRTC architecture, shown in Figure 2.2.2, is inspired by the SIP (Session Initiation Protocol) based communications. As it is depicted, each browser is associated with a WebRTC server, situated in its network domain, which is responsible for session initialization and negotiation with its opposite server. We assume that each browser has downloaded via a web server a JavaScript web application exploiting the local WebRTC APIs. Upon successful session establishment, a media and data path connecting directly the two browsers opens. Signaling messages exchanged between
browsers and servers and/or between servers are responsible for the management and closing of such sessions. WebRTC servers can modify, translate and manage the messages as required. Between browsers and WebRTC servers various protocols can be used for their communication including HTTP, and WebSockets.

![Figure 2.2.2: Generic WebRTC architecture](image)

**WebRTC**

In the following sections, attention will be paid on the required signaling and how WebRTC embodies it. Next, the main APIs will be discussed and the enabling STUN, TURN and ICE network protocols will be analyzed. Then, the potential topologies will be presented and, finally, information will be provided about the Codecs that WebRTC uses.

**Signaling**

Signaling refers to a mechanism to coordinate session management and send control messages. Signaling methods and protocols are not specified by WebRTC, hence any relevant protocol can be used as carrier (e.g. SIP or XMPP). It exchanges three types of information: session control messages, network configuration and media capabilities. [48]

Signaling is used during session establishment or update to exchange an offer and an answer between browsers. SDP (Session Description Protocol) is the legacy protocol that can be used to this end. For example, the SDP protocol can be used for the capabilities negotiation between browsers while setting up or updating a peer connection. As an alternative to SDP the WebRTC specifies JSEP (JavaScript Session Establishment Protocol)
for implementing such offer/answer exchanges. JSEP signaling (see Figure 2.2.3) can be used in the WebRTC framework as SDP in SIP-based communications that is to describe and negotiate a session between two browsers via the mediating session managers. Once session is established, media and data can be transferred directly between the two browser peers. To illustrate how open and flexible the WebRTC architecture is, JSEP signaling can be transferred from a browser to a WebRTC server via WebSockets or HTTP and then to be translated to SIP/SDP signaling, enabling thus communication between a browser and a VoIP phone.

![JSEP architecture](image)

**Figure 2.2.3: JSEP architecture**

**Code Sample**

In Figure Table 2.2.1, we see a sample code from the WebRTC W3C Working Draft [49], which presents signaling and how it is used.

```javascript
var signalingChannel = new SignalingChannel();
var configuration = { "iceServers": [{ "url": "stun:stun.example.org" } ] };
var pc;

// call start() to initiate
function start() {
    pc = new RTCPeerConnection(configuration);

    // send any ice candidates to the other peer
    pc.onicecandidate = function (evt) {
        if (evt.candidate)
            signalingChannel.send(JSON.stringify({ "candidate": evt.candidate }));
    };
}"
```
// let the "negotiationneeded" event trigger offer generation
pc.onnegotiationneeded = function () {
    pc.createOffer(localDescCreated, logError);
}

// once remote stream arrives, show it in the remote video element
pc.onaddstream = function (evt) {
    remoteView.srcObject = evt.stream;
};

// get a local stream, show it in a self-view and add it to be sent
navigator.mediaDevices.getUserMedia({ "audio": true, "video": true }, function (stream) {
    selfView.srcObject = stream;
    pc.addStream(stream);
}, logError);

function localDescCreated(desc) {
    pc.setLocalDescription(desc, function () {
        signalingChannel.send(JSON.stringify({ "sdp": pc.localDescription }));
    }, logError);
}

signalingChannel.onmessage = function (evt) {
    if (!pc)
        start();

    var message = JSON.parse(evt.data);
    if (message.sdp)
        pc.setRemoteDescription(new RTCSessionDescription(message.sdp), function () {
            // if we received an offer, we need to answer
            if (pc.remoteDescription.type == "offer")
                pc.createAnswer(localDescCreated, logError);
        }, logError);
    else
        pc.addIceCandidate(new RTCIceCandidate(message.candidate),
            function () {} , logError);
};

function logError(error) {
    log(error.name + ": " + error.message);
}

Table 2.2.1: Code Sample for Signaling (See Ref. [49])

31
WebRTC main APIs

WebRTC consists of three main APIs

- MediaStream
- RTCPeerConnection
- RTCDDataChannel

MediaStream API

MediaStream API uses getUserMedia() method to obtain audio and video. There are two main components in the MediaStream API which are the MediaStreamTrack and MediaStream interfaces. The MediaStreamTrack object refers to media sources for instance audio or video; it makes use of several groups of MediaStreamTrack objects combined into one unit that can be rendered in a media element or recorded. [50]

As we can observe from the Figure 2.2.4, MediaStream has an input, which it could be generated by the navigator.getUserMedia(), and an output, which it could be either a video element or an RTCPeerConnection.

![Figure 2.2.4: MediaStream architecture from (See Ref. [50])](image)

Specifically, with getUserMedia() a web application can request access to local media devices such as, for instance, the user’s camera and microphone. Then, the MediaStream API can display the media streaming content and send it to a remote user. The web browsers that currently support the MediaStream API are the Chrome, Opera and Mozilla Firefox browsers. [45], [48]
In Table. 2.2.2 below, a sample code is presented for obtaining a local video stream [51]. \textit{Constraints} adjust the contents of \texttt{MediaStream}, for instance media type, resolution, etc. In \texttt{successCallback()} function we get the video stream from the webcam. The \texttt{errorCallback()} function is used if an error appears. Finally, on the last line, \texttt{getUserMedia()} is called having three parameters: The constraint object and the two functions mentioned above, \texttt{successCallback} and \texttt{errorCallback}.

```javascript
var constraints = {video: true};
function successCallback(localMediaStream) {
    window.stream = localMediaStream;
    var video = document.querySelector("video");
    video.src = window.URL.createObjectURL(localMediaStream);
    video.play();
}
function errorCallback(error){
    console.log("navigator.getUserMedia error: ", error);
}
navigator.getUserMedia(constraints, successCallback, errorCallback);
```

Table 2.2.2: Sample code for video stream (See Ref. [51])

**RTCPeerConnection API**

The **RTCPeerConnection API** is used for session management, audio and video communication between peers and the provision of stable and efficient communication of streaming media. The RTCPeerConnection API at first implements session negotiation via the serving WebRTC servers and upon successful establishment of the session, it provides orchestration of audio and video calling (including signal processing and codec handling), media encryption and bandwidth management. Finally, it enables the browser-to-browser communication.

Security and user privacy is seriously taken into account by WebRTC. Hence it takes care to protect users and devices. Specifically, the encryption for media and data is mandatory using protocols such as Secure Real Time Protocol (SRTP) and Datagram Transport Layer Security (DTLS). SRTP provides encryption, integrity and authentication messages. DTLS defines communication privacy for datagram protocols [52]. Finally, another feature of WebRTC security is that it uses a mechanism called sandboxing that helps
users secure their computers from malicious threats [45], [48]. In Table 2.2.3, a sample code is presented for creating RTCPeerConnection [51].

```
function createConnection() {
    var servers = null;
    window.localPeerConnection = new webkitRTCPeerConnection(servers,
        {optional: [{RtpDataChannels: true}]});
    trace('Created local peer connection object localPeerConnection');

    try {
        // Reliable Data Channels not yet supported in Chrome
        sendChannel = localPeerConnection.createDataChannel("sendDataChannel",
            {reliable: false});
        trace('Created send data channel');
    } catch (e) {
        alert('Failed to create data channel. ' +
            'You need Chrome M25 or later with RtpDataChannel enabled');
        trace('createDataChannel() failed with exception: ' + e.message);
    }
    localPeerConnection.onicecandidate = gotLocalCandidate;
    sendChannel.onopen = handleSendChannelStateChange;
    sendChannel.onclose = handleSendChannelStateChange;

    window.remotePeerConnection = new webkitRTCPeerConnection(servers,
        {optional: [{RtpDataChannels: true}]});
    trace('Created remote peer connection object remotePeerConnection');

    remotePeerConnection.onicecandidate = gotRemoteIceCandidate;
    remotePeerConnection.ondatachannel = gotReceiveChannel;

    localPeerConnection.createOffer(gotLocalDescription);
}
```

Table 2.2.3: Sample code for RTCPeerConnection (See Ref. [51])

**RTCDataChannel API**

*RTCDataChannel API* empowers the interchange of arbitrary data using peer-to-peer communication between peers. Data exchange takes place over bidirectional Data Channels where incoming and outgoing user data are encapsulated within SCTP (Stream Control Transmission Protocol) packets. SCTP is specified by the IETF as RFC 4960 standard [53] and has been selected as the transport protocol for Data Channels. The
RTCDataChannel API can be used in gaming applications, file transfer, remote desktop applications, real-time text chat, etc. The main features of RTCDataChannel are low latency, high throughput, does not need a server in-between the two peers to exchange data, tends to be much faster than WebSockets and can be secure (over DTLS). RTCDataChannel supported by SCTP protocol are available in Google Chrome, Mozilla Firefox, and Opera browsers as well as in Android. [45], [48]

RTCDataChannel supports a bi-directional connection which resembles a lot of the WebSockets connection. As such, when a connection is established, it can send messages, close the connection or send error notifications. Specifically, it supports strings and some of the binary types in JavaScript such as Blob and ArrayBuffer. The above types can be useful in applications for instance file transfer and multiplayer gaming [1].

In Table 2.2.4, we have gathered from [49] and [54] the basic attributes of RTCDataChannel.

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCDataChannel.ordered</td>
<td>A Boolean type that returns either true which means it guarantees the data will be delivered in order or false which means the data will be delivered out-of-order (no guarantee).</td>
</tr>
<tr>
<td>RTCDataChannel.protocol</td>
<td>A DOMString type is returned that has the name of the sub-protocol that is being used or it returns an empty string.</td>
</tr>
<tr>
<td>RTCDataChannel.readyState</td>
<td>A RTCDataChannelState type that represents the state of the data connection. It returns an enum that it can be either of the following four: “connecting”, “open”, “closing”, “closed”.</td>
</tr>
<tr>
<td>RTCDataChannel.id</td>
<td>An unsigned short type that returns the id of the RTCDataChannel. It is set at the beginning of the RTCDataChannel creation.</td>
</tr>
<tr>
<td>RTCDataChannel.label</td>
<td>A DOMString type which returns a unique label of each RTCDataChannel object is used.</td>
</tr>
<tr>
<td>RTCDataChannel.bufferedAmount</td>
<td>An unsigned long type that returns the number of bytes that have been queued enabling send().</td>
</tr>
<tr>
<td>RTCDataChannel.binaryType</td>
<td>A DOMString type that returns the value “blob” as a default or the value which was</td>
</tr>
</tbody>
</table>
**RTCDataChannel.reliable**

A Boolean type which returns either true which supports reliable mode or false which supports unreliable mode.

| Table 2.2.4: RTCDataChannel API Attributes (See Refs. [49], [54]) |

In Table 2.2.5, we present the event handlers of RTCDataChannel based on [49] and [54].

**EVENT HANDLERS**

<table>
<thead>
<tr>
<th>RTCDataChannel.open</th>
<th>The channel is open when dataChannel connection is established.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCDataChannel.onmessage</td>
<td>The event is sent when a message is available to be sent via the channel.</td>
</tr>
<tr>
<td>RTCDataChannel.onclose</td>
<td>The event is sent when dataChannel is closed.</td>
</tr>
<tr>
<td>RTCDataChannel.onerror</td>
<td>The event is sent when an error is presented.</td>
</tr>
</tbody>
</table>

| Table 2.2.5: RTCDataChannel API Event Handlers (See Refs. [49], [54]) |

In Table 2.2.6, we introduce the two methods of RTCDataChannel based on [49].

**METHODS**

<table>
<thead>
<tr>
<th>RTCDataChannel.close()</th>
<th>Channel is closed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCDataChannel.send()</td>
<td>Data are sent via the channel.</td>
</tr>
</tbody>
</table>

| Table 2.2.6: RTCDataChannel API Methods (See Ref. [49]) |

<table>
<thead>
<tr>
<th>TCP</th>
<th>UDP</th>
<th>SCTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>Reliable</td>
<td>Unreliable</td>
</tr>
<tr>
<td>Delivery</td>
<td>Ordered</td>
<td>Unordered</td>
</tr>
<tr>
<td>Transmission</td>
<td>Byte-oriented</td>
<td>Message-oriented</td>
</tr>
<tr>
<td>Flow Control</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Congestion Control</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

| Table 2.2.7: Compering TCP, UDP and SCTP protocols (See Refs. [55], [56]) |

Moreover, as shown in Tables 2.2.4 and 2.2.7, RTCDataChannel can work either way, reliable or unreliable. Reliable means using TCP protocol. It guarantees the transmission of messages as well as they are delivered in the right order. This has as a result
to make the transmission slower. Unreliable means using UDP protocol. UDP does not guarantee that each message will be delivered to the other peer and also it does not guarantee the right order of the messages. As a result, the overhead for retransmissions and acknowledges is removed which gives RTCDataChannel the opportunity to be faster. [1]

In more specific, RTCDataChannel API performance is more or less the same when we use either mode. There are no packet losses. But, if a packet is not properly delivered, or get lost, in reliable mode the packets that follow will be blocked and there be a delay until the lost packet is being retransmitted. Multiple data channels can be used within the same session to avoid such problems. [1]

In addition, TCP protocol supports transmission based on byte-oriented transfer and also provides flow and congestion control mechanisms. On the contrary, UDP protocol supports message-oriented transmission and does not guarantee the control of the flow or of the congestion mechanisms. But with RTCDataChannel we have a third option, that of SCTP protocol. SCTP is a transport protocol that is set over DTLS for security. Alternatively, SCTP can run over UDP as shown in Figure 2.2.5.

![Diagram](image)

**Figure 2.2.5: Basic Diagram (See Ref. [57])**

In Figure 2.2.6, we observe the WebRTC Protocol layers. In WebRTC, for security reasons, encryption is obligatory. Specifically, RTCDataChannel use Datagram Transport Layer Security (DTLS) so as to secure all data. All browsers that support WebRTC are standardized to have DTLS. If large amount of data is needed to be transferred with security then FileSystem API is recommended as a good solution to the issue [1] In contrast, audio and video data are transmitted using the Secure RTP (SRTP) protocol [57]. As we saw in Figure 2.2.5, arbitrary data, via the RTCDataChannel, are transmitted using SCTP over DTLS so as to secure the data transfer.
In Table 2.2.8, a sample code is presented for using RTCDATAChannel functions [1]. We observe that a peer connection has been established and a dataChannel has been created having its own label (“myLabel”) and defining the dataChannel settings. For instance, in this case, dataChannelOptions is defined and has two parameters. The first parameter does not guarantee order for data transmission. The second parameter sets the maximum retransmission time to send a failed message at 3000 milliseconds. Continuously, RTCDATAChannel uses the functions onerror, onmessage, onopen, and onclose.

```javascript
var peerConnection = new RTCPeerConnection();

var dataChannelOptions = {
    ordered: false, // do not guarantee order
    maxRetransmitTime: 3000, // in milliseconds
};

// Establish your peer connection using your signaling channel here
var dataChannel = peerConnection.createDataChannel("myLabel", dataChannelOptions);

dataChannel.onerror = function (error) {
    console.log("Data Channel Error:", error);
};

dataChannel.onmessage = function (event) {
    console.log("Got Data Channel Message:", event.data);
};
```
File Sharing

According to Ristic [1], another function of RTCDdataChannel that is worth mentioning is the file sharing and the possibility of connecting to multiple clients. In order to transfer large files File API must be used. Additionally, in order to establish the connection RTCPeerConnection is used and for the creation of data channels among peers RTCDDataChannel is used.

A few points that should be considered when attempting to send files over RTCDDataChannel are: the file size, file transfer speed and the chunk size. First of all, the file size of which we want to send from one peer to another. There are two solutions of how to handle a file size. When the file size is small, it can be saved and loaded as a Blob, then the solution is to use File API and use afterwards a reliable channel (meaning TCP protocol) to send the file. On the other hand, if the file size is big, then we have to use a solution that requires the use of chunks. The procedure has as follows: First the file is loaded and saved to an offline storage using for instance FileSystem API and then the file is sent to another peer, having a chunkId metadata for recognition. [1]

The speed of which the file size is send, it depends on the transport, if it is reliable (TCP) or unreliable (UDP). There is no absolute answer because each file transfer is send under different conditions. Finally, the chunk size is the smallest data size for an application. Because of the size limit when sending a file, chunk size is required. For the time being, the maximum acceptable chunk size is 16KB. [1]
WebRTC Network Protocols

NAT (Network Address Translation) give the opportunity to each device in a private local network to get a public IP address. However, WebRTC cannot be used without a public IP address at the session negotiation phase [58]. Hence, to achieve NAT traversal, WebRTC uses three different types of network protocols, namely:

- STUN
- TURN
- ICE

STUN

STUN stands for Session Traversal Utilities for NAT and is a client-server protocol. The server part sits behind a NAT so it discovers the user’s public IP address and TCP port and let the STUN client be aware of them. This procedure helps a WebRTC peer to get its public IP address and port, and passes them via WebRTC signaling to the other peer (see Figure 2.2.7) [58]. Hence, STUN gives permission to the WebRTC media and data flows (for instance real-time voice, video, and messaging) between peers.

Figure 2.2.7: STUN Network Architecture
**TURN**

*TURN* stands for Traversal Using Relays around NAT and is used as a fallback solution to STUN when STUN cannot be used. A TURN server is used as a relay for the audio, video and data streams between peers (see Figure 2.2.8) [58]. In particular, it offers to a WebRTC peer the required public IP address for communication outside its LAN.

![Figure 2.2.8: TURN Network Architecture](image)

**ICE**

*ICE* stands for Interactive Connectivity Establishment and is a framework for connecting peers. The mission of ICE is to try finding the best solution for WebRTC peers choosing between STUN and TURN.

At first, ICE tries to connect peers directly via STUN. If that fails, then ICE uses a TURN relay server [58].

**Network Topologies**

WebRTC is very flexible to support any network configuration. Depending on the application and the number of the participating peers in a WebRTC session, several types of network topologies can be established.

The simplest connection that concerns the communication between just two peers is the direct **peer-to-peer connection** between the two endpoints (see Figure 2.2.9). [48]
For conference-like WebRTC sessions, where more than two peers participate, *mesh*, *star* and *MCU* topologies can be set up. In a **Mesh topology** each peer connects directly to all other available peers maintaining several peer-to-peer connections (Figure 2.2.10). A Mesh topology is easy to be established. The disadvantages are that it can be used for limited peers and that it requires from the application to multiplex media and data from all open connections so a conference is enabled. [48]

![Mesh topology diagram](image)

**Figure 2.2.10: Mesh topology**

In a **Star topology** one peer is assigned the role of coordinator and maintains the connections with any other peer. The central peer is also responsible to send the multiplexed data to all the other peers (Figure 2.2.11). The disadvantage is that the central peer should be robust enough to undertake this load, while there are not special requirements from the other peers. [48]

![Star topology diagram](image)

**Figure 2.2.11: Star topology**
The most robust network topology for conference-style communications is the MCU (Multipoint Control Unit) (Figure 2.2.12). MCU is a mediating server that can maintain connections with a great number of participants and distribute multiplexed media content to them. Whenever a peer decides to leave a conference the MCU simply terminates this connection. A MCU can handle different resolutions and codecs to enable interoperability between different peers. It is mainly used for video conferencing [48].

![Figure 2.2.12: MCU topology](image)

**WebRTC Codecs**

Currently WebRTC supports the Opus, iSAC and iLBC audio codecs as well as the VP8 video codec.

**Opus** is an open source and royalty free audio codec developed by IETF and standardized as RFC 6716 in 2012 [59]. It can be used in both, speech interaction and music transmission via the Internet. In order to create Opus, Skype and Xiph.org collaborated offering their SILK and CELT codecs technology respectively. Opus can be used in a wide range of real-time communications including interactive applications such as Skype, WebRTC applications, Voice over IP, chat in games, videoconferencing, etc. Opus provides high quality audio and it works from low bitrate narrowband speech at 6 kbit/s to very high quality stereo music at 510 kbit/s. Furthermore, some of the features that Opus provides are: different sampling rates from 8 kHz to 48 kHz, packet loss concealment, frame sizes from 2.5 ms up to 60 ms, dynamically adaptable bitrate, audio bandwidth and frame size, up to 255 channels [60].

**iSAC** stands for internet Speech Audio Codec and is a robust, and adaptive wideband audio and speech codec. It provides low delay and as a result it offers good quality in real-
time communications. iSAC is developed by Global IP Solutions and is appropriate for streaming audio and Voice over IP (VoIP) applications. Google Talk is one of the applications that use iSAC codec [61], [62].

**iLBC** stands for internet Low Bitrate Codec and is a royalty free narrow band speech codec. It was developed by Global IP Solutions and is used in many Voice over IP (VoIP) and streaming audio applications [63].

**VP8** is a highly efficient video compression technology that was developed by On2 Technologies. In 2010, Google acquired On2 and made VP8 available as part of the WebM Project. WebM is a royalty-free media file format designed for the web and defines the file container structure for both video and audio streams. The structure of WebM files is based on Matroska container [64].

**WebRTC Use Cases**

According to the official WebRTC website [11], more than 1,000,000,000 users have made use of WebRTC open source technology so far. Ericsson Labs was the first that implemented a WebRTC application in 2011 [65]. In 2013, Mozilla in cooperation with Google presented the successful connection between Firefox and Chrome browsers using the WebRTC RTCPeerConnection API without the need for plugins [66].

Nowadays, we can find several open platforms based on the WebRTC technology that provide users with the ability to customize the platform according to their needs. The application domains can vary from telemedicine and education to media, entertainment and video conference. Tokbox [67], BlueJeans [68], Zingaya [69] are some of these platforms. In addition, GearCloud Labs has recently presented a platform called Mixology that provides multiple live video streams and interactive graphics in real-time [70]. Furthermore, Google has added a plugin called Google Hangouts which is based on WebRTC technology and offers to users plenty features such as group conversations and more [71], [72].

It is worth mentioning several studies that have been developed in the last years concerning WebRTC technology, Peer-to-Peer communication, and HTML5.

Firstly, Nurminen et al. [73], proposed Peer-to-Peer (P2P) streaming for Video on Demand (VoD) using HTML5 and WebRTC technology. They believe it is a good alternative solution according to nowadays demands. Video on Demand services augmented rapidly in the last years and are responsible for a big part of the internet traffic nowadays.
The authors’ proposal for P2P video streaming between browsers has the advantage that it can be used without the use of plugins. But there are limitations such as the resources via browser are much more limited than for native applications and another limitation concerns mobile devices due to limited CPU and bandwidth capacity. (HTML5 provide browsers the opportunity to connect with each other in real-time using APIs.)

Rodriguez et al. [74], referred to videoconferencing services using browser real-time communication and more specifically the use of WebRTC technology. They proposed the Multipoint Control Unit (MCU) architecture as a good solution for advanced videoconferencing.

Lopez Fernandez et al. [75], mentioned that WebRTC technology gives the opportunity to converge www, desktop and mobile multimedia real-time communications services. They proposed Kurento, a media server technology which is developed using an open source software and to be compatible with WebRTC. Kurento provides interoperability between mobile and desktop real-time communication services.

In recent years, web games have gained ground in the market. According to Zhang [76], there are still problems in content experience effect and operation. The author proposes on how to improve experience effect and to optimize system architecture, database and cache using HTML5 technology. Final goal is the web game to become faster, more stably and be used not only in one platforms but be compatible in many platforms for instance in mobile internet robots.

Vogt et al. [77], proposed an architecture named Browser-based Open Publishing (BOPlish) for decentralized content-publishing among browsers through WebRTC Data Channels. Meaning, users can share or retrieve content without needing to install a web server or depending on third-party providers.

Trajkovska et al. [78], referred to the opportunities and challenges of implementing P2P streaming applications in the web. They analyzed the challenges and the potentials of extending WebRTC use so as to develop JavaScript APIs for P2P streaming algorithms.

Johnston et al. [79], discuss the use of WebRTC within an enterprise especially in security area such as access control, firewall traversal, and peer-to-peer data flows. In addition, they examine the use of policy compliance for instance in recording, logging, etc.

Amirante et al. [80], focused on integration of legacy SIP-based systems with the browser-enabled architectures. The authors identified the interoperability requirements and then presented examples dealing with the integration of RTCWEB clients into an existing standard-based collaboration platform.
In addition, Becke et al. [81] referred to RTCWeb protocol along with the use of SCTP (Stream Control Transmission Protocol) protocol. They mainly focused on non-media data transfer. They studied the performance of non-media data and how the SCTP protocol can be improved in order to reduce the delay by applying an appropriate stream scheduler and support for the interleaving of user messages. The authors observe and study the performance impact of non-media data by using several congestion control algorithms for SCTP. Finally, Eriksson et al. [82] introduced WebRTC and HTML5. They discussed the potentials that a web platform provides to people with a big variety of content.

Another future aspect is the potential to combine WebRTC with cloud technology. Specifically, cloud-based gaming has begun to stand out in the field of online gaming.

Many companies try to combine applications based on WebRTC with cloud technology. For instance, Aculab Company [83] developed Aculab Cloud WebRTC demo. In addition, Mantis [84] is considered to be the next-generation Cloud technology for WebRTC. Cloud Platforms that are being developed for online games and are known in the field of cloud gaming are: OnLive Games [85] and Gaikai [86].

In recent scientific papers we read about new studies that join the field of cloud. Specifically, Cai [87] present mobile cloud-based video gaming (MCVG). Furthermore, in 2011 a survey was conducted by Dinh et al. [88] on mobile cloud computing. The survey introduced mobile cloud computing, analyzing pros and cons, presented applications that can be used in the fields of mobile learning (m-learning), mobile healthcare (m-healthcare), mobile commerce (m-commerce), mobile gaming (m-game) and real-time experience for instance travel, shopping, etc.

According to Dey [89], Cloud Mobile Media combine three different developments which concern device, platform and network. Specifically, the use of smartphones and tablets, the use of public clouds and the use of mobile broadband networks globally. The above developments are the new generation of ubiquitous multimedia services on mobile devices. In this paper, Dey presents not only the opportunities and the challenges but also possible directions to confront the challenges. In another study, the authors Suselbeck et al. [90] propose the reduction of additional delay in MMOGs using the Cloud.

Shea et al. [91], present state-of-of-the art cloud gaming platforms such as OnLive Games [85] and Gaikai [86] and measure their performance, latency, streaming quality. Additionally, they present the challenges that can occur in cloud gaming. In addition, Alonso et al. [92] introduce MCU in the cloud. Moreover, Kyoung III et al. [93], presented cloud-based gaming service platform that supports multiple devices. Finally, Huang et al.
introduced an open cloud gaming system called GamingAnywhere.

In conclusion, the short period (2012-2014) over which the above scientific works have been reported is worth noticeable. That is because we observe the evolution, the rapid development, the several combinations, and the further potentials of the WebRTC. From the above references it is clear how WebRTC has evolved in the use of real-time communication among browsers in a peer-to-peer communication without the use of plugins utilizing JavaScript and HTML5. Finally, recent scientific works also show that cloud technology can be combined with WebRTC technology. The combination of the two technologies is very promising and interesting developments have been actualized concerning cloud web gaming and cloud mobile media.

2.3 X3DOM

The combination of WebGL with JavaScript has led to the development of X3DOM, an open-source framework that allows us to work with 3D content without the use of plugins. WebGL is a cross-platform and royalty-free web standard for 3D graphics based on OpenGL ES 2.0. X3DOM is a combination of X3D (Extensible 3D Graphics) and DOM (Document Object Model). X3D is a royalty-free standard for declarative 3D content and DOM provides representations and interaction with objects in HTML documents. It is compatible with Chrome, Firefox, Opera browsers for desktop and iOS, Sony and Firefox for mobile platforms. The aim of X3DOM is that by having a real time X3D scene, it offers the opportunity to change the 3D content by adding, removing or changing DOM elements. In addition, HTML events are supported on 3D objects such as for instance the “onclick” event.

It is utilized to create declarative 3D content in HTML5 web pages. Declarative 3D content means that an interactive 3D scene can be created and displayed utilizing a structured and textual representation instead of writing code. Textual representation in this case means that it is part of an HTML document that expresses a web page.

Ultimately, the open-source Apache/XAMPP server is used in order to run X3DOM which can be downloaded for free.

In the world of gaming, several Web games are developed using mainly 3D graphics in order to implement virtual worlds, etc. Some of the latest technologies that have been used are WebGL and X3DOM. In fact, the X3DOM technology is friendly to use with many
potentials. For instance, Jung at al., [100] after having realized that the major driving technology for the documentation and presentation of culture driven media is real-time 3D content, they employed the virtual museum (VM) model for communication needs. Real-time 3D content is delivered through web browsers within the X3Dom framework. The authors describe application scenarios and measure technological requirements, so as to present virtually, via the web, the cultural heritage artifacts kept in museums.

Mao et al., [101] presented a framework for visualizing 3D city models online via browsers. The framework is based on City Geography Markup Language (CityGML) and X3D and is supported by HTML5. The authors proposed several methods and structures in different scales (blocks, buildings, and facades) in order to display detailed 3D city models and handle the big amount of data. The outcome shows that the applied methodology exhibits visual resemblance of the original 3D city models.

Sons et al.[102], referred to interactive 3D graphics for browser-based applications using XML3D, DOM scripting, CSS, and HTML. Behr et al. [103], proposed an architecture that provides HTML/X3D integration model X3DOM.

**Code Example**

In Table 2.3.1, we observe an x3d scene and the creation of a red box.

```xml
<x3d width='600px' height='400px'>
  <scene>
    <shape>
      <appearance>
        <material diffuseColor='1 0 0'/></appearance>
      <box></box>
    </shape>
  </scene>
</x3d>
```

Table 2.3.1: Code Sample of an X3D Scene and the creation of a red box (See Ref. [104])
2.4 HTML5

HTML5 [105] replace the previous versions of HTML 4.01, XHTML 1.0 and XHTML 1.1. HTML5 is a markup language that is used for structuring and display content on the internet. HTML5 is supported from all the latest versions of web browsers such as Apple Safari, Google Chrome, Mozilla Firefox, Opera and mobile web browsers as well. HTML5 was created from both the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). In 2012, it was first released and by the end of 2014 HTML5 is released as a W3C Recommendation.

There are new characteristics that HTML5 supports which are the following: HTML5 provides new potentialities and functions in audio, video, and canvas elements which are designed so that multimedia and graphical content can be used on the web without plugins. Especially as it concerns 3D graphics, the canvas element provides dynamic rendering for interactive 3D and 2D graphics within any compatible web browser without the use of plugins. Furthermore, WebSockets are supported that provides a next-generation bidirectional communication technology for web applications. In addition, Server-Send Events (SSE) is presented in HTML5. SSE is defined as when a web page is automatically updated from a server. Moreover, HTML5 gives the opportunity to users to share their location using Geolocation. Another feature is Drag-and Drop characteristic where the user can move items in a web page from one side to another. Finally, apart from having new semantic elements such as <header>, <footer> and <section>, HTML5 supports microdata which allows developers to create their own vocabulary and create custom semantics. [106]

The new specification recommendation of HTML5.1 will be released by the end of 2016. [107]

2.5 JavaScript – CSS

JavaScript (JS) is a dynamic lightweight and object-oriented scripting programming language used in the Web. ECMAScript language specification standardized JavaScript and from June 2011 the latest version of ECMA Script 5.1 is being used. [108]

JavaScript is used mainly in the development of HTML pages. It adds client–client-side behavior in a web page. Furthermore, it is supported not only from client-side but also from server-side; used in runtime environments such as Node.js [109], game developments,
desktop and mobile applications. [110] Additionally, HTML pages that conclude JavaScript and scripts also interact with the Document Object Model (DOM). Web browsers produce ‘host objects’ so as to delineate the DOM in JavaScript. [110], [108]

Cascading Styling Sheet (CSS) is a style sheet format determining the layout of an HTML page. W3C retains CSS specification and RFC 2318 [111] provides registration for the use of Internet Media Type (MIME type) with CSS. [112]

### 2.6 Tools

#### 2.6.1 Node.js

For our development, we used Node.js. Node.js is a free open-source platform which is built on Chrome's JavaScript runtime. Node.js provides scalable network applications and is efficient and lightweight. [109] Moreover, it is flexible and easy to install.

Below, a code sample is presented where we observe the built of a HTTP server. The function `createServer` is used which returns an object and to be more specific a method called `listen`. This method returns the port number that the server will listen. In the example the port number is 1337 and runs locally. The function has two variables `req` and `res` meaning request and response respectively. The `res.writeHead` is the response where 200 is the HTTP status and content type defines the HTTP response header. The `res.end` concerns the response which in this case it writes “Hello World” and changes the line. [113] In Table 2.6.6.1, a simple example is presented of how to create and run a node.js server.

```javascript
var http = require('http');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World
');
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

Table 2.6.1.1: Hello World example (See Ref. [114])
The above script is written in server.js file. We run the server by writing at the node.js command prompt > node server.js. It will show at the command prompt > Server running at http://127.0.0.1:1337/ Then, we open a web page at 127.0.0.1:1337 and we will see written on the web page “Hello World”.

2.6.2 Socket.io

Socket.io is an open-source real-time engine that it is used to provide real-time bidirectional event-based communication. It can be used in real-time analytics, instant messaging and chat, binary streaming and document collaboration. The main characteristics of socket.io are fast and reliable. [115] Socket.io is a powerful tool and supports WebSockets, XHR or JSONP connection. It is easy to use and write code in both sides’ client and server. Node.js is used in order to create and develop the server side of socket.io [116].

In Tables 2.6.2.2 and 2.6.2.3 there are code examples using socket.io with node.js. [117] In server side, we define the http server and socket.io and that port 80 will be used for this application. Inside handler function we define the web page that we want to load. In addition, io.on(‘connection’, function (socket){}) opens a socket connection and socket.emit broadcasts the data to all the users that are connected at that time. Socket.emit provides the opportunity to emit custom events.

On client side, two scripts are presented. The first script we define socket.io library and in the second script we define the address which in this case is run locally. The function socket.on is used in order to open the socket connection and socket.emit is used to emit data via the server. Where is written ‘My other event’, we can define an event.

Server side

```javascript
var app = require('http').createServer(handler);
var io = require('socket.io')(app);
var fs = require('fs');

app.listen(80);

function handler (req, res) {
    fs.readFile(__dirname + '/index.html',
        function (err, data) {
            ```
if (err) {
    res.writeHead(500);
    return res.end('Error loading index.html');
}

res.writeHead(200);
res.end(data);
});
}

io.on('connection', function (socket) {
    socket.emit('news', { hello: 'world' });
    socket.on('my other event', function (data) {
        console.log(data);
    });
});

Table 2.6.2.2: Using Node.js and socket.io – Server Side (See Ref. [117])

Client side

<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io('http://localhost');
    socket.on('news', function (data) {
        console.log(data);
        socket.emit('my other event', { my: 'data' });
    });
</script>

Table 2.6.2.3: Using Node.js and socket.io – Client Side (See Ref. [117])
CHAPTER 3: BENCHMARKING WEBSOCKET & WEBRTC CONCERNING DATA TRANSFER

After conducting a study concerning WebRTC and WebSockets, a benchmark has been set up in order to underline the differences between WebRTC and WebSockets as it concerns communication performance. That is, we wanted to measure the end-to-end latency that is needed in order to transfer data from browser to browser for WebRTC and from client to server and server to client for WebSockets.

The time is calculated in milliseconds. Specifically, in order to compare the WebRTC and WebSockets technologies, we tried to take as many measurements as possible so as to be able to have safe conclusions concerning the average latency that is needed for the transfer of specific data. In this chapter, we present the code that we used in each experiment and our results.

According to the standards we reviewed in Chapter 2, WebSocket provides a bi-directional communication between a client and a server using a single socket over a TCP connection. On the other hand, WebRTC supports browser-to-browser communication using peer-to-peer connection. Our goal is to measure how long it takes to send augmented string messages from one client to another using the above technologies separately. To this end, we calculate the data size (in bytes) of each message that is send and received by the client and the corresponding transfer time in ms (milliseconds).

In order to conduct this experiment, equipment was used from the Multimedia Content Laboratory (MCLab) of TEI of Crete. Specifically, we used three desktop computers with the following characteristics:

- Windows 7 Professional N
- Processor: Intel® Core™ i3-3220 CPU @ 3.30GHz
- RAM: 4.00GB
- System type: 64-bit Operating System
- Browser that was used: Google Chrome version 38.0.2125.111m

In addition, we used node.js as web server [109] in our benchmark. We chose this technology because it is well-known and has the characteristics that we want in order to implement our benchmark. Moreover, we used websocket [35] Client & Server Library which implements the WebSocket Protocol as specified in RFC6455 and finally we used DataChannel.js [118] library for implementing WebRTC code. We should notice that the library that we used in order to develop our WebSocket benchmark has a default value for
maximum received frame size that equals to 64KB = 65056 bytes. If a client tries to transfer more than 65056 bytes then the websocket connection automatically closes and no data go through the server. Specifically, in our tests, websocket handles messages up to 45056 bytes (D=11*2^{12}) and WebRTC handles messages up to 2,883,584 bytes (D=11*2^{18} bytes).

For our benchmark, we used a particular string which is “Hello World”. Having a stable string it was easier to augment it in order to measure how long it takes to send the same data size using different technology.

We ran several tests and took measurements over the time so as to calculate the mean value of data size and to record the time it was required for their transfer. We first ran the benchmark using WebSockets and measured the latency for all communication legs: client to server data transfer, data processing at server side, and server to client data transfer. Then we ran the benchmark using WebRTC technology and measured the time that was needed for the data to be transferred from one client to another.

We used JavaScript language in order to program both WebSocket and WebRTC communication for our benchmark. In the code, we use a JavaScript function where we formulate the requests and the responses and calculate the end-to-end latency and the data size for each message. Every 1000 ms a new string type message is being sent, which then is augmented (doubled) by multiplying it by 2 before resubmit it. As a result, the data size, D, in bytes goes like this: D=11*2^{k-1}, k=1, 2... 18 (with maximum D to equal to 2.883.584 bytes).
3.1 WebSockets Benchmark Code Structure

In Figure 3.1.1, we describe the code for the WebSocket benchmark that we implemented. Both clients 1 and 2 proceed with websocket handshaking with the server. Once each handshake is accepted, the connection is established. The socket opens and is ready to accept or to send messages. In order to measure the data size of our messages and record the time that it is needed to send a message from Client 1 to Client 2 via the server; we created a loop in our code having data size and time as main variables. In the loop, every 1000 ms Client 1 sends a specific string message which is “Hello World”. The message is doubled in every new message that is sent. Meaning that the first message will be “Hello World”, the 2nd message will be “Hello WorldHello World”, the 3rd message will be “Hello WorldHello World Hello WorldHello World” and goes on. The loop continues until the browser cannot handle the data size of the messages due to invalid string length. Then, protocol connection closes and ultimately the WebSocket connection closes.

The procedure to run the benchmark is: Once Client 1 and 2 open the benchmark webpage then Client 1 clicks on the “Send” button and messages begin to transfer every 1000 ms to Client 2. The “Stop” button is used when we see that the browser cannot transfer more data.
Figure 3.1.1: WebSocket Benchmark Code Structure
3.2 WebRTC Benchmark Code Structure

In Figure 3.2.2, we present the code for the WebRTC benchmark that we implemented. In order to implement our benchmark we used the DataChannel.js library. Once the connection is established, similar to the process we followed at the WebSocket Benchmark, a loop is created in our code where every 1000 ms; a string message is sent to Client 2. Specifically, every 1000 ms a message “Hello World” is sent, which is doubled before resubmit it. It means that the first message would be “Hello World”, the 2nd message would be “Hello WorldHello World”, the 3rd message would be “Hello WorldHello World Hello WorldHello WorldHello World” and goes on. The loop continues until the browser cannot handle the data size of the messages. Finally, the connection closes.

The procedure to run the benchmark is: The “Connection” button establishes the connection between the two clients. Then the “Send” button is clicked by Client 1 and messages begin to send every 1000 ms to client 2. The “Stop” button is used when we observe that the browser cannot transfer more data.

Figure 3.2.2: WebRTC Benchmark Code Structure
3.3 WebSockets Measurements

We will begin by presenting our measurements that concern WebSocket technology. For our tests, we used node.js version v0.10.28 and WebSocket Library version 1.0.16.

In Figure 3.3.1, we present the average time that is needed to transfer messages of various data sizes from client to server and from server to client. We observe that the average time is estimated to be around 366 milliseconds for the client – server side (upload). On the other hand, the average time for the server-client side (download) is 137 milliseconds. We see that the upload time is approximately 2.7 times longer than the download time. This notice is in line with the fact that downlinks are always faster than uplinks. We also notice that transfer time at both directions is almost independent of data size. This characteristic of websockets can be especially valuable when large messages need to be transferred between game peers. However, this might be due to the fact that in our implementation the websocket channel is able to handle only messages of one frame long. For messages that required their segmentation into more than one websocket frames, the websocket session was automatically terminated. Hence, for messages larger than the maximum frame size (that is 64KB for us) might the latter notice not be valid. Finally, Figure 3.3.1 illustrates the total end-to-end time for data transmission, that is the sum of client-server and server-client times.

![Websocket: Client-Server, Server-Client & Client-to-Client Data Size in relation to Time](image)

Figure 3.3.1: Compare average time in relation to Data Size between Client-Server & Server-Client
In Figure 3.3.2, we illustrate the average end-to-end (e2e) time between two clients in relation to data size. We see that the larger data size that is transferred is 45056 bytes and it needs 514 milliseconds. The average time for smaller data sizes to be delivered is approximately 502 milliseconds. We also notice, here, that e2e time is almost independent of the data size.

Figure 3.3.2: Client-to-Client data transfer versus e2e time
In Figure 3.3.3, the average end-to-end time is segmented into the three legs of a websocket communication between two clients that is time for the client1-server transfer, time for server processing and time for the server-client2 transfer. The second time includes the time it takes for the server to read data from the one websocket, to find the correct output websocket and finally to push data to the latter. We notice again that the upload time is longer than the download time and we see that the server processes the data almost directly to the client.

![WebSocket Chart]

**Figure 3.3.3:** Client-server, server-client & server processing
3.4 WebRTC Measurements

In Figure 3.4.1, we illustrate the WebRTC measurements. Specifically, we see that the average time increases linearly in relation to the increase of data size. We can see that WebRTC can handle amounts of data up to 2,883,584 bytes (D=11*2^{18} bytes). We should mention that the negotiation time for establishing the WebRTC session is not included in the end-to-end transfer time.

![WebRTC: E2E transfer time in relation to Data size](image)

Figure 3.4.1: E2E transfer time in WebRTC
Figure 3.4.2, shows how much time it takes for the negotiation phase and the data transfer phase to be completed in WebRTC. For the negotiation phase, it takes about 32 milliseconds to be completed. The average time to send and receive all the messages (meaning up to 2.883.584 bytes) is approximately 2642 milliseconds.

Figure 3.4.2: Calculations for the Negotiation time and Data Transfer in WebRTC
3.5 WebRTC vs WebSockets

In Figure 3.4.3, we see the difference between WebRTC and WebSockets concerning the time that is needed to transfer string type data of the same size between two clients. In this figure, the maximum data size that is transferred is 45056 bytes ($D=11*2^{12}$ bytes), which corresponds to the websockets limit. We observe that WebSocket graphic is independent of data size, while the WebRTC graphic is increasing. For the last message, where 45056 bytes are transferred, WebRTC needs 345 milliseconds in comparison to WebSockets that transfers the same message in 514 milliseconds. Hence, we can observe from the figure that WebRTC is faster than WebSockets for all message sizes.

![WebRTC vs WebSockets](image.png)

Figure 3.4.3: Transfer time vs data size: WebRTC vs WebSockets
3.6 Conclusions

We implemented a benchmark so as to compare the performance of two technologies: WebSockets and WebRTC, as it concerns their ability to transfer data. We should mention that WebRTC is based on browser-to-browser communication and WebSockets is based on client-server communication.

In Figure 3.3.1, we observed the average time that is needed to transfer string messages from a client to a server (upload) and from a server to a client (download). We see that the upload time is approximately 366 milliseconds and for the download time is 137 milliseconds. In Figure 3.3.2, we presented the average time that is needed to send a message from the one client to another via the server. The time ranges between 499 to 514 milliseconds to deliver each string message. In Figure 3.3.3, we presented a complete picture of the websocket session where 72.75% of the time belongs to the client-server part (upload time) and 27% belongs to server-client part (download time). The server processing is almost 0.25% of the total time due to the fact that the data transmission via the server takes just 0.25 milliseconds.

On the other hand, in Figure 3.4.1, we presented the WebRTC results. Specifically, the messages are delivered from the one client to the other transferring up to 2,883,584 bytes (D=11*2^{18} bytes). We observe that the graphic is linear. In Figure 3.4.2, we presented the average time for the negotiation phase from all the tests that were conducted. In addition, we presented the average time from all the tests concerning the data transfers. In Figure 3.4.3, we compare the results between WebRTC and WebSockets for the same amount of data.

In conclusion, having conducted the above benchmark, we resulted that WebRTC is a good choice for our implementation.
CHAPTER 4: IMPLEMENTATION

In this chapter, we describe the implementation of a demo 3D collaborative environment online game for two users. We developed a web-based communication environment using WebRTC technology for Web 3D gaming purposes. We managed to bridge two of the latest technologies from the fields of Multimedia & Graphics: WebRTC over X3DOM. WebRTC is based on real-time communication using peer-to-peer connection over graphics. Our goal was to implement a web-based application on which we could transfer both X3D data objects as well as string data using WebRTC technology.

Our application consists of a 3D collaborative environment, a video & audio chat application, a chat application to deliver text messages between two users and the opportunity to add 3D objects from the menu onto the virtual world. Specifically, in the latter case, each user can choose a house from the menu and place it in the virtual world. Both users can see each other actions.

In order to develop our application we use the following tools: Apache server for the graphics, node.js as our web server and socket.io as part of Signaling. The code is written in HTML5 and JavaScript. Ultimately, for compatibility reasons, we used Google Chrome browser to run the application.

Our application can be described as shown in Figure 4.1. Our implementation is divided in two parts: client side and server side. In client side we have an .xhtml file called client.xhtml and support all 3 APIs of WebRTC which are MediaStream API, RTCPeerConnection API and RTCDataChannel API. Additionally, we used socket.io for Signaling and X3DOM methods so as to handle 3D objects. For the server side we have a JavaScript file called server.js which is developed by node.js. Ultimately, the outcome is a demo 3D collaborative environment online game which supports video, audio and text chat application and a 3D virtual world where users not only can insert 3D objects and change the position of them but also recognizing each other actions.
Figure 4.1: Pictorial view of Implementation
In order to implement our application, we followed the following steps as shown in Figure 4.2 and described in the following.

Create the Server

In order to load the client.xhtml page we need to create our server. So, first of all, we established the communication part of our application. In order to do that, we used node.js. Once we have installed node.js on our computer we are ready to use it. We create server.js file so as to develop the server side of our application.

Node.js is used to create an HTTP server and to request handlers. We used createServer function in order to return an object; this has a method named “listen” which indicates the port number where our HTTP server will listen to. Socket.io is used for the Signaling part of WebRTC so as to connect the two web users and transfer data and 3D objects. In order to run the server, we use the command line and we write

> node server.js

In Table 4.1, we present part of the code that is written is server.js file.

```javascript
var server = require('http').createServer(handler);
var app = server.listen(8080);
var io = require('socket.io').listen(app)
, fs = require('fs');

function handler (req, res)
```

In Figure 4.2: Implementation Steps

- Create server using node.js
- Load X3D world
- Video & Audio chat
- Text Messages
- X3D objects (e.g. 3D houses)
Load the Virtual World

The second step is to load the virtual world. For our application, as a test – bed we are going to use the virtual 3D world called Educational Virtual Environment for Mathematics (EViE-m) platform [10] as shown in Figure 6 so as to present web communication based on WebRTC technology over X3DOM.

By using EViE-m virtual world we have the opportunity to develop and present functionalities of the above technologies such as:

- Create chat between two users while playing the game
- To be able to manipulate 3D objects by adding 3D houses and being able to move them inside the virtual world.

We used the latest versions of X3DOM: x3dom.css and x3dom-full.js as shown in Table 4.2. Moreover, we use socket.io library as shown in Table 4.3.

```javascript
fs.readFile('client.xhtml', function (err, data) {
    if (err) {
        res.writeHead(500);
        return res.end('Error loading client.xhtml');
    }
    res.writeHead(200);
    res.end(data);
});

io.sockets.on('connection', function (socket) {
    ........ (Commands)........
});
```

Table 4.1: CODE – server.js

Table 4.2: X3DOM libraries
Table 4.3: Socket.io library

In Figure 4.3 we present EViE-m virtual world.

![Figure 4.3: EViE-m virtual world](image)

**Video & Audio Chat application**

The third step was to develop video and audio chat using WebRTC technology. In order to do that, we used MediaStream API and RTCPeerConnection API. As we mentioned in Chapter 2, MediaStream API is used to obtain audio and video stream using getUserMedia() method. Additionally, we use RTCPeerConnection API to establish peer-to-peer connection, session management and session negotiation so as to have efficient communication between peers.

In order to establish the connection, JSEP (JavaScript Session Establishment Protocol) is used. JSEP is an offer/answer architecture that entails the swap between peers of offer and answer. As a result, offers and answers use Session Description Protocol format.
(SDP) [58] for their communication and go through socket connection. RTCPeerConnection handles equably the communication of audio and video streaming between peers.

In Figure 4.4, we see the video & audio chat application. We have four buttons. When Start Video button is clicked, the browser will automatically ask if the user allows the activation of web camera and microphone. If the user selects “Allow” then the web camera will be activated and the video stream will begin. When both users have activated their web cameras then one of them must press the Connection button. Then, each user will see each other and the video chat call will be established. If the users want to end up their conversation, End Video Call will be used in this case. If the user wants to stop the video stream, the Stop Video button must be pressed.

![Video & Audio Chat Application](image)

**Figure 4.4: Video & Audio Call Application**

**CODE for video & audio call application**

We used navigator.getUserMedia() as we described in Chapter 2 in order to develop our video & audio call application. Below, we present the code when start video button is clicked. In this function, we first make sure to be compatible with several browsers. Then we define the three parameters which are: the constraints meaning video=true and audio=true, the successCallback where in this function we define our local video source and permit the video to play, and errorCallback function which will notify us if any error occurs with the video transmission.
In Table 4.4 we see `startVideo()` function. We define the parameters that are needed, we check for the compatibility of the browsers and we define the parameters of `navigator.getUserMedia()` as we referred above.

```javascript
var localStream = null;
var sourcevid = document.getElementById('webrtc-sourcevid');
sourcevid.autoplay = true;

function startVideo()
{
    navigator.getUserMedia = navigator.getUserMedia || navigator.webkitGetUserMedia ||
    window.navigator.mozGetUserMedia || navigator.msGetUserMedia;
    navigator.getUserMedia({video: true, audio: true}, successCallback, errorCallback);
    function successCallback(stream)
    {
        //……Commands………
        localStream = stream;
        sourcevid.src = window.URL.createObjectURL(stream);
        sourcevid.play();
        // ……Commands………
    }
    function errorCallback(error)
    {
        console.log('The error is: ' + error);
    }
}
```

Table 4.4: CODE - Start Video() function

**Chat application exchanging text messages**

The fourth step was to develop a chat application exchanging text messages. In order to implement this application we used DataChannel.js library [118]. As we mentioned in Chapter 2, RTCDataChannel API is used to exchange arbitrary data between peers. In Figure 8, we see the part where the users can exchange text messages. One of the two users
presses “Connect” button and RTCDDataChannel API is established between the two peers. Additionally, a user id is given for each user so as to be identified each time a message is sent by each one of them. After the connection is established, the users can exchange messages to each other. In Figure 4.5 we introduce the fields of the text chat application that we used for exchanging text messages.

![Figure 4.5: Chat application exchanging text messages](image)

We open a new channel using the following command

- var channel = new DataChannel();

Our application is developed for two peers. We can define the number of users (data session connection) [118] that our application can handle by using the following command

- channel.direction = 'one-to-one';

The commands that we have used in DataChannel library are shown in the following Table 4.5:

<table>
<thead>
<tr>
<th>Commands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel.onopen</td>
<td>Open channel</td>
</tr>
<tr>
<td>channel.onmessage</td>
<td>Receive message</td>
</tr>
<tr>
<td>channel.onleave</td>
<td>Leave channel</td>
</tr>
<tr>
<td>channel.connect()</td>
<td>Search existing channel</td>
</tr>
</tbody>
</table>

Table 4.5: Commands used in DataChannel library
X3D objects

The fifth step is to combine the two technologies: WebRTC and X3DOM. X3DOM provides the opportunity to use and manipulate X3D objects in a virtual world. Using WebRTC we manage not only to transfer generic data but also to transfer 3D objects from one peer to the other peer in real time.

In our application, each user has the opportunity to select a house from the menu, insert and have the right to move the house in the X3D scene. Specifically, when a house is selected and placed in an X3D scene, then both users have the right to move the house at any point during the game. In order to select and move a house in the X3D scene we supply each house with an id and by its translation. In Table 4.6, we present the code that we wrote so as to define the id and the position of each house.

```javascript
socket.on("id", function (data) {
  id = data.home_id;
});
socket.on("trans", function (data) {
  trans = data.home_trans;

  if((id != "") && (trans != "")) {
    document.getElementById(id).setAttribute("translation", trans);
    id = "";
    trans = "";
  }
});
```

Table 4.6: Defining id and translation for each house

Inside the socket, we describe the house and provide the opportunity for both users to see the house simultaneously in the virtual world. In order to use x3d files, we use the inline method from X3DOM documentation. According to [119], inline is a grouping node and provides the opportunity using url to load nodes from another X3D scene.

Moveable method is used from x3dom uiEvents html example [120] and gives the opportunity to move the house inside the X3D scene. This is provided from X3DOM documentation. [9]
In Table 4.7, we present the code for selecting “houseA”. A channel is opened so as to transfer the data that has to do with house A. The same structure code is used for HouseB and HouseC.

```javascript
var homeA=document.getElementById("houseA");
sendChannel.send(homeA);

if(data.home == "A")
{
    var transform = document.createElement('Transform');
    transform.setAttribute('id', 'homeA');
    transform.setAttribute('translation', '-6 0 0');
    transform.setAttribute('rotation', '0 0 1 0');
    var inline = document.createElement('Inline');
    inline.setAttribute('url', 'houseA.x3d');
    transform.appendChild(inline);
    document.getElementsByTagName('Scene')[0].appendChild(transform);
}
```

Table 4.7: Code for: Channel opened and HouseA information

In order to locate the house in the X3D scene and have the opportunity to move it in the scene we use the following code as shown in Table 4.8.

```javascript
homeA.onclick = function()
{
    var transform = document.createElement('Transform');
    transform.setAttribute('id', 'homeA');
    transform.setAttribute('translation', '-6 0 0');
    transform.setAttribute('rotation', '0 0 1 0');
    var inline = document.createElement('Inline');
    inline.setAttribute('url', 'houseA.x3d');
    transform.appendChild(inline);
    document.getElementsByTagName('Scene')[0].appendChild(transform);
    var boxes = document.getElementById("boxes");
    runtime = boxes.runtime;
    boxes.addEventListener('mouseup', stop, false);
    var homeA = document.getElementById("homeA");
    homeA.addEventListener('mousedown', start, false);
    new x3dom.Moveable(boxes, homeA, moveCallback, 0);
    socket.emit('getHome', { home: "A" });
}
```

Table 4.8: Code for: When a house is located in the X3D scene and it is moveable
In Figure 4.6, we present a user’s virtual world with all the houses located in different positions.

Figure 4.6: The houses located by the user in the virtual world
In Figure 4.7, we present our implementation as shown with the use of Google Chrome browser.

Figure 4.7: Implementation as shown with the use of Google Chrome Browser
4.1 Running the application

We will present our implementation localhost. At first, we activate Apache server so as to be able to load the X3D scene.

Figure 4.1.1: Activating Apache Server

Continuously, we run our server using node.js command prompt so as to be able to load our client.xhtml page.

Figure 4.1.2: Run node.js server
We open client.xhtml page as we present in Figure 4.1.3. In order to use all the features of WebRTC technology we must first click the “Start” button. If we want to stop, we then select the “Stop” button.

Figure 4.1.3: Open client.xhtml page
If we want to start video chat call then “Start Video” button must be selected as shown in Figure 4.1.4. Once is clicked then a message bar pops up asking if we want to use our microphone and camera. If we select to “Allow” the use of our web camera and microphone, then audio and video will start streaming. If we choose to “Deny” then we will not be in position to use the web camera for video chat.

Figure 4.1.4: Use of video chat
If we want to insert a house in the virtual world, all we have to do is to select one of the three houses that we have in the menu. For instance, we select “HouseA”. Once we choose the house, the house appears in the virtual world. Then, the user can change the position of the house in the virtual world. Additionally, the second user that is connected has the right to move the house. In Figure 4.1.5 we see “HouseA” in the virtual world. In Figure 4.1.6 we see all three houses positioned by the two users in the virtual world. Finally, at all times each user can change the visual point of viewing the virtual world. Each user has the opportunity to watch the virtual world from a different angle. Additionally, the user can zoom in and zoom out.

Figure 4.1.5: Selecting HouseA
In Figure 4.1.7 we see an example of exchanging text messages. At first, both users must click on “Connect” button. A random user id is created for each user so as each user to identify each other. Then the users can start exchanging text messages with each other.

1 user

Me: Hello
5044619: Hello

2 user

5020951: Hello
Me: Hello

Figure 4.1.7: Exchanging text messages
CHAPTER 5: CONCLUSIONS & FUTURE WORK

In this thesis, we conducted a survey on state-of-the-art technologies that can support Web 3D Gaming. Our goal was to research on state-of-the-art technologies, develop a benchmark for WebSocket and WebRTC concerning data size transfer in relation to time and finally to implement a demo 3D collaborative online game introducing WebRTC over X3DOM technology.

From our research, we saw that online gaming was mainly supported by Peer-to-Peer (P2P) and Client-Server architectures. We observe that the recently introduced WebSockets and WebRTC technologies seem to ideally support such architectures.

WebRTC provides browser-to-browser communication based on peer-to-peer connection. It offers real-time connection. Unfortunately, for the time being, WebRTC is not supported by all the current browsers. As a result, in order to use WebRTC, browsers must be compatible with each other, support protocols such as Real-Time Protocol (RTP), audio and video codecs, etc. On the other hand, WebSockets is based on client-to-server connection; its basic characteristic is that all data must go through the server in order to be transmitted from one peer to another.

After having conducted a research on the latest technologies, we completed a benchmark for WebSocket and WebRTC technologies concerning data transfer in relation to time. We observed that WebRTC and WebSocket technology deliver the data quickly and reliably. We were able to conduct several tests so as to have a more complete view of the outcome. In WebRTC we observed that time increases linearly in relation to data size. On the other hand, due to the library that we chose to implement our WebSocket benchmark, the socket stops delivering data over 65056 bytes.

For the implementation, we used the virtual world named EViE-m, providing thus a demo web 3D collaborative environment. With the latter, end users are provided with a rich 3D graphics environment using web browsers without any need for third party software or plugin installation. We combined audio, video, data and 3d objects to be transferred using WebRTC technology.

The tools that we used for the implementation were node.js and socket.io in order to use WebRTC and X3DOM methods and functions.
In conclusion, in the present thesis, we proposed real-time communication using peer-to-peer connection for web 3D gaming. We tried to bridge two fields: Multimedia and Graphics and combine two state-of-the-art technologies: WebRTC over X3DOM.

WebRTC was selected over WebSocket technology for our implementation because WebRTC provides peer-to-peer connection, supports browser-to-browser communication, no server to intervene, can select between reliable and unreliable data messages which means that we can choose between TCP and UDP connections.

In addition, WebRTC is open source, supports real-time communication (RTC), can be used for video, audio and data, plugins are not needed. Moreover, WebRTC provides low latency. RTCDataChannel API is used to transfer not only data but also X3D objects as well. RTCDataChannel API functions resemble WebSocket functions and usage. On the other hand, WebSocket uses TCP connection.

The perspective of WebRTC in Web 3D gaming is tremendous. The opportunity to play in real-time without a server to intervene and absorb all the data and then send to all potential users is a great deal. Moreover, users have the opportunity to interact with each other via chat messaging or using video call in real-time.

According to recent scientific publications, multiplayer online games are mainly based on P2P connection using several architectures. Our study indicates the possibility and the advantages of using WebRTC over X3DOM technology for web 3D gaming.

The field of Web 3D Gaming is developing fast and many WebRTC applications are on their rise. Our thesis work can be enriched and extended to multiple users and mobile applications while incorporating cloud-based technology.

**Future Work**

WebRTC is one of the latest and fastest growing technologies in the field of real-time communication. Being able to connect peers using browser-to-browser communication makes it easier to build new applications and be used in real-time.

This thesis presented a demo 3D collaborative environment implementation developed in WebRTC technology. This demo combines audio, video and data transfer as well as X3D objects in an X3D scene using RTCDataChannel API. In our implementation this is supported for two users. As a result, there are future aspects of our development that we should take into consideration.
First of all, this demo can be developed for multiple users. In this case, it is important to choose the appropriate topology to support a multi-player online game. For instance, Mesh topology is a good consideration.

Furthermore, it can be developed to be supported for mobile devices such as cell phones and tablets. Nowadays, more and more applications and several games are created specifically so as to work on mobile devices. As a result, in our case, mobile browsers must support WebRTC technology. Real-time communication plays a significant role in online gaming. Ultimately, it would be very interesting to see our work to be supported and developed using cloud technology.
REFERENCES


[89] S. Dey, "Cloud Mobile Media: Opportunities, challenges, and directions," in Computing, Networking...
and Communications (ICNC), 2012 International Conference on, 2012.


