Norwegian University of
Science and Technology

# A Flexible Network Interface for a Real-time Simulation Framework


**Lars Ivar Hatledal**



Simulation and Visualization
Submission date: June 2017
Supervisor: Arne Styve, IIR
Co-supervisor: Houxiang Zhang, IHB



Norwegian University of Science and Technology
Department of ICT and Natural Sciences

# Summary

This thesis presents a flexible network interface for the real-time simulation framework, *Vico*. In particular, the interface allows remote clients to connect to simulations over the network using a variety of transport mechanisms. Furthermore, new transports can be added to the system due to the extensible software design. Additionally, individual clients can choose how data payloads are sent/received over the wire. Both the textual JSON format and the binary Google Protocol Buffers format are supported for static message types. Again, this mechanism is extensible such that additional formats can be supported in the future. Allowing multiple transports and encoding schemes ensures that the system is accessible, allowing a wide variety of clients to interact with the simulations. Messages from clients are handled by so called *Message Handlers*. The system comes bundled with a set of default handlers, some of which allows clients to implement real-time 3D visualization and plotting through pre-defined message types. The content and structure of these messages has been developed as part of the thesis. In order to facilitate special requirements, a special handler allows annotated Java methods to be exposed dynamically over the network using JSON RPC 2.0 compliant messages. Additionally, simulation designers are free to add new handlers to fit their needs.

The details of the implementation will be presented, as well as performance metrics for the utilized network transports and serialization formats. To show the usefulness of the system, both a web client and Unity3d client was implemented an tested. In order to test the system as an enabler for multi-display solutions, the Unity3d client was used to project a simulation onto the cylindrical dome in the Visualization lab located at NTNU Aalesund.

# Preface

This is my master's thesis in Simulation and Visualization titled ”A Flexible Network Interface for a Real-time Simulation Framework” as part of the study program Simulation and Visualization at NTNU, carried out during the spring semester of 2017.

I would like to thank my supervisors, Arne Styve and Houxiang Zhang, for their excellent guidance and support during this process.

Ålesund, 13 June 2017, Lars Ivar Hatledal

# Table of Contents

**Summary** i

**Preface** ii

**Table of Contents** v

**List of Tables** vii

**List of Figures** x

**List of Code Listings** xi

**Abbreviations** xii

**1 Introduction** 1
- 1.1 Motivation . . . 2
- 1.2 Research questions . . . 2
- 1.3 Structure of this work . . . 3

**2 Basic Theory** 5
- 2.1 Computer networking . . . 5
  - 2.1.1 Protocols . . . 5
  - 2.1.2 Middleware . . . 6
  - 2.1.3 Brokers . . . 8
- 2.2 Data serialization formats . . . 9
  - 2.2.1 JSON . . . 9
  - 2.2.2 Google Protocol Buffers . . . 9
- 2.3 WebGL . . . 10
- 2.4 CIGI . . . 10
- 2.5 Multi-display systems . . . 11

3 **Literature Review** **13**
3.1 Web-based simulation . . . 13
3.2 Networking in games . . . 15
3.2.1 Peer-to-peer . . . 16
3.2.2 Client/server . . . 16
3.2.3 Client-side predication . . . 16
3.3 Multi-display solutions . . . 17
3.4 Online Virtual Worlds . . . 17
3.4.1 Second Life . . . 17
3.5 Comparisons of middleware solutions . . . 18
3.6 Comparisons of serialization formats . . . 18
3.7 Synchronized wave visualization . . . 19

4 **Materials and Method** **21**
4.1 Gradle . . . 21
4.2 Three.js . . . 21
4.3 Unity3d . . . 22
4.4 Other 3rd party dependencies . . . 22
4.5 Workflow . . . 22
4.6 Benchmark setup . . . 23
4.7 Visualization lab . . . 24

5 **Results** **27**
5.1 Server-side Implementation . . . 27
5.1.1 Framing . . . 27
5.1.2 End-points . . . 28
5.1.3 Message handling . . . 30
5.1.4 Messages for 3D visualization . . . 34
5.1.5 Water . . . 38
5.1.6 Directory service . . . 39
5.2 Client-side implementation . . . 39
5.2.1 Web application . . . 40
5.2.2 Unity3d . . . 41
5.3 Benchmarks . . . 43
5.3.1 Request-reply . . . 44
5.3.2 Publication . . . 45
5.4 Synchronized Wave Visualization . . . 46
5.5 Multi-projector rendering . . . 46

6 **Discussion** **53**
6.1 Wave visualization . . . 53
6.2 Industry standards . . . 53
6.2.1 DDS . . . 53
6.2.2 CIGI . . . 54
6.3 Performance . . . 54
6.3.1 Networking . . . 54

6.3.2 Serialization . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.4 Multi-display rendering . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.5 Miscellaneous . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.5.1 Broker architecture . . . . . . . . . . . . . . . . . . . . . . . . 55
6.5.2 UDP support . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.5.3 Loading 3D models . . . . . . . . . . . . . . . . . . . . . . . . 57
6.5.4 Simulation playback . . . . . . . . . . . . . . . . . . . . . . . 57

**7 Conclusion** **59**

**Bibliography** **61**

**Appendix** **65**

# List of Tables

4.1 Third party software libraries used in this project . . . . . . . . . . . . . . 22
4.2 Specification of the computers used for the benchmark . . . . . . . . . . 24

5.1 UPD header . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Implemented message handlers . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3 Message requesting a simulation update. The payload is encoded using JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.4 Simulation setup response from the *Requesthandler* . . . . . . . . . . . . 33
5.5 Measured throughput . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.6 Mean time to parse 1000 messages . . . . . . . . . . . . . . . . . . . . . . 45

# List of Figures


1.1 Vico core architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Overview of this work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.1 Example ZeroMQ framing . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Broker as Directory Service (ZeroMQ, 2012) . . . . . . . . . . . . . . . . 8
2.3 CIGI (SISO, 2012) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 Multi-projector System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 Local WBS (Byrne et al., 2010) . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 Remote WBS (Byrne et al., 2010) . . . . . . . . . . . . . . . . . . . . . . 14
3.3 Hybrid WBS (Byrne et al., 2010) . . . . . . . . . . . . . . . . . . . . . . . 15
3.4 Screenshot from the 2013 title Battefield 4 showing a player driving a Jet Ski on a dynamic water surface. . . . . . . . . . . . . . . . . . . . . . . 19

4.1 Typical Git workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Benchmark physical configuration. Connectivity is restricted to the local area network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.3 View of the visualization lab, which features 12 projectors that projects onto a large cylindrical wall . . . . . . . . . . . . . . . . . . . . . . . . . 25

5.1 Message framing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.2 Message routing. Adapted from (Buschmann et al., 1999) . . . . . . . . . 30
5.3 Message Translator. Adapted from (Buschmann et al., 1999) . . . . . . . 32
5.4 Example of rendering geometry for visual and collision . . . . . . . . . . 36
5.5 A collection of supported geometries: Mesh, Line, Height-field, Plane, Cylinder, Capusle, Sphere and Box . . . . . . . . . . . . . . . . . . . . . . 36
5.6 Point cloud rendering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.7 Example of a height-field as textured terrain, rendered in the browser client 37
5.8 Drum simulation featuring the curve type for representing a wire . . . . . 38
5.9 Wave synchronization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.10 Directory Service GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

5.11 Selection page for simulations in the browser . . . . . . . . . . . . . . . . 40
5.12 Render settings in the browser . . . . . . . . . . . . . . . . . . . . . . . . 41
5.13 Plotting in the browser . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.14 New Unity3d menu items. *Vico fetch* will initiate a query for available simulations, which will populate the *Load Vico Simulation* menu . . . . . 42
5.15 Curve implementation in Unity3D . . . . . . . . . . . . . . . . . . . . . . 43
5.16 Simulation scene rendered in different implementations . . . . . . . . . . . 44
5.20 Waves synchronized between different render implementations . . . . . . 46
5.21 The test setup. Two clients were used, each powering a projector. One of the clients was assigned the master role, publishing it's camera transform to the second client acting as a slave. The simulation was running on a remote computer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.22 Scene rendered using two projectors in the Visualization lab. Warping and blending effects were achieved using master student Rolf-Magnus Hjørungdal's Unity3d asset. . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.17 100 request-reply using binary encoding . . . . . . . . . . . . . . . . . . . 49
5.18 100 request-reply using textual encoding . . . . . . . . . . . . . . . . . . . 50
5.19 Time used to complete 100 request-reply for the various transport - both using GPB and JSON encoding of the payload data . . . . . . . . . . . . 51

6.1 Design alternatives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.2 Example of simulation scene with multiple 3D models . . . . . . . . . . . 57

# Listings


2.1 Example .json file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Example JSON RPC call . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Example .proto file . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
5.1 VicoServer.java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.2 WritableConnection.java . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.3 UDP send implementation . . . . . . . . . . . . . . . . . . . . . . . . 29
5.4 MessageHandler.java . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.5 VicoService.java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.6 SimulationMsg.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.7 EntityMsg.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.8 CurveMsg.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.9 ConnectMsg.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
7.1 CommonProto.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
7.2 RequestProto.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
7.3 SubscriptionProto.proto . . . . . . . . . . . . . . . . . . . . . . . . . 68
7.4 PlotProto.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
7.5 SpawnProto.proto . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.6 Master.cs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
7.7 Slave.cs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
7.8 STLLoader.cs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
7.9 RemoteOBJLoader.cs . . . . . . . . . . . . . . . . . . . . . . . . . . 74
7.10 AbstractClient.cs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
7.11 Socket.js . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
7.12 VicoFrame.java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.13 VicoMsg.java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.14 RemoteManager.java . . . . . . . . . . . . . . . . . . . . . . . . . . 84

# Abbreviations

| | | |
|---|---|---|
| WBS | = | Web Based Simulation |
| CIGI | = | Common Image Generator Interface |
| DDS | = | Data Distribution Service |
| HTTP | = | Hypertext Transfer Protocol |
| IP | = | Internet Protocol |
| TCP | = | Transmission Control Protocol |
| UDP | = | Universal Datagram Protocol |
| JSON | = | JavaScript Object Notation |
| GPB | = | Google Protocol Buffers |
| XML | = | eXtensible Markup Language |
| FMI | = | Functional Mock-up Interface |
| API | = | Application Programming Interface |
| MB | = | Megabyte |
| NTNU | = | Norwegian University of Science and Technology |
| GUI | = | Graphical User Interface |
| OBJ | = | Wavefront Object File |
| STL | = | STereoLithography |
| GLTF | = | GL Transmission Format |
| VR | = | Virtual Reality |
| HMD | = | Head Mounted Display |
| FOV | = | Field of View |
| IG | = | Image Generator |
| RPC | = | Remote Procedure Call |
| GUID/UUID | = | Globally/Universally Unique Identifier |
| VCS | = | Version Control System |
| IDE | = | Integrated Development Environment |
| DSL | = | Domain Specific Language |
| IDL | = | Interface Definition Language |
| HTML | = | Hypertext Markup Language |
| P2P | = | Peer-to-peer |
| IoT | = | Internet of Things |

Chapter 1

# Introduction

Vico is an in-house simulation framework written in Java targeting real-time simulations of cyber-mechanical systems in the time-domain. Vico started out as a virtual prototyping tool for cranes, but evolved into a general purpose simulation system. Simulations are written in Java, but can make use of models written in other tools/languages using the FMI (Blochwitz et al., 2012) standard. Vico's core software architecture is based around the Entity-Component model. A simulation is built up of *Entities*, which are merely placeholders for *Components*. A view of the architecture is given in Figure. 1.1. An Entity cannot be extended, thus logic and data can only be introduced through creating and assigning Components to them. This approach favours composition over inheritance, and is very powerful as the the characteristics of an Entity can be transformed during simulation by adding and/or removing components.

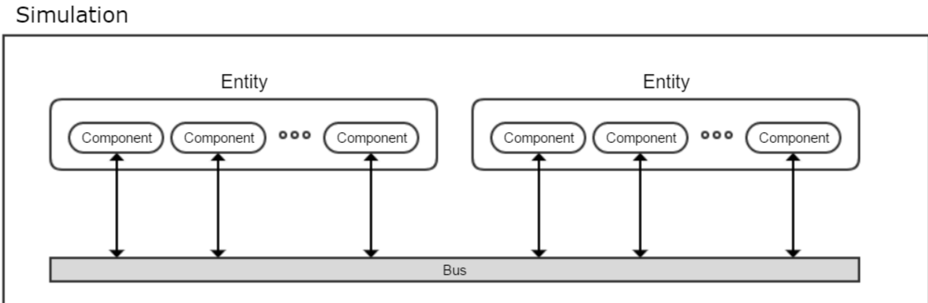



**Figure 1.1:** Vico core architecture

## 1.1 Motivation

Traditional software applications are installed on a local computer, and in order for it to work as expected, the computer must meet the requirements of the software in terms of operating system, dependencies and hardware. However, if the application provided a network interface, the application could run on a centralized server. This would allow clients implemented in any languages, running on any platform to take advantages of the services provided. Using this approach, several benefits can be identified:

- For web clients, any software updates are automatically available.
- For desktop clients, server side updates which do not alter the communication stream are automatically available.
- Simulation and visualization can be completely de-coupled (running on different computers). Allowing simulations to run on dedicated server software, while still allowing them to be visualized (on another platform).
- With the server responsible for running the simulation code, the computing hardware requirements for the clients are decreased. Allowing even mobile devices to view/interact with complex simulations.
- With the clients responsible for running front-end code such as 3D visualization, the graphical hardware requirement for the server is decreased.
- Virtual collaboration can be facilitated (multiple users can interact with the same simulation simultaneously).
- Allows for distributed rendering.
- Open interface allows for others to implement their own presentation solution.
- Services can be accessed from any language/tool with socket capabilities.

The main idea for this work is to improve the current networking capabilities of the simulation framework Vico. The results of the work should allow other applications, both locally and remotely, to efficiently interface against running simulations. The interface could be used to control the simulations or visualize data both in 3D and 2D. An important motivation is to make the solution flexible, in terms of using technologies that enables a variety of 3rd party tools to use it. An overview of the system and how this works fits into the picture is given in Figure. 1.2.

The Vico framework has several prerequisites that must be met by the local computer in order for it to run. Among other things it must run on Windows, have a powerful CPU and requires several applications to be present on the system. However, by accessing Vico through a server, none of these requirements applies to the client, allowing even mobile devices to interact with complex real-time simulations.

## 1.2 Research questions

Research questions to be raised and answered in this thesis are:

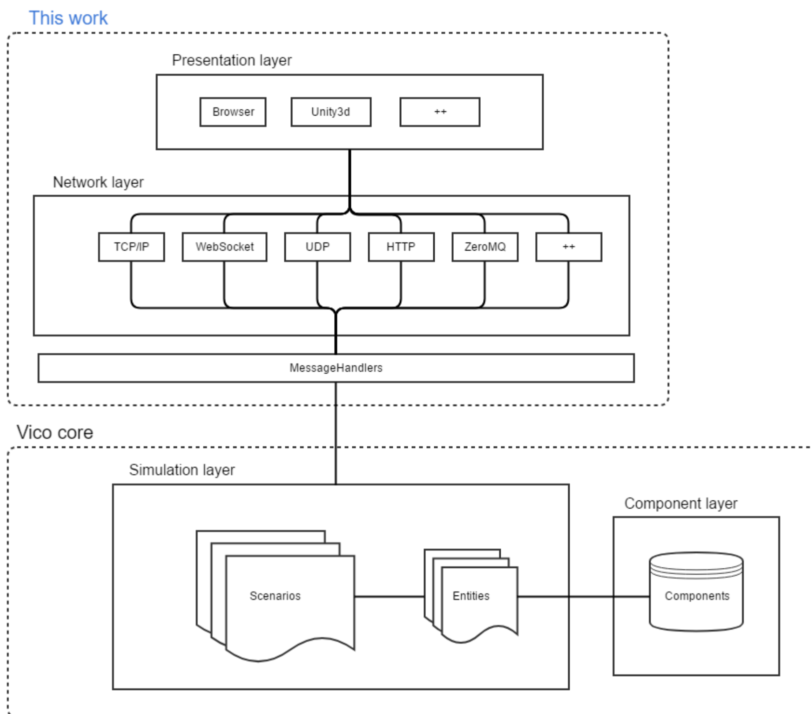



**Figure 1.2:** Overview of this work

- Performance of transport protocols. How do they compare?
- Textual vs. binary data encoding, what are the implications?
- Performance of web vs. networked desktop apps. How much has web applications matured?
- How to solve issues related to synchronization and visualization of dynamic water surfaces?
- What standards are beneficial? Have they been commonly adopted?

## 1.3 Structure of this work

This thesis presents a flexible network interface for the simulation framework Vico. The interface can be used to interact with (e.g. modify certain aspects of a simulation) and visualize data (e.g. 2D plots and 3D scenes). The implemented system is flexible, allowing multiple protocols to communicate with the host. Furthermore, both textual and binary representation of message payloads can be used. The usefulness is demonstrated through the implementation of two different clients.

The thesis is organized as follows. Firstly, some basic theory is given, allowing the reader to better understand some underlying concepts. Subsequently, a review of the related research work is given, followed by a description of the material and methods used. What follows are the results, which will contain implementation details as well as a set of case studies and is followed by a chapter designated for discussions. Finally, a conclusion is given.

The appendix contains a few highlighted snippets of source-code. Most notably, the GBP schemas, but also some Java, C# and JavaScript code. Attached in the provided zip file is all the client-side code for the Web and Unity3d projects, all the GPB schemas and relevant server-side Java code. All of Vico is not included. As such, the code is not executable, but should provide an understanding of the implemented system regardless.

# Chapter 2

# Basic Theory

This chapter introduces the reader to some basic concepts and technologies required to better understand the concepts used and discussed later.

## 2.1 Computer networking

### 2.1.1 Protocols

A network protocol defines rules and conventions for communication between network devices. Below is an introduction to the protocols relevant for this project.

#### TCP/IP

TCP/IP (Forouzan, 2002) is a two layer program. The higher level layer, Transmission Control Protocol (TCP), manages transmission of data by assembling messages into smaller packets, which is reassembled on the receiving end after beeing sent over the Internet. The lower layer, Internet Protocol (IP) handles the address part of each packet so that it get to the right destination. TCP is the most commonly used protocol on the Internet. A reason for that is that it is reliable. Packets sent over the Internet may get lost due to a number of reasons, and when that occours the TCP client will re-request the package from the server until the whole packet is complete.

#### UDP

The User Datagram Protocol (UDP) is a protocol for sending messages to other hosts on an Internet Protocol (IP) network. It is formally defined in RFC 768 (Postel, 1980). UDP uses a simple connection-less transmission model, thus no handshaking is involved. UDP is faster than TCP because there is no form of flow control or error correction, thus messages are not guaranteed to be delivered and no assumptions of the ordering of the packets can be made. A single message sent through UDP can be no larger than 65,507 bytes (Forouzan,

2002). Larger messages must be divided into smaller packets and re-assembled in the receiving application. UDP is suitable when speed is preferred over reliability. In some real-time systems, given that the messages are non-critical, it may be preferable to drop packages rather than waiting for them.

#### HTTP

The Hypertext Transfer Protocol (HTTP) is the foundation of data communication for the World Wide Web. Communication between a client and server using HTTP is always initiated by the client in a request-response pattern. HTTP utilizes TCP/IP internally. Unlike ”raw” TCP/IP connections, which are session oriented, HTTP connections are always terminated once a request (initiated by the client) has been completed. This trait makes it impossible for the server to push messages to clients. Before the introduction of WebSockets along with HTML5, HTTP was the only means a web page had for communication.

#### WebSocket

A WebSocket is, as the name suggests, designed to be implemented in web browsers and web servers. However, it can be used by any client or server application. WebSockets were designed to bring real-time bi-directional streams to web browsers. Before the introduction of WebSockets in HTML5, full-duplex transmissions between client and server was not straightforward. However, some methods for real-time data exchange based on HTTP has been available using polling, long-polling and streaming mechanisms. But these methods involve unnecessary HTTP request and response headers, which introduce latency, and the server itself cannot initiate a connection using the standard HTTP model (Loreto et al., 2011). The initial handshake of a WebSocket connection resembles HTTP, allowing servers to handle HTTP and WebSocket connections on the same port.

### 2.1.2 Middleware

A middleware is a distribution infrastructure software that shields applications from many inherent and accidental complexities of operating system and networks (Buschmann et al., 1999).

#### DDS

The Data Distribution Service for real-time systems (DDS) is a standard managed by the Object Management Group (OMG), that aims to enable scalable, real-time, dependable and interoperable data exchanges using a publish-subscribe pattern. The request-reply pattern was not originally apart of the standard, but was recently proposed to be included as an extension (OMG, 2016). DDS interfaces are defined using OMG Interface Definition Language (IDL). Some of the features of DDS are automatic discovery of publishers and subscribers, Quality of Service (QoS) and delay tolerant networking. IoT (Internet of Things) applications are the main consumers of the DDS middleware. Both commercial and open-source implementations of DDS are available. Implementations exists in Ada, C, C++, Java, Scala, Lua, Pharo and Ruby.

### ZeroMQ

ZeroMQ (Hintjens, 2013) is a asynchronous messaging library, aimed at use in distributed or concurrent applications and comes with several built-in messaging patters, with the two most relevant beeing:

- **Request-Reply** which connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- **Pub-sub** which connects a set of publishers to a set of subscribers. This is a data distribution pattern.

The following transport protocols can be used to send messages between ZeroMQ sockets:

- **inproc** local in-process (inter-thread) communication transport.
- **IPC** local inter-process communication transport (UNIX only).
- **TCP** unicast transport using TCP.
- **PGM, EPGM** reliable multi-cast transport using PGM.

The inproc and IPC protocols can only be used for internal messaging between threads, useful for building concurrent applications.

ZeroMQ is not a neutral carrier: it imposes a framing on the transport protocols it uses. This framing is not compatible with existing protocols, which tend to use their own framing. An example is shown in Figure.2.1, where the leading 5 is the size of bytes in the remainder of the frame. A message can have multiple such frames, and each message will begin with the total number of bytes in the remainder of the message.

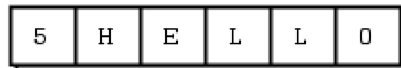



**Figure 2.1:** Example ZeroMQ framing

ZeroMQ comes with the low-level C API. High-level bindings exist in 40+ languages including Python, Java, PHP, Ruby, C, C++, C#, Erlang, Perl, and more. Also native implementations written in Java (JeroMQ) and C# (NetMQ) are available.

### RakNet

RakNet (Oculus, 2014) is a networking middleware designed for games. Originally authored as a commercial product by Jenkins Software LLC. Then bought by Oculus, which again was acquired by Facebook. In 2014 the source code was released to the public.

RakNet is a C++ library that provides UDP and TCP transport. Also RakNet offers built-in re-transmission and reordering of packets, turning UDP into a reliable and faster solution than TCP in networks where there is little loss of packets (Reis et al., 2011).

### 2.1.3 Brokers

Software architectures of most messaging systems is distinctive by the messaging server, *broker*, in the middle (ZeroMQ, 2012), to which applications are connected. No application is speaking directly to the other application, rather all communication passes through it. A broker is responsible for coordinating communication, such as forwarding requests, as well as transmitting results and expectations (Buschmann et al., 1999).

The functionality of the broker can be split into two separate parts:

1. Broker has a repository of applications running on the network. It knows that application *X* runs on host *Y* and that messages intended for *X* should be sent to *Y*. It acts like a directory service.

2. Broker does the message transfer itself.

The main benefit of the broker model is that applications don't have to have any idea about location of other applications. The only address they need is the network address of the broker, which then routes the messages to the right applications based on some criteria, like a simulation id.

#### Broker as directory service

In this setup, the broker is lightweight and is only responsible of keeping a record of which end-points are available. When a new end-point is created it will notify the broker about it's existence. If the end-point terminates it will consequently disappear from the broker's record. Clients can query the broker for available end-points, in which the broker will respond with the necessary information for client to connect directly to the desired end-point. The concept is shown in Figure. 2.2.

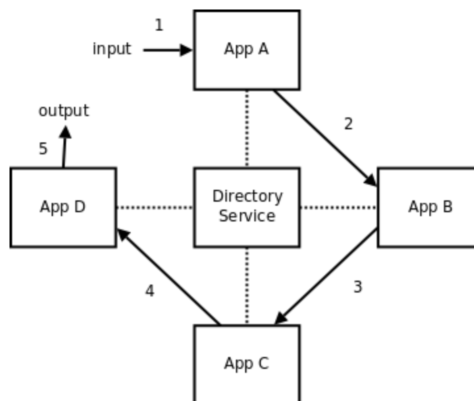



**Figure 2.2:** Broker as Directory Service (ZeroMQ, 2012)

## 2.2 Data serialization formats

Serialization (also known as marshalling) is the process of translating data structures into a format that can be stored or transmitted and reconstructed later. When the resulting series of bits is re-read according to the serialization format (known as deserialization or unmarshalling), it can be used to create a semantically identical clone of the original object. A huge number of such formats exists today, with a varying degree of adaptation. Below is an description of formats relevant to this project.

### 2.2.1 JSON

JavaScript Object Notation (JSON) is a simple text based standard for data transmission. As the name suggests, it was originally derived from JavaScript. However, it is independent of any programming languages. Today, most programming languages has built-in support for parsing/generating JSON. Because of the close relationship with JavaScript, JSON is often the go-to format to use when serializing data in a web application. An example JSON string is given in Listing 2.1.

**Listing 2.1:** Example .json file

```
{
    uuid: "9daa3ff8-22f9-46fb-9095-302749af0797",
    name = "root",
    position: {x: 0, y: 0, z: 0},
    quaternion = {x: 0, y: 0, z: 0, w: 0},
    children: []
}
```

#### JSON RPC

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol (Group et al., 2012). JSON-RPC uses JSON as the data format and is designed to be simple. The newest version of the protocol is 2.0. An example of a JSON-RPC call is given in Listing. 2.2. Here, a method called subtract is given two parameters 23, and 42. The result sent back to the invoker is 19.

**Listing 2.2:** Example JSON RPC call

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}
```

### 2.2.2 Google Protocol Buffers

Protocol Buffers are Google's mechanism for serializing structured data. Official implementations exists in C++, C#, GO, Java and Python, however unofficial ones exist also for other languages such as JavaScript. Compared to common alternatives for data serialization over the wire, such as XML and JSON, protocol buffers generate much smaller data packages because it uses a binary format. Protocol Buffer messages are compiled using a predefined schema. This helps developers to avoid creating erroneous messages as the compilation will fail in the presence of syntax errors. The schema is specified in a file with a *.proto* extension. An example of such a file is given in Listing. 2.3.

**Listing 2.3:** Example .proto file

```
syntax = "proto3"; // specify which version we are using

include "math.proto";  //definitions from other .proto files can be included

message Transform {
    string uuid = 1;
    string name = 2;
    Vector3 position = 3;
    Quaternion quaternion = 4
    repeated Transform children = 5;
}
```

## 2.3 WebGL

WebGL (Web Graphics Library) (Marrin, 2011) is a JavaScript API for rendering interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins. WebGL does so by introducing an API that closely conforms to OpenGL ES 2.0 that can be used in HTML5 *canvas* elements (Mozilla, 2017). Before WebGL, developers had to rely on plug-ins or native applications and ask their users to download and install software in order to deliver a true 3D experience (Parisi, 2012).

## 2.4 CIGI

The Common Image Generator Interface (CIGI) is an interface designed to promote a standard way for a host device to communicate with an image generator (IG) in the simulation industry (Phelps, 2002). CIGI wishes to promote interoperability among image generators, in order to reduce integration costs. The basic idea is shown in Fig. 2.3, where CIGI is shown to be present both on the host (server-side) and IG (client-side). Normally, the CIGI protocol interfaces one host (transmitter) and one IG (receiver), but UDP multi-cast could be used to support multiple receivers. The Host would select one IG as a ”master” and would ignore all IG-to-Host packets except those originating from the master. Both synchronous and asynchronous operation is supported by CIGI. As CIGI is a data packaging protocol it is independent of the transport medium. Any suitable medium such as TCP/IP, UDP or shared memory could be used. Multiple vendors has chosen to support CIGI, with the official web page listing roughly 20 of them. In 2014, CIGI became an international standard through the Simulation Interoperability Standards Organization (SISO).

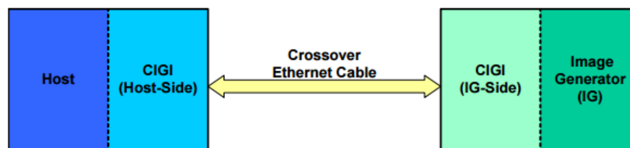



**Figure 2.3:** CIGI (SISO, 2012)

## 2.5 Multi-display systems

Multi-display setups are common in training simulators, where they are used in order to project the simulation onto a larger surface, i.e. a dome. Simple configurations requires only one computer, which must be powerful enough to render to multiple outputs. However, such a solution scales badly and cannot cover large areas. In order to overcome this challenge without sacrificing image resolution, the rendering needs to be distributed among several computers. An example of a multi-projector layout for a dome is shown in Figure. 2.4.

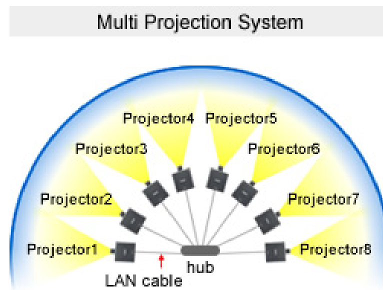



**Figure 2.4:** Multi-projector System

Chapter 3

# Literature Review

This chapter gives an overview of literature related to this work.

## 3.1 Web-based simulation

Web-based simulation (WBS) refers to the use of web technologies to develop, execute, and analyze simulation models where the primary interface is accessed through a web browser (Byrne et al., 2010). Web-based technologies are especially beneficial to non-experts as they can significantly reduce the costs required for a new user to experiment with and learn about a simulation and modeling tool. Increasing capabilities of the latest web browsers have facilitated the accessibility, interoperability and mobility of the web. By developing web-based simulation and modeling tools, the accessibility of these tools can be increased. A layperson considering using a tool developed as a native application may be reluctant to download and install the software on his personal computer (due to fear of viruses and other concerns). On the other hand, the same person may be more willing to open a simple web page that provides an interface for that same tool.

A review of WBS and supporting tools is given in (Byrne et al., 2010). They identify a number of advantages and disadvantages of this model. Some of the advantages are identified as:

- *Collaboration*. Accessing a centralized simulation over the Web allows for collaboration between connected clients.
- *Cross platform capability*. Web applications runs on any operating systems without compiling. This capability relieves the application developer from having to worry about a clients configuration.
- *Controlled access*. Access can be controlled to a Web-based simulation application through the use of passwords, and limited time-span access can be allocated.

- *Versioning, customization and maintenance.* All modifications to the application can be made through the server, allowing for frequent modifications, customization's and updates to be made without having to request user to update their system locally.

A number of disadvantages are also noted, some of which include:

- *Loss in speed.* The user can experience loss in speed when interacting with the tool due to download time and network traffic.

- *Graphical user interface limitation.* The interface provided by the Web are more limited than the desktop counter-part, although improvements are continuous. However, (Wiedemann, 2001) notes that the effort in replicating a complex interface provided by traditional desktop-based simulation tools on the Web is very high and might not be possible.

Different architectures for WBS applications are often characterized by how the components of the application are divided between these two systems. (Byrne et al., 2010) define three primary WBS architectures  local, remote and hybrid  according to the role taken by the web browser.

A *local* architecture (Fig. 3.1) is primarily client-based, thus the simulations and visualization are executed within the users web browser. The server is only responsible of providing the application code, manage user sessions and store persistent data.

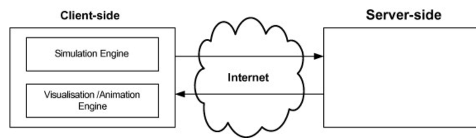



**Figure 3.1:** Local WBS (Byrne et al., 2010)

A *remote* architecture (Fig. 3.2) on the other hand is primarily server-based with the simulation and visualization renderer both executed on the server. The client is merely showing the generated data from the server, such as text and static charts.

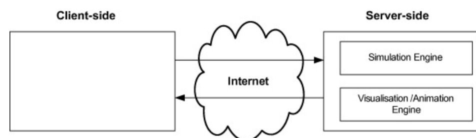



**Figure 3.2:** Remote WBS (Byrne et al., 2010)

A *hybrid* architecture (Fig. 3.3) lies in-between these two extremes, with the server responsible for simulation, while the client handles visualization rendering.

Earlier, local solutions was mainly facilitated using plug-ins. However, today's browsers will typically block plug-ins from running on the client machine due to the security risks involved. This can be circumvented by the user, but requires some technical expertise. Due to this, developers are moving away from these solutions. Luckily, the capabilities

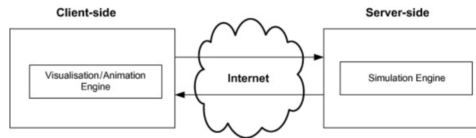



**Figure 3.3:** Hybrid WBS (Byrne et al., 2010)

of the web are evolving, allowing more complex code to run directly in the browser. I.e. HTML5 introduced WebGL in 2011, enabling GPU accelerated rendering in the browser without the need of plug-ins (Marrin, 2011). Several authors have applied WebGL to provide full 3D rendering in web-based simulations. A client-side approach is taken by McMullen et al. (McMullen et al., 2012), who implemented simulation of abstract models in JavaScript, with visualization in WebGL for simulation. A client-side web application of an environmental simulation model is presented in (Walker and Chapra, 2014), which also offers a discussion of the benefits, limitations and potential uses of client-side web applications. The tool Insight Maker (Fortmann-Roe, 2014) is a general-purpose web-based simulation and modeling tool. The environment provides a graphical model construction interface that is implemented purely in client-side code that runs on users machines. In InsightMaker, the client is responsible for model construction, simulation and displaying the results. The server's responsibility is model storing, user management and collaborate editing. A framework for 3D interactive applications on the web was implemented in (Halic et al., 2011). Because the solution required access to custom peripherals, a plug-in was required to be installed on the client machine.

The remote and hybrid architectures are commonly used when migrating existing simulation models to the web as these models are often computationally demanding and their legacy code is not easily translated to JavaScript - the standard client-side scripting language. (Pang et al., 2013) took a hybrid approach in a system for interactive e-learning environment for high performance buildings (HPB). They used using a single Functional Mock-up Unit (FMU) as the simulation back-end running on the server. A generic approach to combine cloud computing and system simulation is given by Bittner et.al. in (Bittner et al., 2015). They present a web interface for uploading, managing and analyzing simulation models. The simulations are based on FMI and runs on the server. Although not a WBS in sense of simulation, a framework for browser-based multiplayer online games was presented in (Chen and Xu, 2011). Which shows that the capabilities of the web are improving, allowing high performance 3D rendering as well as real-time bi-directional communication through WebSockets.

In one of my previous works (Hatledal et al., 2015), a hybrid architecture based on WebSockets and WebGL were implemented, as part of a virtual crane prototyping system.

## 3.2 Networking in games

In the following sections a brief introduction to game networking and how it has evolved with the years, solving some of the problems that comes with it is given.

### 3.2.1 Peer-to-peer

Peer-to-peer (P2P) was the first means of establishing a networked connection between two players in a game. The idea behind the basic P2P model is to have all players start from a common initial state. In order for the game to progress, at each game tick the clients will transmit a set of commands. These will in turn be executed on all connected player machines. The amount of data transmitted is relatively low, allowing games to include a large number of entities. However, this approaches has several limitations. It is very difficult to ensure that the game plays out identically on each players machine. A slight difference in execution of commands, will result in a completely de-synchronized state over time. Another limitation is that in order to ensure that the game plays out identically on all machines, it is necessary to wait until all players command's for that turn has been received before it can be simulated. This means that each player in the game has latency equal to the most lagged player. Finally, because the game must start from a common initial state, it is very hard to facilitate players joining a game once it has started.

A definition of P2P networking and how it differs from a client/server architecture is given in (Schollmeier, 2001), while (Neumann et al., 2007) highlights some of the challenges connected to the peer-to-peer networking model.

### 3.2.2 Client/server

In a pure client/server design, contrary to the P2P scheme, no game code is executed on the client machine. In stead user input is sent to a centralized server, which in turn computes and returns the updated state of the world. The frequency of updates from the server might be lower than than the rendering, which could lead to stuttering motions. In order to counter this, interpolation of values client side are usually performed. With this model, the quality of game experience now depended on the connection between the client and server, and not on the most lagged player. It also made it possible to let players come and go during the game, and the number of players in a game could increase as the required bandwidth on average per-player was reduced. Some background on how client / server architectures work in many on-line action games are presented in (Bernier, 2001).

### 3.2.3 Client-side predication

In earlier networked games like Quake, the players could notice the latency between the computer and the server when issuing commands that would alter the game world. This was due to the fact the client was merely doing what the server said it should do. I.e. when moving the character forward, the command would have to be sent to the server, then the server did some calculations and returned the updated state of the character. The slower the connection, the more noticeable this delay would be. In today's first person shooters there are no such delays. This is thanks to client-side prediction developed by John Carmack for the game QuakeWorld in 1998. Client-side prediction works by putting more logic into the client software. That is, in stead of waiting for the server to compute the next state, the client will make computations on it's own. Then, when the server finally responds, the state is corrected to match the server's. This is a simplified explanation, (Fiedler, 2010) and also (Bernier, 2001) provides a more in-depth description of the process.

## 3.3 Multi-display solutions

Unity3d offers a clustered rendering solution as a separate license [1]. This solutions allows multiple computers to simulate the same scene in-sync with each other and display the result on a cluster of displays. As a result, a scene can be rendered in a dome, CAVE or other layouts using multiple displays. This works by having the same project installed on all machines, running in lock-step synchronization, using the P2P network topology mention in Section 3.2.1. Each machine runs the same simulation, but differs only in the rendering output, rendering only its portion of the entire multi-display setup.

(Obidowski and Jha, 2010) utilized CIGI as part of the software stack used for *The Advanced Deployable Day/Night Simulation Technology Demonstration Project*, with a requirement for at least 16 synchronized Image Generators to render scenes at 60 Hz for a greater than 20 million pixel Ultra-High Resolution laser projection system. Their solution uses a CIGI relay to multicast packet data to all IGs, each with a unique render number. Although all IGs receive the full CIGI packets, each one renders data only in its pre-set FOV, established during the initialization.

## 3.4 Online Virtual Worlds

An online virtual world is an electronic environment where people can work and interact with the digital environment in a somewhat realistic manner (Bainbridge, 2007).

### 3.4.1 Second Life

Second Life (Rymaszewski, 2007) is an online virtual world, developed by Linden Labs. It is similar in many ways to massively multi-player online role-playing games, but unlike games, does not promote a specific goal for the users to pursue. Users of Second Life create visual representations of themselves, called avatars, and are able to interact with places, objects and other avatars. The world is accessed through a client, and multiple client implementations exists.

#### OpenSimulator

In 2007, when Linden Labs made the Second Life client software open-source, an alternative server solution named OpenSimulator which was compliant with existing Second Life clients was developed. OpenSimulator (Fishwick, 2009) is is an open source multi-platform, multi-user 3D application server. It can be used to create a virtual environment (or world) which can be accessed through a variety of clients, on multiple protocols. The default physics engine used by OpenSimulator is the Open Dynamics Engine (ODE), but others can used as well.

[1] `https://docs.unity3d.com/Manual/ClusterRendering.html`

## 3.5 Comparisons of middleware solutions

Several authors has performed comparisons of different middleware solution in terms of speed, ease-of-use and functionality (Dworak et al., 2011; Rizano et al., 2013). (Dworak et al., 2011) makes a comparison of a set of middlewares that supports the requst-reply pattern, for the purpose of replacing the use of CORBA for operation of the CERN accelerators. These include ICE, Thrift, ZeroMQ, YAMT4, RTI (DDS implementation) and QPID. A couple of DDS implementations, CoreDX and OpenSplice, were discarded for further evaluation due to complexity or licensing issues. Their findings suggests that the DDS API, though well documented, is not easy to use nor compact. Additionally, they found the amount of settings and concepts provided by the standard overwhelming, rendering the products to be cumbersome and difficult to use. Three libraries were qualified for further prototyping; Ice, ZeroMQ and YAMI4, with ZeroMQ receiving the best score based on their requirements. After further evaluation, ZeroMQ was chosen as the new middleware (Dworak et al., 2012).

(Rizano et al., 2013) presents the performance evaluation of three publish-subscriber middlewares; ZeroMQ, OpenDDS and ORTE (Open Real-Time Ethernet). The evaluation focused on real-time performance in the context of embedded systems. For their use-case, ZeroMQ was found to be the more performant solution with the lowest average latencies and the lowest worst case latency.

## 3.6 Comparisons of serialization formats

Choosing an appropriate serialization format for a networked software application is important, thus several authors has made comparisons of various formats in order to find a good fit for their application.

In (Nurseitov et al., 2009), JSON is compared against XML (Extensible Markup Language). The authors found JSON to be significantly faster than XML when comparing resource utilization and relative performance of applications that use the interchangeable formats.

(Müller et al., 2010) found that Google Protocol Buffers, with their use case, reduced their message sizes with 75%, and processing time by at least 11 and 59% compared to XML and SOAP (Simple Object Access Protocol) respectively. Furthermore, they note the importance of choosing an appropriate choice of communication protocols with regards to the Internet of Things.

In (Dworak et al., 2012) some 20+ alternatives were reviewed for serialization in conjunction with their ZeroMQ middleware, with MessagePack being chosen over other candidates such as Apache Avro, CORBA serialization module and Google Protocol Buffers because of direct support for dynamic typing, compact binary serialization, support for all needed data types, operating systems and programming languages, good documentation, product maturity and the active community behind it.

(Maeda, 2012) compares twelve object serialization libraries in XML, JSON and binary formats from qualitative and quantitative aspects. A common example is chosen and it is serialized to a file using each library in a supported format. The size of the serialized file and the processing time are measured during the execution to compare all object seri-

alization libraries. The authors found Google Protocol Buffers to be the fastest to serialize, while Thrift using JSON was faster to de-serialize.

It should be noted that there will be no single best solution with regards to serialization formats, as requirements will be different from application to application.

## 3.7 Synchronized wave visualization

No literature was found on how synchronized wave visualization are handled in client/server architectures. However, games like Battlefield 4 (2013) and Battlefield 1 (2016) do employ such a mechanism. An example of such waves from Battlefield 4 are given in Figure. 3.4. In these multi-player games, players can drive boats in rough weather with significant wave heights. Therefore, the different clients needs to synchronize the waves. As the games are closed-source, it can only be assumed that they employ a strategy where waves are generated by a heuristic wave generator and time and/or a seed is shared.

**Figure 3.4:** Screenshot from the 2013 title Battefield 4 showing a player driving a Jet Ski on a dynamic water surface.

# Chapter 4

# Materials and Method

This chapter describes the materials and methods used to complete this project.

## 4.1 Gradle

Gradle is the build system used by Vico. Similarly to Maven, Gradle keeps the project independent of a particular IDE and manages artifact management. What makes Gradle so powerful is the fact that the build process is managed using a Groovy based DSL. Actually, full fledged Groovy programs and even Java code, can run within the build files. Alternative build systems like Ant and Maven is based on XML, making them much less flexible.

The Vico source code itself is split into several Gradle sub-projects. This is to help clarifying what is what, and not mix dependencies. Some sub-projects include core, maths, geometry, rendering and machine-learning. Some of these projects can be considered as plugins to Vico.. The Java code developed for this project is managed as Vico sub-projects, logically separated from the core and functions as a plugin.

## 4.2 Three.js

HTML5 brought along the WebGL standard, which paved way for a whole new set of browser-based content. No longer was it necessary to download miscellaneous plug-ins in order to run GPU enabled 3D content in the browser. Three.js is one of many new libraries that emerged after this introduction. Among these libraries, three.js is perhaps the most successful and widely adopted. Compared to 3D game engines on desktop, three.js and many other similar libraries for WebGL, lacks an IDE and other visual tools, however it makes up for this with an easy to use API and flexibility.

## 4.3 Unity3d

Unity3d is a professional cross-platform game engine developed by Unity Technologies. The engine itself is written in C/C++, whereas user code (scripts) are written in C#, UnityScript or Boo. Code can be edited using Visual Studio or MonoDevelop. Mono is used in order for C# to work on platforms other than Windows. Creating scenes, managing assets etc. are done through an editor. As the production version of Unity3d only support .NET 3.5 and lower, this work relies on a beta version of Unity3d (build Unity 2017.1.0b5). This version can run modern C# code, which is a requirement from some of the 3rd party dependencies used.

## 4.4 Other 3rd party dependencies

Table. 4.1 lists the third party software libraries used in this project. The project could not have been carried out without them.

**Table 4.1:** Third party software libraries used in this project

| Name | Language | Purpose |
|---|---|---|
| Canvas.js | JavaScript | Plotting data in the browser |
| Runtime OBJ import | C# | Import .OBJ 3D models in Unity during runtme |
| Google Protocol Buffers | C# / Java | Encoding/decoding protocol buffers in Java / C# (Unity3d) |
| Protobuf.js | JavaScript | Encoding/decoding protocol buffers in the browser |
| Java-WebSockets | Java | WebSocket support in Java |
| JSON.NET | C# | JSON support in C# (Unity3d) |
| WebSocket-sharp | C# | WebSocket support in C# (Unity3d) |
| ZeroMQ | C# | ZeroMQ support in C# (Unity3d) |
| JeroMQ | Java | ZeroMQ support in Java |
| GSON | Java | JSON support in Java |
| Three.js | JavaScript | WebGL rendering in the browser |

## 4.5 Workflow

In this section the workflow used during the development is described.

**JIRA** JIRA is a web-based issue tracking system, developed by Atlassian. In this project it has been utilized to plan and monitor code development. Development has been driven forward by so called *sprints*, which is a key part of the Scrum methodology (Schwaber and Beedle, 2002). Each sprints starts by identifying the work that should be completed

during the next iteration. The sprint ends with a sprint review and sprint retrospective. In this project, the duration of a sprint was defined as two weeks.

**Confluence** Confluence is a web-based team collaboration software developed by Atlassian. In this project it has been used to create meeting notes, Gliffy diagrams and to store project related texts and wiki creation.

**Git** Using a version control system (VCS) is absolutely crucial for any kind of large scale software project. A VCS allows you to track changes throughout the lifetime of the project, allows teams to collaborate, makes it easy to test new features, deliver and maintain customer deliveries etc. Vico uses Git as it's VCS. Actually Vico is the the sum of multiple git repositories. For instance, the clients developed in this project are independent git repositories.

The workflow used for this project looks similar to that shown in Fig. 4.1. Master is the stable branch, with develop being the spearhead in terms of new functionality and features. New ideas etc, may occasionally branch out from develop and merged back in if deemed worthy of keeping. Once new features has been completed and tested on develop they are merged back into master.

**Figure 4.1:** Typical Git workflow

## 4.6 Benchmark setup

In order to objectively assess the performance of the implemented transports, some benchmarks has been carried out. Here, two computers are used. One acts as a server and runs the simulation code, while the other acts as a client. To ensure a stable test scenario, the two computers are connected by Ethernet through a network switch. This ensures that unpredictable behaviour that may arise when connected through the Internet are eliminated. The switch used is an 5-port Netgear GS605 Gigabit Ethernet switch. Specifications for the computers used are given in Table. 4.2, and the physical configuration is shown in Figure. 4.2.

**Table 4.2:** Specification of the computers used for the benchmark

| Role | Type | CPU | RAM | Network |
|---|---|---|---|---|
| Server | Desktop | i7-4770 @ 3.40 GHz | 16 GB | Gigabit Ethernet |
| Client | Laptop | i7-6600U @ 2.40 GHz | 16 GB | Gigabit Ethernet |

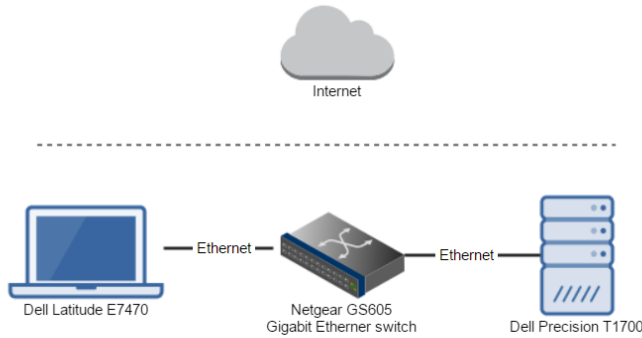



**Figure 4.2:** Benchmark physical configuration. Connectivity is restricted to the local area network

## 4.7 Visualization lab

The visualization lab located at NTNU Aalesund is used to test the system as an enabler for multi-display solutions. The lab features 12 projectors and a cylindrical shaped wall as seen in Figure. 4.3.

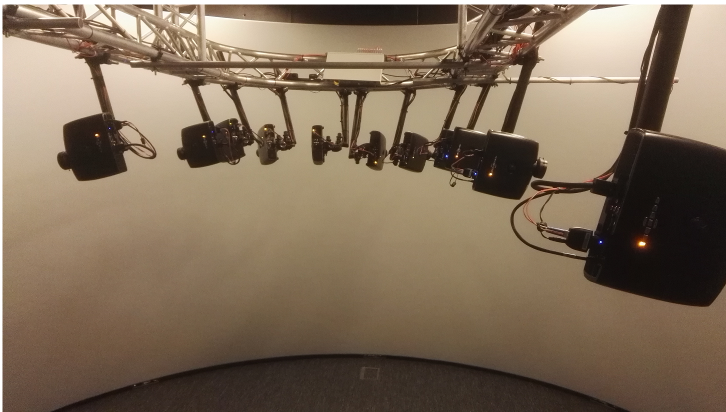


**Figure 4.3:** View of the visualization lab, which features 12 projectors that projects onto a large cylindrical wall

# Chapter 5

# Results

This chapter presents the results of this thesis, which includes implementation details for the server-side code as well as the clients developed. Additionally, benchmarks are presented and it is shown how the system can be used to achieve clustered rendering.

## 5.1 Server-side Implementation

This section presents the reader with details regarding the server-side implementation of the system. That is, the software layer that handles the link between Vico simulations and remote clients. In order for a Vico simulation to be made accessible over the network a *RemoteManager* must be added. The content of this class is shown in Listing.7.14. In short, this class is responsible for managing end-points and message handlers.

### 5.1.1 Framing

In order to simplify messaging, messages are wrapped in a set of frames. In this context, a frame is simply a chunk of bytes led by a four byte header indicating the length of the data. An arbitrary number of frames can be included in a message. Also, an additional four byte header is present at the beginning of the message indicating the total length of the remaining message.

Imposing such a message structure is useful for several reasons:

1. It adds consistency. Data arriving from different sources has the same format.

2. Frames can be implemented as a deque, allowing frames to be popped of the stack as they are consumed.

3. Makes it possible for protocols without a built-in mechanism for indicating the end of a message, such as TCP and UDP, to easily receive messages.

An example is given in Figure. 5.1, where a message with two frames is defined. When decoded, the message prints ”Hello World!”. However, do note that a frame is not self-describing. The type of content a frame holds, be it a string or a binary blob, depends on the context.

| Header | Data | | | | | | Notes |
|---|---|---|---|---|---|---|---|
| 19 | | | | | | | |
| 5 | 72 | 101 | 108 | 108 | 111 | | Hello |
| 6 | 87 | 111 | 114 | 108 | 100 | 33 | World! |

**Figure 5.1:** Message framing

The Java implementation of is given in Listing. 7.12 (a frame) and Listing. 7.13 (the message).

### 5.1.2 End-points

In order to ensure that a wide variety of clients can connect, multiple end-points has been implemented. These are: TCP/IP, UDP, WebSockets and ZeroMQ. New end-points can be added to the system, as long as they comply with the interface shown in Listing. 5.1.

**Listing 5.1:** VicoServer.java

```java
public interface VicoServer {
    public int getPort();
    public void start();
    public void stop();
    public String getName();
}
```

Clients that connects to one of the end-points are wrapped in a generalized interface shown in Listing. 5.2, allowing other parts of the software to write data to it without knowing how it eventually will be delivered. The write method is expected to return immediately. As such, implementations are expected to save the data to a buffer and write it as soon as possible on another thread.

**Listing 5.2:** WritableConnection.java

```java
public interface WritableConnection {
    public void close(); //close the connection
    public boolean isOpen(); // is the connection open?
    public void write(byte[] data); //write data
}
```

#### TCP/IP

The implemented TCP/IP server uses the built in *java.net.ServerSocket* class. The implementation spawns a new thread for each connected client. Furthermore, each client spawns

two additional threads. One for reading and one for writing. The implementations keeps an outgoing message buffer, such that send invocations can return immediately.

### UDP

Java comes bundled with UDP support through the *java.net.DatagramSocket*, and is used to implement the server side UDP support.

Because UDP datagrams can be no larger than 65 kilobyte, larger messages needs to be split into smaller messages and sent separately. Logic for handling this was added, and the implementation is given in Listing. 5.3. Here, the data is split into as many smaller messages as necessary. Each chunk is then sent separately. Each message consist of an 24 byte header given in Table. 5.1, containing the necessary information to reassemble the message on the receiver end.

**Table 5.1:** UPD header

| Frame | Description | Datatype |
|---|---|---|
| *0* | Time stamp | int64 |
| *1* | Full-size message length | int32 |
| *2* | Number of sub-messages | int32 |
| *3* | Current message number | int32 |
| *4* | Sub-message length | int32 |

**Listing 5.3:** UDP send implementation

```
@Override
public void send(byte[] data) {

    long timestamp = System.currentTimeMillis();
    List<byte[]> chunks = getChunks(data); //split data
    for (int i = 0; i < chunks.size(); i++) {

        byte[] chunk = chunks.get(i);
        ByteBuffer buf = ByteBuffer.allocate(chunk.length + 24);
        // UDP HEADER
        buf.putLong(timestamp); //time stamp
        buf.putInt(data.length); //original message length
        buf.putInt(chunks.size()); //number of sub-messages
        buf.putInt(i); //message number
        buf.putInt(chunk.length); //sub-message length
        //
        buf.put(chunk);
        buf.flip();

        byte[] send = buf.array();
        try {
            serverSocket.send(new DatagramPacket(send, send.length, remoteAddress, remotePort));
        } catch (IOException ex) {
            Logger.getLogger(UDPConnection.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

### ZeroMQ

The ZeroMQ end-point uses the pure java implementation of ZeroMQ, JeroMQ (JeroMQ, 2017). While it would be natural to implement the ZeroMQ end-point using the commonly used PUB-SUB and REQ-REP patterns, this would require clients to establish two separate

connections. One for each pattern. Furthermore, ZeroMQ handles subscriptions internally, making subscribing to a PUB socket different than one of the other types of implemented transports. Also, the REQ-REP pattern does not allow asynchronous messaging.

To overcome this, the implementation uses a DEALER-ROUTER pattern. This allows asynchronous messaging, and both request-reply and publish type messages are handled on a single port. Using this pattern, it is necessary for clients to unwrap the first frame in an incoming message as it contains the sender identification and can be safely ignored in this use case.

#### WebSocket

As Java SE does not contain a WebSocket implementation, the 3rd part library Java-WebSockets (Rajlich, 2017) was used to implement the WebSocket server. WebSockets allows for messages to be sent/received either as a string or as binary data. As we are encoding messages using frames as described in 5.1.1, only binary data is sent/received.

### 5.1.3 Message handling

In order for the system to properly respond to a received message, all messages sent/received starts with a key frame. This frame holds a single byte, indicating the message type. Each valid key maps to a message handler, which are responsible for decoding the remainder of the message and act upon it. This mechanism is extensible, allowing simulation creators to add new message handlers, and is an implementation of the *Message Router* pattern (Buschmann et al., 1999) shown in Figure. 5.2. The system contains several default handlers as seen in Table. 5.2.

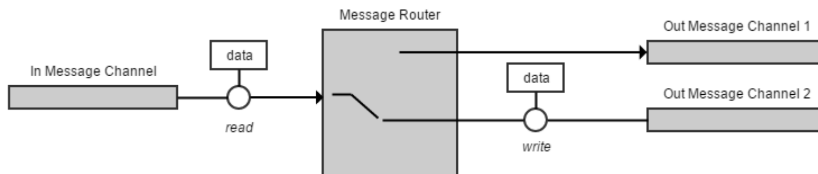



**Figure 5.2:** Message routing. Adapted from (Buschmann et al., 1999)

Message handlers must implement the *MessageHandler* interface shown in Listing. 5.4. The first parameter is the remote connection, while the second is the message to handle. The system can be extended by adding new message handlers. A message handler is mapped to a unique key, and will be triggered when a incoming message contains that key in the first frame. Keys 0x000 - 0x015 are reserved for internal use. The different handlers and their keys are shown in Table. 5.2.

**Listing 5.4:** MessageHandler.java

```
public interface MessageHandler {
    public void handle(WritableConnection con, VicoMsg msg);
}
```

**Table 5.2:** Implemented message handlers

| Key | Handler |
|---|---|
| 0x000 | RequestHandler |
| 0x001 | SubscriptionHandler |
| 0x002 | KeyHandler |
| 0x003 | PlotHandler |
| 0x004 | ServiceHandler |
| 0x005 | SpawnHandler |
| 0x006-0x015 | Reserved |

A key element to the implemented system is that clients can choose to use and implement responses for individual handlers (with the exception of the *SubscriptionHandler* which depends on the *RequestHandler* for the initial setup). A client that is only interested in plotting can for instance choose to only implement support for the *PlotHandler* messages.

### Asynchronous messaging

All implemented handlers that responds with a message supports asynchronous messaging, and expects an unique id as the first[1] frame in the message. Clients that are not asynchronous are expected to use an empty frame. This id is then wrapped into the response message, allowing a client to invoke a callback mapped to that particular id.

### Multiple format support

To increase the flexibility of the system, the system support payloads to be encoded in multiple formats. In this work, JSON and GBP was implemented. However, the design is extensible, allowing further formats to be added.

As mentioned previously, a frame is not self describing. So, in order for a handler to know what format a frame contains, the frame preceding the formatted frame is used to identify the format used.

The idea is shown in Figure. 5.3, and follows the *Message Translator* pattern (Buschmann et al., 1999).

[1] While the key frame is always the first frame in a message, it is popped of the stack before redirected to the appropriate handler

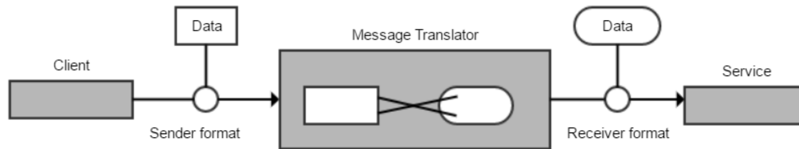



**Figure 5.3:** Message Translator. Adapted from (Buschmann et al., 1999)

### Thread safety

In many cases, handlers are tasked with collecting data from the simulation which is running in a separate thread. In order for the handler thread not to interfere with the running simulation, the Vico core was modified to allow inserting two types of tasks that are invoked as part of the simulation loop. These are the blocking *invokeAndWait*, and the non-blocking *invokeLater* methods which both takes a *Runnable* as a parameter, mimicking the Java swing API. This Runnable is then invoked by the simulation thread, ensuring thread safety. Whenever a handler is to retrieve simulation data or modify the simulation in some way, one of these functions should be used. Alternatively, the handler could register a listener (which is invoked regularly by the simulation at certain state events such as *init*, *step*, *post-step* and *terminate*).

### RequestHandler

The *RequestHandler* is used for retrieving simulation setup and updates in a request-reply fashion. Payloads uses the schema defined in Listing 7.1. The client sends a *Request* and receives a *Response*.

A complete message sent from the client to the server, requesting a simulation setup could look as described in Table 5.3. In this case, JSON is used to encode the payload. If GPB were to be used, frame nr. 3 would read 0x000 and the content in frame nr. 4 would not translate to a JSON string, but a GPB message.

**Table 5.3:** Message requesting a simulation update. The payload is encoded using JSON

| Frame | Description | Value | Notes |
|---|---|---|---|
| *1* | Handler key | 0x000 | 0x000 redirects to the RequestHandler |
| *2* | ID | xxxx-xxxx | Some unique string used to support asynchronous messages |
| *3* | Payload Format | 0x001 | 0x001 signals that the following payload is encoded as a JSON string |
| *4* | Payload | {type: 0} | Request simulation setup |

Similarly, the message sent back to the client would look as in Table 5.4

**Table 5.4:** Simulation setup response from the *Requesthandler*

| Frame | Description | Value | Notes |
|---|---|---|---|
| *1* | Handler key | 0x000 | Message originates from the RequestHandler |
| *2* | ID | xxxx-xxxx | Same ID as in Request |
| *3* | Payload Format | 0x001 | 0x001 signals that the following payload is encoded as a JSON string |
| *4* | Payload | {simulationSetup: ...} | Simulation setup |

### SubscriptionHandler

As the name suggests, the *SubscriptionHandler* is responsible for managing subscriptions. Once a client has subscribed, it will continuously push updates to it. These updates include general simulation updates which are sent at regular intervals, as well as non-regular events such as added or removed entities.

### ServiceHandler

The *ServiceHandler* is an interface to a JSON-RPC API. Simulation designers can register custom classes to this handler in order to easily expose the desired parts of their simulations. An example of such a service is given in Listing. 5.5. A method in this service can then be accessed remotely by sending the string {*"jsonrpc": "2.0", "method": "VicoService.setAttemptRealTime", "params": [true]*} as payload to the handler. This interface is incredible powerful, but comes at the expense of performance as it depends on reflection. Because reflection involves types that are dynamically resolved, certain Java virtual machine optimization's cannot be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications (Oracle, 2015).

**Listing 5.5:** VicoService.java

```
public class VicoService extends RpcService {

    private final VicoSimulation sim;

    public VicoService(VicoSimulation sim) {
        super("VicoService");
        this.sim = sim;
    }

    @RpcMethod
    public void setAttemptRealTime(boolean flag) {
        sim.setAttemptRealtime(flag);
    }

    @RpcMethod
    public void setTimeStep(double timeStep) {
        sim.getTime().setTimeStep(timeStep);
    }

}
```

#### KeyHandler

The *KeyHandler* allows key presses captured in the client application to be registered with a Vico simulation. This allows clients to trigger simulation code that reacts to key input.

#### PlotHandler

The *PlotHandler* enables clients to plot data from a running simulation. A client can ask for available plots, and subsequently ask for the necessary data to initialize the plot. After initialization, the client can regularly ask for updates at custom intervals. The schema used for messages designated to this handler is given in Listing. 7.4.

#### SpawnHandler

This handler makes it possible for remote clients to spawn objects within a simulation. The client can define the type of shape as well as position, rotation and initial velocity of the spawned object. The schema used for messages designated to this handler is given in Listing. 7.5.

### 5.1.4 Messages for 3D visualization

This section is used to identify and describe key message types used by the system, which allows remote clients to render a Vico simulation.

#### Simulation message

The simulation message consists of the simulation's UUID, name, time and additionally an entity, which is the root of the scene-graph. Because an entity is a tree, all entities contained in the simulation are accessible given the root. Both a setup message and a lighter update message is defined. The schema is given in Listing. 5.6. These two messages contains all the information necessary to setup and continuously render a simulation.

**Listing 5.6:** SimulationMsg.proto

```
message PBSimulationSetup {

    string name = 1;
    string uuid = 2;
    PBTime time = 3;
    PBEntity root = 4;

}

message PBSimulationUpdate {

    PBTime time = 1;
    PBEntityUpdate root = 2;

}
```

#### Entity message

An entity is a node in a scene-graph, and contains the entity's UUID, name, children and a list of attached components. The transform, geometries, and curves are components, and could reside inside the general list of components, but because they are so common, they

have been given their own message types for performance reasons and ease of parsing. As with the simulation message, both a setup and a update message is defined. The complete schema is given in Listing. 5.7. The transform contains the position and rotation of the entity, and is given in local coordinates. As such, an implementation is required to build a scene-graph. Local coordinates was chosen because, as compared to world coordinates, they are more effective as they do not change when the parent changes. Allowing for optimization's to be made.

**Listing 5.7:** EntityMsg.proto

```
message PBEntity {

    string uuid = 1;
    string name = 2;
    PBTransform transform = 3;
    repeated PBGeometry geometries = 4;
    repeated PBCurve curves = 5;
    repeated PBEntity children = 6;
    repeated PBComponent components = 7;

}

message PBEntityUpdate {

    string uuid = 1;
    PBTransform transform = 2;
    repeated PBCurveUpdate curves = 3;
    repeated PBEntityUpdate children = 4;
    repeated PBComponent components = 5;

}
```

### Geometry message

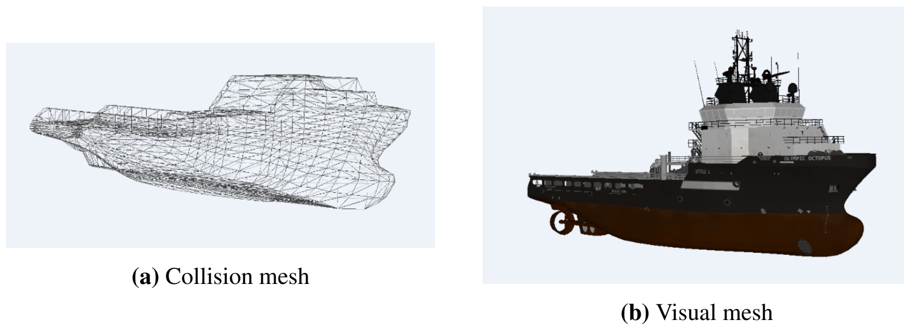
A geometry message consists of:

- Either a color or a texture
- An offset position relative to it's entity
- An offset rotation relative to it's entity
- A bounding-box
- A flag indicating that it is a visual geometry.
- A flag indicating that it is a collision geometry
- The actual shape

Note that both the visual and collision flag can be set to true, in which case the client should load the geometry twice, but with different materials. Collision geometries should be rendered as a wire-frame, while visual models should use the color/texture/material set. An example is given in Figure. 5.4.

Supported shapes consists of:

- **Line** - A simple line between two points
- **Box** - A box is defined by three floats describing it's half-extents.
- **Sphere** - A sphere is defined only by it's radius.



(a) Collision mesh

(b) Visual mesh

**Figure 5.4:** Example of rendering geometry for visual and collision

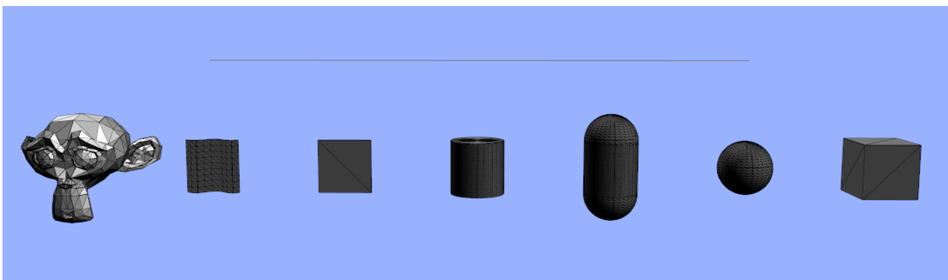
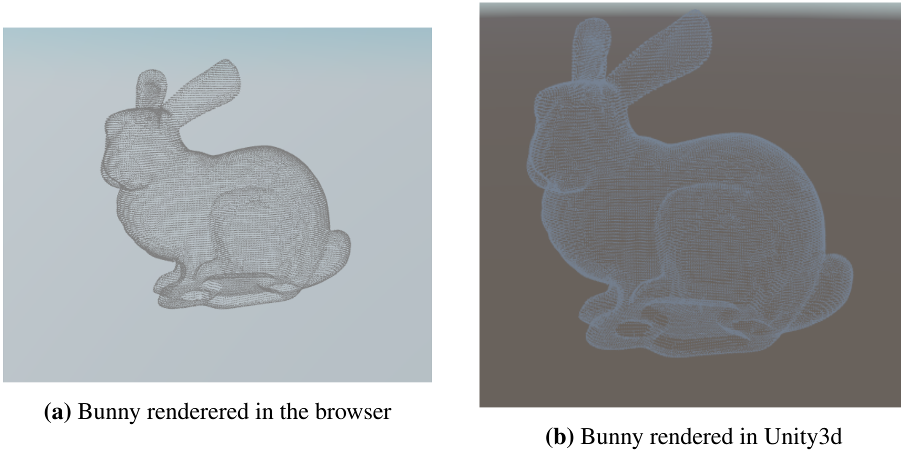


**Figure 5.5:** A collection of supported geometries: Mesh, Line, Height-field, Plane, Cylinder, Capusle, Sphere and Box

- **Cylinder** - A cylinder is defined by it's height and radius.
- **Capsule** - A capsule is defined by it's height and radius.
- **Point Cloud** - A point cloud is defined by a list of points and a color, or optionally vertex colors
- **Plane** - A plane is defined by a height and a width.
- **Height-field** A height-field is defined by a height and a width, as well as the number of width- and height segments. Optionally, a list of height values can be defined. In which case the height-field functions as a terrain.
- **Mesh** - This message merely contains a link to the location of 3D model, which could be either a file stored on the simulation host or a URL pointing to a location on the Internet. It is up to the client to do a separate query for the required data over HTTP, which could then be loaded asynchronously. The format of the model will be one that Vico supports, which currently is .OBJ and .STL.

The different types of supported geometries (except the point cloud) are shown in Figure. 5.5. Additionally a height-field used as terrain with texture is shown in Figure. 5.7, and a point cloud is shown in Figure.5.6.



**(a)** Bunny renderered in the browser

**(b)** Bunny rendered in Unity3d

**Figure 5.6:** Point cloud rendering

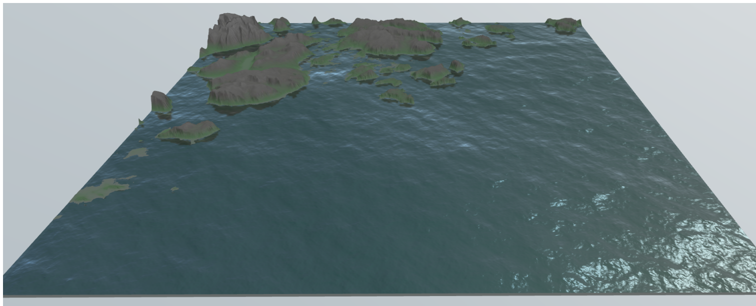


**Figure 5.7:** Example of a height-field as textured terrain, rendered in the browser client

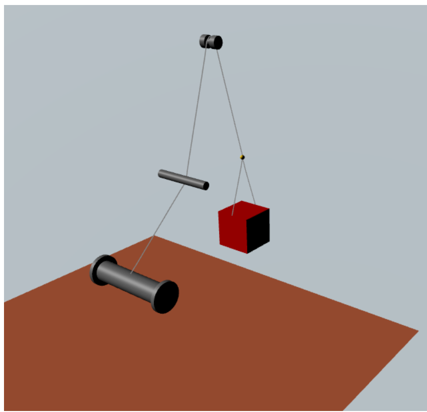
### Curve message

Curves are used to represent any type of curve, such as wires, cables etc. It is defined by a radius, color and a set of points as seen in Listing 5.8. Curves can be dynamic, so a update message that contains a new set of points is also defined. The number of updated points does not need to be equal to the initial.

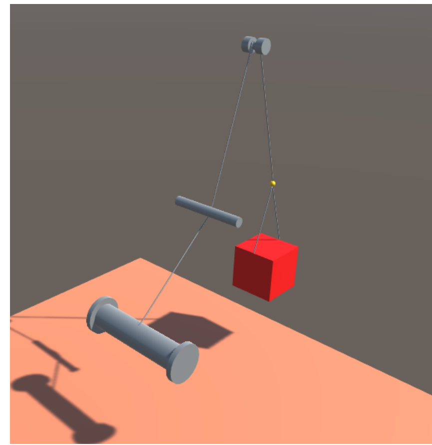
**Listing 5.8:** CurveMsg.proto

```
message PBCurve {

    string uuid = 1;
    float radius = 2;
    PBColor color = 3;
    repeated PBVec3 points = 4;

}

message PBCurveUpdate {

    string uuid = 1;
    repeated PBVec3 points = 2;

}
```

Usage of the curve type is demonstrated in Figure. 5.8



**(a)** Scene rendered in the browser



**(b)** Scene rendered in Unity3d

**Figure 5.8:** Drum simulation featuring the curve type for representing a wire

### 5.1.5 Water

Water is represented by a dynamic height-field. In order for clients to visualize the water with waves, the server could send the current height information for each point in the grid, but this would equal to roughly 30MB of data per second for a grid with resolution of 512x512 (single-precision floating-point at 30hz). Instead, waves are visualized by letting the server and any connected clients implement the same wave model. In this way, time is the only continuous variable that needs to be shared, and is controlled by the server as seen in Figure. 5.9. Additionally, model specific variables must be synced.

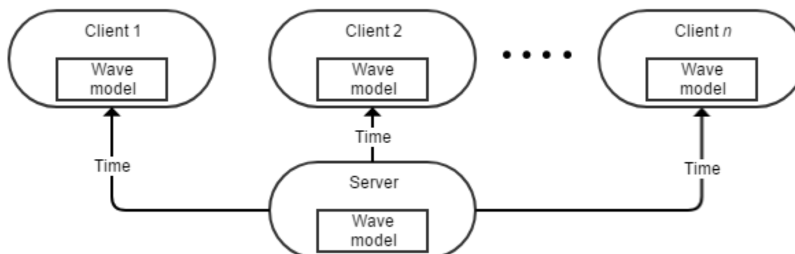



**Figure 5.9:** Wave synchronization

### 5.1.6 Directory service

The directory service is a standalone application server that is responsible for keeping track of active simulations. When a new simulation is created, it will notify to the directory service about it's existence along with the necessary data needed for clients to reach it.

When clients wants to access a simulation, it will query the service for available simulations. The information received will contain the necessary information for the client to access the selected simulation directly, as seen in Listing. 5.9.

A view of the application GUI is given in Figure. 5.10, where an administrator can monitor which simulations are active, and view basic information about them such as the network address and ports.

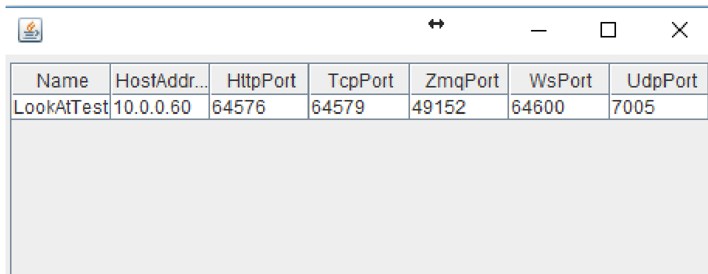



**Figure 5.10:** Directory Service GUI

ZeroMQ is used for communication with the simulations. The service uses a *Dealer* socket, while the simulation uses a *Router* socket. Heartbeats, which are small messages signifying "I'm alive", are sent between them so that disconnects can be detected. With ZeroMQ, the order of which to start clients and servers does not matter, allowing the directory service to start after a simulation has started. If the directory service should crash, it can be restarted and clients will automatically resend their connection info.

A complete description of the information sent is given in Listing. 5.9.

**Listing 5.9:** ConnectMsg.proto

```
message ConnectMsg {

    string uuid = 1;
    string simulationName = 2;
    string description = 3;
    string hostAddress = 4;

    map<string, uint32> ports = 5;

}
```

Clients will query the directory service for available simulation through a HTTP request, which will return a list of all available simulations as a JSON formatted string.

## 5.2 Client-side implementation

This chapter describes details of the implementation of two client applications that has been implemented to make use of the server side solution described previously. These are:

- a web application written in HTML5/JavaScript
- a desktop application written in C# under the game engine Unity3d

### 5.2.1 Web application

The browser client consists of three web pages. One for selecting a simulation, another for 3D rendering/interaction and a last one for displaying plots.

For communication with the simulation back-end, the WebSocket API is used. Both JSON and GPB is supported message formats. JSON is built into the JavaScript API, while the open-source JavaScript protobuf.js (Wirtz, 2016) is used to encode/decode GPB data. Once decoded, the structure of a GPB message is identical to that of an equivalent JSON message. Thus allowing the same code to handle parsing for both message types. Listing. 7.11 shows the code handling networking for the web client.

**Simulation Select** The selection page as seen in Figure. 5.11, lets the user define the address of the directory service server. Once this has been done, a HTTP request will be sent to the server and a list of available simulations will be presented to the user through a drop-down menu. The user can then choose to either render the simulation in 3D or view 2D plots.

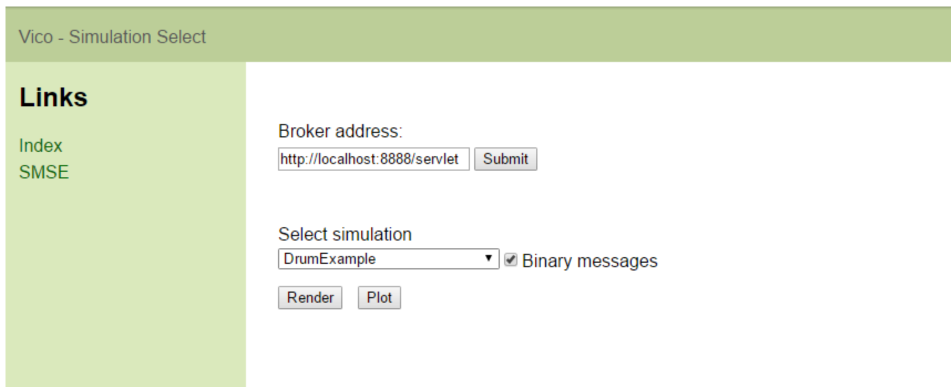



**Figure 5.11:** Selection page for simulations in the browser

**Rendering/Interaction** 3D rendering in the browser is facilitated using WebGL. In order to simplify content creation, the open source WebGL framework *three.js* (Cabello, 2010) is used.

A settings panel, shown in Figure.5.12, allows the user to toggle the visibility of collisions models (wireframe models) and axes of the transforms.

**Plotting** Using this page, the user can select among the available plots from the simulation and visualize them as the simulation is progressing, as seen in Figure. 5.13. The 3rd JavaScript library Canvas.js (fenopix, 2013) is used to facilitate dynamic plotting in

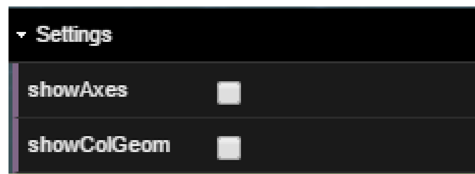



**Figure 5.12:** Render settings in the browser

the browser. Each plot is wrapped in an collapsible accordion, and is only rendered when visible.

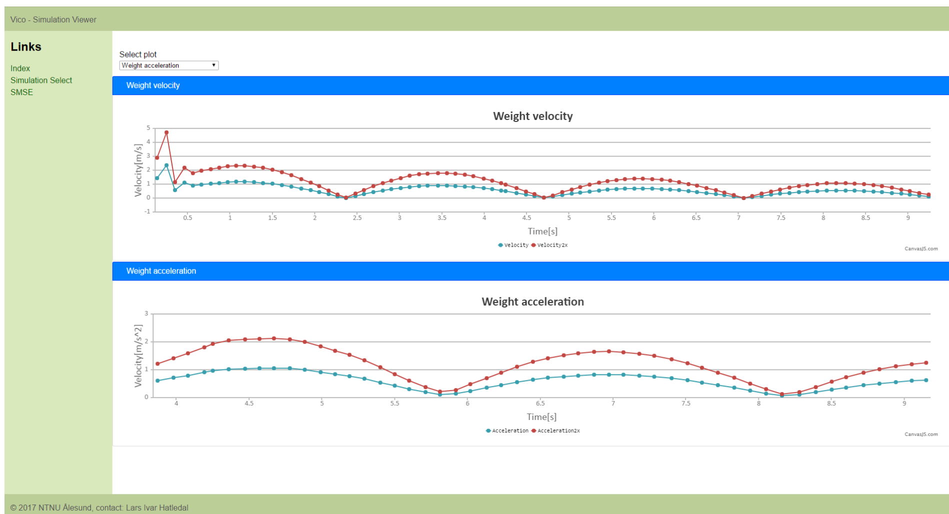



**Figure 5.13:** Plotting in the browser

### 5.2.2 Unity3d

The Unity3d client is implemented using the C# programming language. Because the C# version used by Unity3D is quite old, some third party libraries such as ZeroMQ and Google Protocol Buffers will not work with the current regular release of Unity3d (Unity 5.6). Fortunately enough, the Unity team has been releasing experimental builds that can run newer C# code. Such an experimental build (Unity 2017.1.0b5) was used to implement the Unity3d client, allowing ZeroMQ sockets and Google Protocol Buffer to be utilized.

All available transport protocols has been implemented for this client. That is, TCP/IP, UDP, ZeroMQ and WebSockets. However, the implemented UDP client can only handle messages smaller than 65 kilobyte, limiting its usage to simulation scenes with a small number of objects. Listing. 7.10 shows the code doing the heavy lifting w.r.t networking. Both JSON and GBP can be used for encoding/decoding of the message payloads. Parsing for both is equal, as JSON messages are converted to equivalent Protocol Buffer generated

classes before any parsing is performed.

In order for users of the Unity3d client to easily select among the available remote simulations, a GUI element was added to the editor's menu bar. When clicking the menu element, a script will query the directory service for simulations over HTTP. For each simulation, a GUI element is dynamically created and will appear in the menu bar. The new menu items is shown in Figure. 5.14. On selection, the remote simulation will load. Multiple simulations can run in the same scene at once. Also multiple instances of the same simulation can run within same scene, where each instance can choose which transport and serialization method to use.

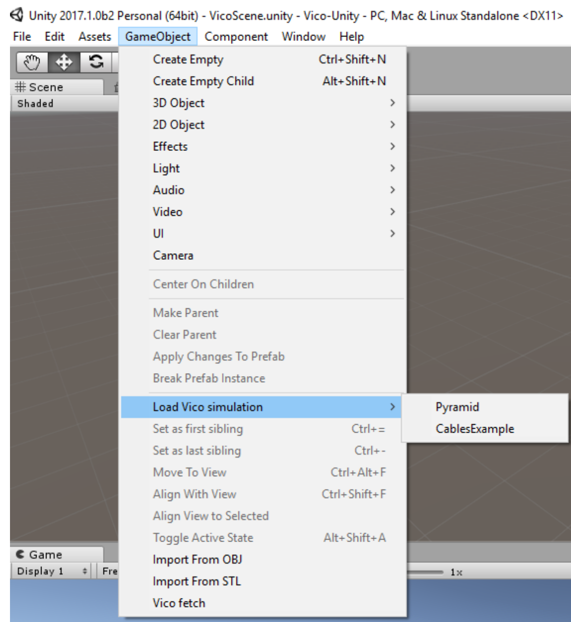



**Figure 5.14:** New Unity3d menu items. *Vico fetch* will initiate a query for available simulations, which will populate the *Load Vico Simulation* menu

**Missing assets** Unity3d lacks built in support for lines, point clouds, curves, heightfields and run-time import of 3D models. These had to implemented in order to properly visualize the remote simulations.

- **Line -** Lines are implemented using the built in LineRenderer component, which allows lines to be rendered in-game (in contrast to Debug.DrawLine which is only visible in the scene view).

- **Point cloud -** Point clouds are implemented as a Mesh, which is rendered using GL.POINTS.

- **Curve -** Curves were implemented by porting the code used by Three.js to create the

*TubeBufferGeometry* [2] to C#. The result can be viewed in Fig. 5.15. This geometry is used to render curved 3D objects such as wires etc.

- **OBJ support -** Support for runtime loading of OBJ (Wavefront Object File) 3D models, including material support, was achieved using the free *Runtime OBJ Importer* from the Unity3d asset store. However, this importer only works with files located on the file system. In order to load .OBJ, .MTL and texture files from an URL source, the code in Listing. 7.9 was added. Using this code, data can be read from the remote resource over HTTP and is subsequently saved to the local file-system, allowing the standard loader to read the files. Once completed, the files are deleted.
- **STL support -** Unity3d does not include a loader for the .STL (STereoLithography) 3D format. This support was implemented, and can be used to load models both from the editor and during run-time. Additionally, the loader can load data from an URL. The code is shown in Listing. 7.8.

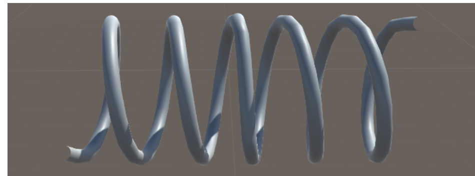


**Figure 5.15:** Curve implementation in Unity3D

**Multi-display support** In order to support visualization of the simulations in multi-display environments such as domes, the Unity3d client was fitted with client side code that makes it possible to project the simulation onto multiple displays. This solution works by running multiple instances of Unity3d, each controlling a projector. One of the instances is assigned the master role, while the others connect to the master as slaves. ZeroMQ is used for networking and the PUB-SUB pattern is used.

- **Master** The master script (see Listing. 7.6) is attached to the main camera, and will continuously publish the position and rotation of it's transform.
- **Slaves** The slave script (see Listing. 7.7) is attached to the main camera. Subsequently, the transform will be updated according to the transform received from the publisher. The slave is responsible for configuring the view offset between itself and the master.

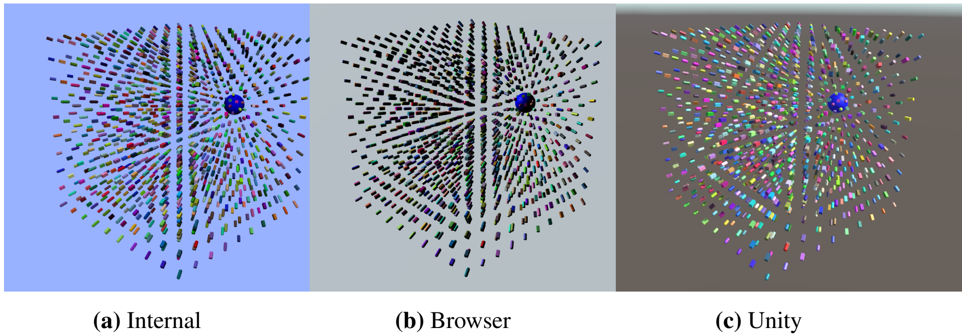
## 5.3 Benchmarks

In order to see the performance of the implemented networking options, a simulation scene with a large number of objects were created. That is, 1001 entities - each with an attached

[2] `https://threejs.org/docs/#api/geometries/TubeBufferGeometry`

geometry. The scene is shown in Figure. 5.16, and features a sphere moving in a prescribed motion around the objects. Attached to the other objects is a script that makes them face the sphere at all times. The simulation is not advanced, and isn't really a simulation per se, but functions well for a benchmark because of the shear number of interactive objects.

As a result the messages generated in this benchmark are larger than a single UDP datagram, generating results for this option requires a client that can handle multi-part datagrams. The current implementation was able to reconstruct messages with GPB payloads, as these requires fewer partial messages, but not those using JSON. This is why such results are not present in the presented figures.



**(a)** Internal **(b)** Browser **(c)** Unity

**Figure 5.16:** Simulation scene rendered in different implementations

### 5.3.1 Request-reply

In this benchmark, the client requests the simulation for 100 simulation updates. Each request is sent as soon as the previous has been received. No parsing of the message payload is performed.

Figure. 5.17 shows the performance of the various transport when using GPB to encode the payload data. Figure. 5.18 shows the same data when using JSON encoding.

Finally, a comparison of the total time required by each transport and encoding scheme is given in Figure. 5.19. Here we clearly see that overall GPB is much faster than JSON, which is natural due to the smaller size of the messages sent.

From the figures, we clearly notice that using GPB is more efficient w.r.t time. Overall, we notice that UDP is, not surprisingly, the fastest method of delivery. However, it is considerable harder to implement a client for it. We also notice that the WebSocket option in Unity3d is the slowest, while interestingly, the same protocol in the browsers are considerable more efficient, even though they are implemented in a dynamic language. Actually, when considering GPB payloads, Firefox is faster than the TCP/IP transport in Unity3d and performs similar to the ZeroMQ option. Here ZeroMQ shows the advantage of using a middleware. While ZeroMQ uses TCP/IP under the hood, it beats the "raw" TCP/IP implementation. Probably due to a number of optimization's in the library.

### 5.3.2 Publication

In this benchmark the server sends 1000 simulation updates to the client as quickly as possible. Compared to the first benchmark, the client does not send data back to the server between updates. The objective of the benchmark is to measure the throughput, which is the amount of data that can be transferred per unit time (Comer, 2008). In particular, we are measuring the *goodput*, which is the effective data rate achieved by the application. It is worth nothing that throughput is a measure of capacity, not speed (Comer, 2008).

As we are interested in benchmarking the network performance, the message payload is not parsed once received. Table. 5.5 shows the collected results.

**Table 5.5:** Measured throughput

| **Platform** | **Browser** | | | | **Unity** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Transport** | *ws-firefox* | | *ws-chrome* | | *ws-unity* | | *tcp-unity* | | *zmq-unity* | |
| **Encoding** | GPB | JSON | GPB | JSON | GPB | JSON | GPB | JSON | GPB | JSON |
| **Bytes transferred (MB)** | 81.23 | 196.4 | 81.23 | 196.4 | 81.23 | 196.4 | 81.23 | 196.4 | 81.23 | 196.4 |
| **Time elapsed (s)** | 0.721 | 4.06 | 4.28 | 4.85 | 3.39 | 9.619 | 0.738 | 4.633 | 1.424 | 3.969 |
| **Through-output (mbps)** | 900 | 387 | 152 | 324 | 191 | 164 | 880 | 339 | 456 | 395 |

From this table we see that the message size is 41.3% smaller when using GPB to encode the payload compared to JSON. Also, we notice that the time required to receive 1000 messages are lower using GPB for all transports.

Interestingly, the highest throughput is achieved in the browser, using Firefox. UDP is not listed, because it failed the benchmark. Basically it was not able to receive all of the sub-messages necessary to build complete messages due to unknown reasons.

Using the same setup, another run was performed with the purpose of measuring the parsing performance. The results can be viewed in Table. 5.6, which shows the mean time it takes to parse the message payloads in the Chrome, Firefox and Unity .

**Table 5.6:** Mean time to parse 1000 messages

| | Firefox | Chrome | Unity3d |
|---|---|---|---|
| GPB | 2.82ms | 1.54ms | 2.3ms |
| JSON | 2.92ms | 6.14ms | 53ms |

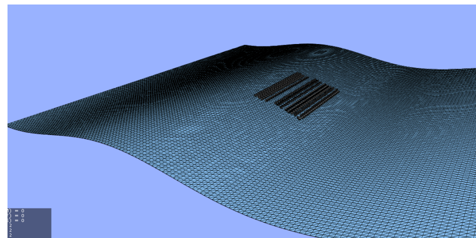
From the table, we see that all platforms require less time to parse messages with GPB encoded payloads compared to JSON. GPB under Chrome proves to be the fastest format to parse, while JSON in Chrome is almost four times as slow. Firefox parses JSON faster than Chrome, but GPB is almost twice as slow. As Unity3d uses a compiled language, it would be expected that it would be the faster option, however, GPB parsing under Unity3d lies in between Chrome and Firefox. This shows that, performance wise, code execution in the browsers can be very good.

The underwhelming performance of JSON parsing under Unity3d can be explained by the fact that the parsing is done by the GPB API, which converts the JSON message to an

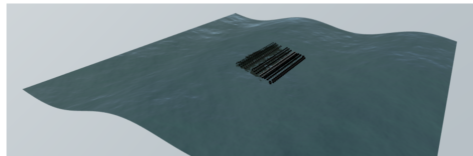
equal GPB representation. This process utilizes reflection, which is considerably slower than normal method invocations.

## 5.4 Synchronized Wave Visualization

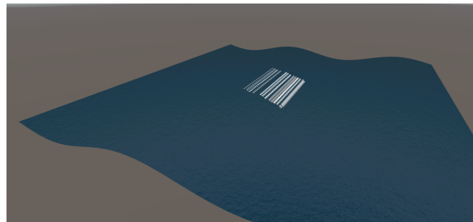
In this case study, a simulation scene featuring a dynamic ocean surface was created. Floating on top of the surface is multiple rigid-bodies with box shapes. A sinusoidal wave generator is used to generate the surface heights client-side, following the approach described in Section 5.1.5. The scene can be viewed in Figure. 5.20, rendered by different implementations at the same point in time.



**(a)** Internal



**(b)** Browser



**(c)** Unity3d

**Figure 5.20:** Waves synchronized between different render implementations

## 5.5 Multi-projector rendering

In this case study, the Unity3d client is used to render a simulation scene onto the cylindrical wall of the Visualization lab introduced in Section. 4.7. Normally, three computers - powering four projectors each, are used to fully cover the canvas area, but for this study only 2 projectors were used. Each powered by it's own computer. The system setup can be seen in Figure. 5.21. One of the clients are assigned the master role, publishing it's

transform to connected clients. The second client then subscribe to the master and updates it's own transform accordingly.

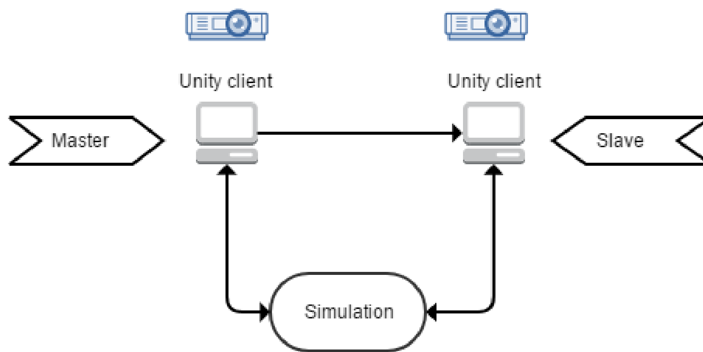



**Figure 5.21:** The test setup. Two clients were used, each powering a projector. One of the clients was assigned the master role, publishing it's camera transform to the second client acting as a slave. The simulation was running on a remote computer

Rolf-Magnus Hjørungdal, another master student, has developed a Unity3d asset that applies warping and blending effects to the camera, such that images from the different projectors does not overlap. This asset was imported to the implemented Unity3d client, and the result can be seen in Figure. 5.22. Some visual artifacts can be seen, due to the fact that the asset was not completely finished at the time of the test.

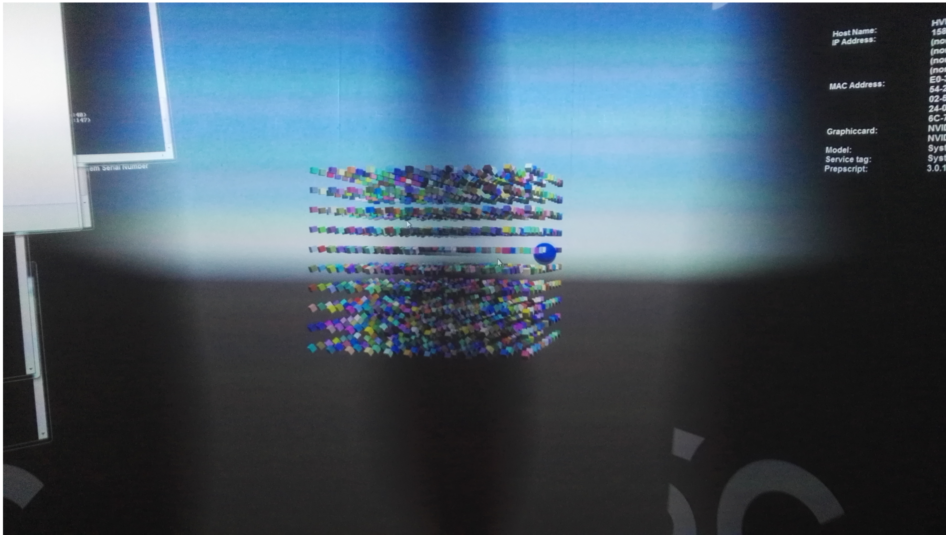
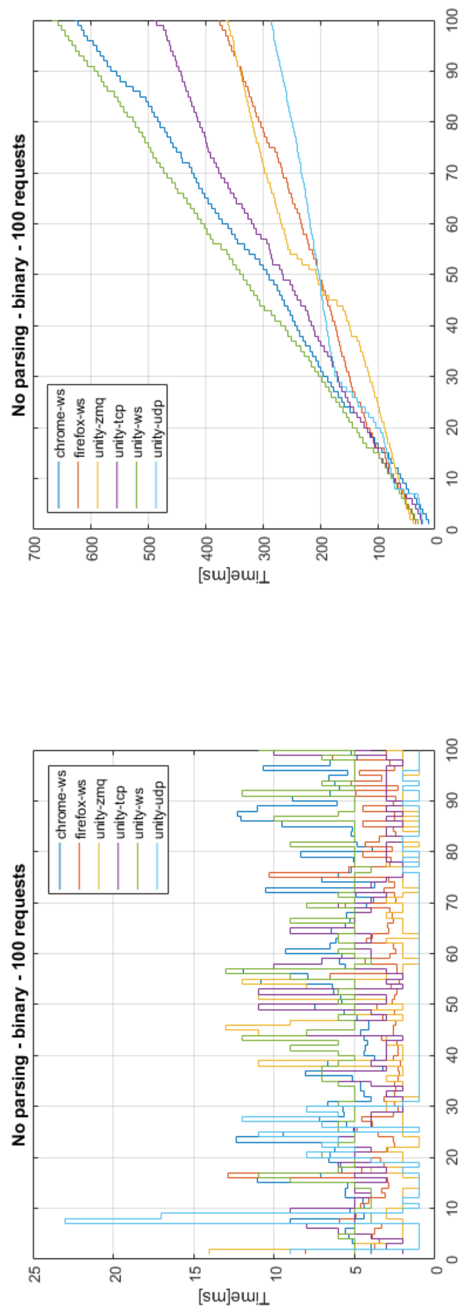



**Figure 5.22:** Scene rendered using two projectors in the Visualization lab. Warping and blending effects were achieved using master student Rolf-Magnus Hjørungdal's Unity3d asset.

The solution runs within the Unity3d editor. As this implies unwanted GUI elements being visible, a script was added that makes it possible to produce full-screen rendering output from within the editor. The script was written by reddit user *digitalsalmon* [3]. As we are not always interested in full-screen mode, logic was added so that the desired mode is read from a configuration file on play.

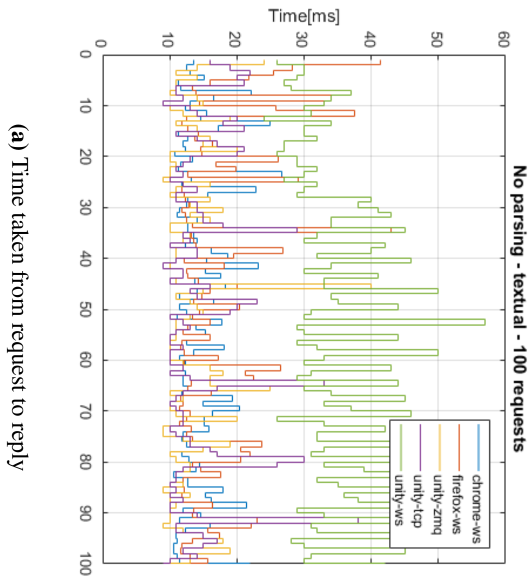
[3]https://www.reddit.com/r/Unity3D/comments/2lymim/full_full_screen_on_play_script_freebie_for/



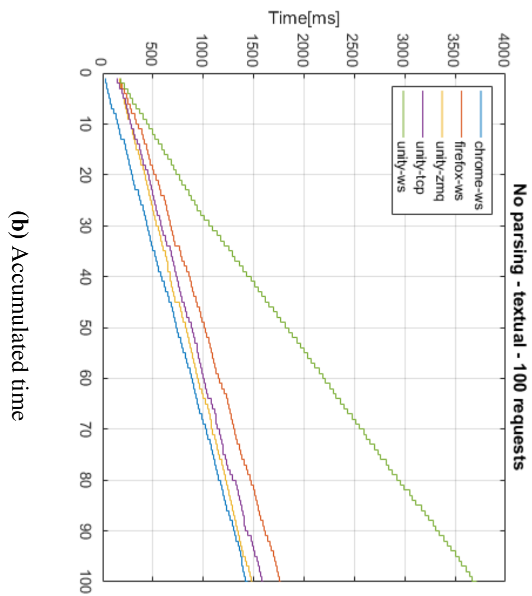

**(a)** Time taken from request to reply

**(b)** Accumulated time

**Figure 5.17:** 100 request-reply using binary encoding




**(a)** Time taken from request to reply




**(b)** Accumulated time

**Figure 5.18:** 100 request-reply using textual encoding

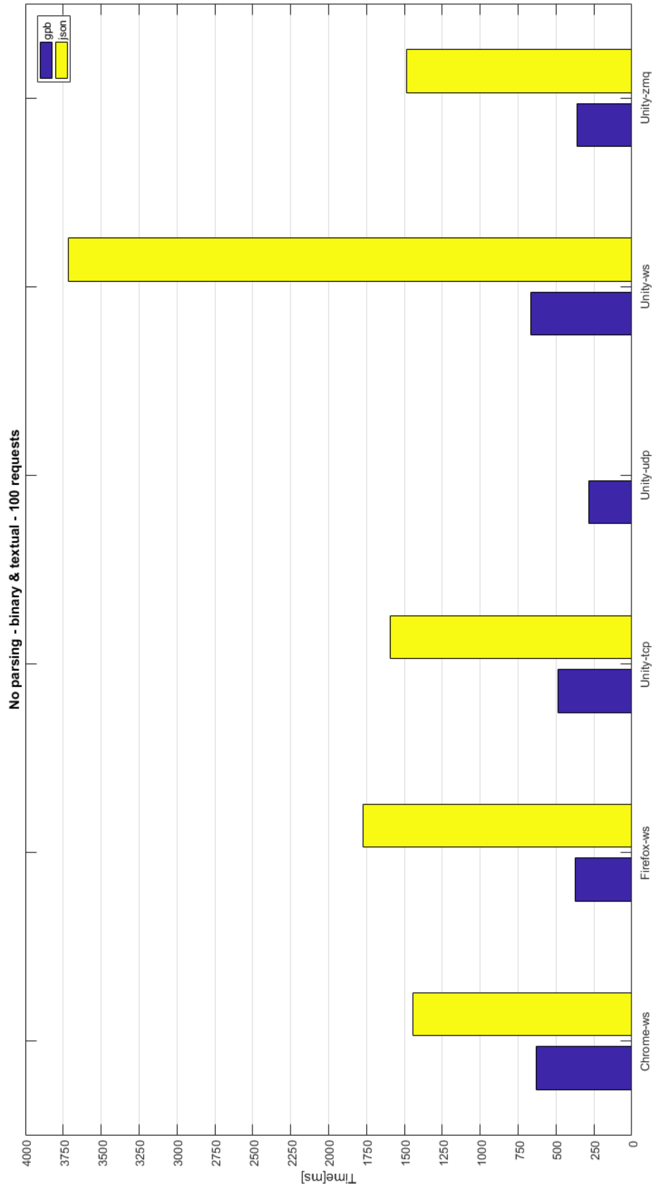



**Figure 5.19:** Time used to complete 100 request-reply for the various transport - both using GPB and JSON encoding of the payload data

Chapter 6

# Discussion

In this chapter key elements such as design choices and performance of the implemented solution is discussed. Also some ideas for future works are touched upon.

## 6.1 Wave visualization

The current approach used for implementation of wave visualization on remote clients, while better than sending raw height data, is not optimal. The reason is that the approach requires the client to know in advance the type of waves a simulation can produce, and implement those accordingly. GLTF (GL Transmission Format) (Khronos, 2017) is a new specification for efficient transmission and loading of 3D scenes and models for GL API's, developed by the Khronos Group. What sets GLTF apart from other similar formats, is that the shader is included in the model. This way, the model would only need to be implemented once. Also, since the model would be implemented in a shader, this solution would be very efficient rendering wise.

## 6.2 Industry standards

### 6.2.1 DDS

As DDS (see Section. 2.1.2 has become an industrial standard for data exchange, it was important to investigate whether or not it could be used in this project. Some of the findings are listed below:

1. Large API that requires a lot of configuration makes it difficult to use.
2. Only one open-source implementation. Others are commercial.
3. Coupled with it's own IDL. This is what makes DDS interoperable, as all nodes speak the same language out of the box.

4. Focuses on publish-subscribe. While request-reply has been added as an extension in the standard, it may not be supported by all implementations.

The conclusion was that DDS would not be considered to be used as a middleware connecting clients with the server. More than anything, DDS seems to identify its intended use in the realm of IoT devices. Furthermore, DDS seems difficult to use alongside other transports, as it's coupled with its own IDL. However, a possibility still exists for it to be used in the future to connect hardware devices with a simulation on the component side of things.

### 6.2.2 CIGI

As described in Section. 2.4, CIGI is a standard for communication between a simulation host and an IG. As such, CIGI was evaluated as a solution for realizing 3D rendering in remote clients. However, because the standard is quite large and packages data in binary form not backed by an schema, it was deemed somewhat unwieldy to implement. A goal of this project was that it should be easy for clients to implement the interface, and CIGI demands considerable effort on the part of the implementer. Both client side and host side. Furthermore, it seems that there is no open-source community around it. The current implementors are commercial, keeping their code closed-source.

Therefore, a more light-weight solution was built, based on GPB with the option to use JSON. However, thanks to the extensible nature of the implemented system, a CIGI implementation may be revisited in the future which could live side by side with the current solution, simply by adding a new handler dedicated for CIGI.

## 6.3 Performance

### 6.3.1 Networking

From the benchmark results in Section. 5.3 we see that Firefox delivers the best performance regardless of payload encoding. This is somewhat unexpected, as JavaScript is dynamically typed and WebSockets comes with a larger overhead compared to the options available under Unity3d and C#. It clearly shows that code executed in the browser can compete with traditional desktop applications performance wise. However, under the same circumstances, Chrome delivers the lowest throughout. While they both use JavaScript, Firefox and Chrome employs competing implementations. In this case, Firefox is clearly the fastest, and shows that users should test different browsers when interacting with resource demanding web pages.

Looking at Unity3d only, as seen in Table. 5.5 raw TCP is the more performant solution. This is not a huge surprise as both WebSockets and ZeroMQ uses TCP themselves, with some additional overheads involved.

The WebSocket solution in Unity3d is overall the worst performing. The added overhead does not explain this, as the browsers which also uses WebSockets perform better. As such, the lower performance is probably due to how the 3rd party library, *Websocket-sharp*, has been implemented.

In the future, it would be beneficial to include results from even more platforms/languages in order to have a better foundation for comparisons.

### 6.3.2 Serialization

In both the implemented clients, GBP is the more efficient format as shown in Table. 5.6. The difference is most notable in the Unity3d client, where the JSON messages will quickly slow down the code execution, because the JSON messages are converted from a JSON string to an equivalent GBP message. This process is slow because reflection is involved. The JSON serialization mechanism is meant to be used by dynamic languages, but was included in the Unity3d client for validation purposes. As noted, GPB is faster also in the dynamic browser environment, even though JSON is considered the de-facto standard in this environment and one would assume that significant efforts from the browser vendors has been put into optimizing it.

Even though messages using JSON encoding is proven to be slower and requires higher bandwidth than equivalent GBP messages, from a user perspective, it is still a useful option to have, as clients do not require a schema and JSON is objectively easier to implement because it is so commonly used.

## 6.4 Multi-display rendering

As shown in Section. 5.5 the solution can be used to render a simulation scene onto multiple connected displays. Compared to Unity3d's clustered rendering, this solution shows multiple advantages.

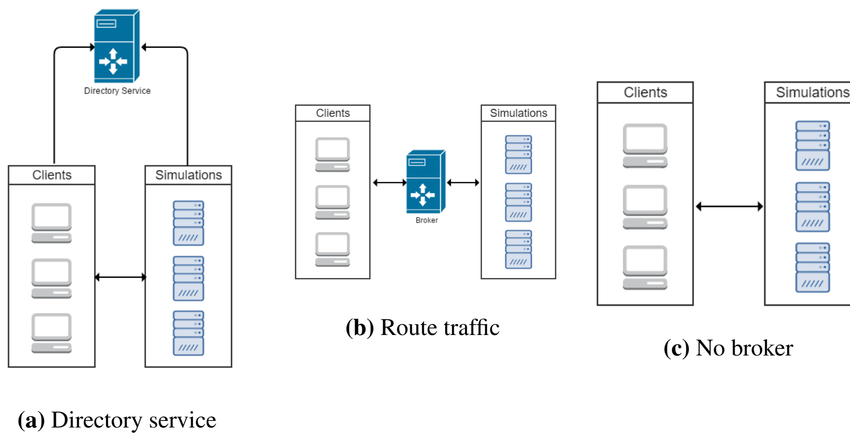
1. It is not locked to Unity3d. Any rendering application can in theory be used.
2. When using Unity3d, the solution does not need to be built. It can run directly from the editor.
3. No forced limitations to the number of clients.
4. Less complexity. Simulation runs on a single computer, no lock-stepping required.
5. The number of clients can vary while running.

## 6.5 Miscellaneous

### 6.5.1 Broker architecture

In the implemented system, clients communicate directly with the simulations after an initial query for available simulations to a broker acting as a directory service. Two alternate solutions could easily be identified:

1. **No broker:** Performance wise, this approach is equal to the current solution, as clients and simulations has a direct line of communications. However, clients cannot discover simulations. The necessary information needed to connect to a simulation




**(a)** Directory service

**(b)** Route traffic

**(c)** No broker

**Figure 6.1:** Design alternatives

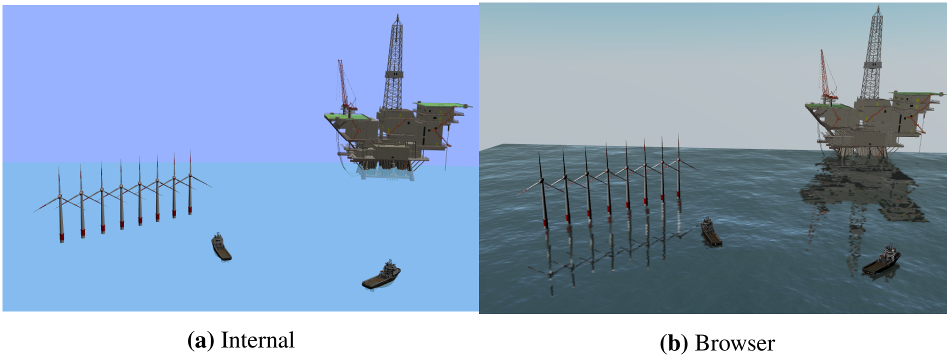
would have to be provided manually by the user, and the user itself would have to get the IP address and ports manually from the simulation host machine. This solution would be OK if the client and simulation machines where physically close to each other, or in cases where the network address of a simulation would not change and clients could be hard-coded with the appropriate addresses. In more complex scenarios, however, this solution would prove a logistically nightmare.

2. **Route all traffic through broker:** In this case, automatic discovery of simulations would be available, as the broker would act as a directory service, and additionally it would handle traffic routing. All traffic would pass through the application. Only a single transport mechanism between broker and simulation would be necessary in this case. Support for other transports client side could be integrated into the broker itself. Other properties of such a solution would be that clients, in addition to the knowing the broker address, would only need some unique identifier to talk to simulations. The big downside of this approach is that communications would naturally be slower. Additionally, with enough data traffic the broker could run out of bandwidth or slow down due to excessive CPU load, slowing down every connected system.

The different architectures are shown in Figure. 6.1.

### 6.5.2 UDP support

Because UDP has a low overhead, supports multi-cast and dropped messages are discarded, it is highly suitable for real-time communication. While a UDP support were implemented for this system, the client side implementation for Unity3d is somewhat lacking. Because the messages sent from the server, when serving medium to large simulation scenes, will exceed the maximum message size for a single UDP datagram, the message must be divided and sent as multi-part messages. This is done server side, but the client



**(a)** Internal **(b)** Browser

**Figure 6.2:** Example of simulation scene with multiple 3D models

side implementation is not able to rebuild multi-part messages properly. Thus, using the implemented client, only simulations that generates messages smaller than 65 kilobyte are handled. Rebuilding a UDP messages is an complex issue, especially when the data stream is continuous, because the messages can appear in the wrong order or they may not appear at all. As RakNet (see Section. 2.1.2) is able to deliver reliable UDP transport, a viable strategy might be to add RakNet to the list of supported end-points in the future.

### 6.5.3 Loading 3D models

When the source of a 3D model is an actual file, either located on the Internet or on a remote file system, the 3D model is, as described in 5.1.4, presented to the client as a link. Allowing a client to load the 3D models at a later time, thus reducing load times. Furthermore, models can be cached, which greatly improves load times in cases a model is re-used by other objects, as it only need to be downloaded once. Both these features are used by the web client, making it very effective at loading complex scenes that contains multiple 3D models - as exemplified in Figure. 6.2. The Unity3d client, however, lacks both and is therefore very ineffective at loading scenes with multiple 3D models. For each 3D model, the current implementation:

1. Downloads the model.
2. Writes the model + materials/textures to file.
3. Loads the model from file.
4. Deletes the file from the file system.

While this works, users might mistake the slowness for a system failure and should be improved by adding a caching mechanism in the future.

### 6.5.4 Simulation playback

As the project progressed, it was made clear that the messages designed for 3D rendering on remote clients could have a second use. By storing the same messages to a file, playback

functionality could be implemented. This allows Vico simulations to be played back in an interactive viewer. This is a very useful feature, especially for slower than real-time simulations which could otherwise prove unsuited for interactive 3D rendering.

Chapter 7

# Conclusion

In this thesis a flexible network interface for a real-time simulation framework was presented. The solution is flexible in that it allows clients to choose among several different transport protocols, increasing the chance that a client can be implemented in language *X* or tool *Y*. In particular, the implemented transports are the TCP/IP, UDP and WebSocket protocols as well as the ZeroMQ middleware. Additionally, HTTP is used to serve static files.

Borrowing the terminology used by (Byrne et al., 2010) for Web-based simulation and applying it to simulations made accessible over the network in general, what has been presented is a *hybrid* architecture as described in Section. 3.1. Simulations runs on a server, while clients accessing the simulation using the developed interface handles visualization/interaction.

Messages received server-side are handled by a modular interface. Unique keys leading a message are used to route messages to the appropriate handler. The implemented system comes bundled with a number of handlers, such as handlers for 3D visualization, plotting and keyboard input. Additionally, new handlers can be seamlessly integrated by simulation designers thanks to the modular and extensible design. It was shown how clients making use of the implemented handlers were able to render simulations scenes and plot data in real-time. Even ocean waves, synchronized across clients, can be rendered.

Furthermore, clients can choose how data payloads are sent/received over the wire. Both the textual JSON format and the binary GPB format are supported for static message types, such as messages for 3D rendering and plotting. This mechanism is also extensible, such that alternate formats can be added in the future. In order to facilitate special requirements, a special handler makes it possible for annotated Java methods to be exposed dynamically over the network using JSON RPC 2.0 compliant messages.

The usefulness of the system is shown through implementation of clients on two different platforms, web and desktop. Using different transport mechanisms, clients are able to interface against the same simulations simultaneously. This makes it possible for clients, even on different platform, to engage in virtual collaboration. Once a client for the system has been implemented, it was shown through a demonstration in the Visualization lab that

clustered rendering can be facilitated.

As mentioned in the previous chapter, a number of future works should be considered in order to further improve the system. Supporting RakNet would provide a way of delivering reliable UPD messages, which could increase network performance for desktop applications. While the Unity3d client is working, it is very slow at loading scenes with multiple 3D models, because it lacks a caching mechanism. This should be implemented. It would also be nice to see a client implemented for Unreal Engine 4, which could provide better performance and graphics than what Unity3d can deliver. In order to make it easier to implement ocean waves across clients, the GLTF format should be looked into. Finally, in order to support industrial IG's, support for CIGI as an addition to the custom messages for rendering introduced in this work should be considered.

# Bibliography

Bainbridge, W. S., 2007. The scientific research potential of virtual worlds. science 317 (5837), 472–476.

Bernier, Y. W., 2001. Latency compensating methods in client/server in-game protocol design and optimization. In: Game Developers Conference. Vol. 98033.

Bittner, S., Oelsner, O., Neidhold, T., 2015. Using fmi in a cloud-based web application for system simulation. In: Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015. No. 118. Linköping University Electronic Press, pp. 845–848.

Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., et al., 2012. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In: Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany. No. 076. Linköping University Electronic Press, pp. 173–184.

Buschmann, F., Henney, K., Schmidth, D., 1999. A pattern language for distributed computing: Pattern-oriented software architecture.

Byrne, J., Heavey, C., Byrne, P. J., 2010. A review of web-based simulation and supporting tools. Simulation modelling practice and theory 18 (3), 253–276.

Cabello, R., 2010. Three. js. `https://github.com/mrdoob/three.js`.

Chen, B., Xu, Z., 2011. A framework for browser-based multiplayer online games using webgl and websocket. In: Multimedia Technology (ICMT), 2011 International Conference on. IEEE, pp. 471–474.

Comer, D. E., 2008. Computer networks and internets. Prentice Hall Press.

Dworak, A., Ehm, F., Charrue, P., Sliwinski, W., 2012. The new cern controls middleware. In: Journal of Physics: Conference Series. Vol. 396. IOP Publishing, p. 012017.

Dworak, A., Sobczak, M., Ehm, F., Sliwinski, W., Charrue, P., 2011. Middleware trends and market leaders 2011. In: Conf. Proc. Vol. 111010. p. FRBHMULT05.

fenopix, 2013. Canvas. js. `http://canvasjs.com/`.

Fiedler, G., 2010. What every programmer needs to know about game networking. URL `http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/`

Fishwick, P. A., 2009. An introduction to opensimulator and virtual environment agent-based m&s applications. In: Simulation conference (WSC), proceedings of the 2009 winter. IEEE, pp. 177–183.

Forouzan, B. A., 2002. TCP/IP protocol suite. McGraw-Hill, Inc.

Fortmann-Roe, S., 2014. Insight maker: A general-purpose tool for web-based modeling & simulation. Simulation Modelling Practice and Theory 47, 28–45.

Group, J.-R. W., et al., 2012. Json-rpc 2.0 specification.

Halic, T., Ahn, W., De, S., 2011. A framework for 3d interactive applications on the web. In: SIGGRAPH Asia 2011 Posters. ACM, p. 58.

Hatledal, L. I., Schaathun, H. G., Zhang, H., 2015. A software architecture for simulation and visualisation based on the functional mock-up interface and web technologies. In: Proceedings of the 56th Conference on Simulation and Modelling (SIMS 56), October, 7-9, 2015, Linköping University, Sweden. No. 119. Linköping University Electronic Press, pp. 123–129.

Hintjens, P., 2013. ZeroMQ: Messaging for Many Applications. " O'Reilly Media, Inc.".

JeroMQ, 2017. Jeromq. `https://github.com/zeromq/jeromq`.

Khronos, 2017. Gltf. `https://github.com/KhronosGroup/glTF`.

Loreto, S., Saint-Andre, P., Salsano, S., Wilkins, G., 2011. Known issues and best practices for the use of long polling and streaming in bidirectional http. Tech. rep.

Maeda, K., 2012. Performance evaluation of object serialization libraries in xml, json and binary formats. In: Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on. IEEE, pp. 177–182.

Marrin, C., 2011. Webgl specification. Khronos WebGL Working Group.

McMullen, T., Hawick, K., Du Preez, V., Pearce, B., 2012. Graphics on web platforms for complex systems modelling and simulation. In: Proceedings of the International Conference on Computer Graphics and Virtual Reality (CGVR). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p. 1.

Mozilla, 2017. Webgl. `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`.

Müller, J., Lorenz, M., Geller, F., Zeier, A., Plattner, H., 2010. Assessment of communication protocols in the epc network-replacing textual soap and xml with binary google protocol buffers encoding. In: Industrial Engineering and Engineering Management (IE&EM), 2010 IEEE 17Th International Conference on. IEEE, pp. 404–409.

Neumann, C., Prigent, N., Varvello, M., Suh, K., 2007. Challenges in peer-to-peer gaming. ACM SIGCOMM Computer Communication Review 37 (1), 79–82.

Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C., 2009. Comparison of json and xml data interchange formats: A case study. Caine 2009, 157–162.

Obidowski, R. M., Jha, R., 2010. Advances in scalable generic image generator technology for the advanced deployable day/night simulation project. In: Vision and Displays for Military and Security Applications. Springer, pp. 75–85.

Oculus, 2014. Raknet. `https://github.com/facebookarchive/RakNet`.

OMG, march 2016. Remote procedure call over dds.
URL `http://www.omg.org/spec/DDS-RPC/1.0/Beta2/PDF`

Oracle, 2015. Trail: The reflection api. `http://docs.oracle.com/javase/tutorial/reflect/index.html`.

Pang, X., Dye, R., Nouidui, T. S., Wetter, M., Deringer, J. J., 2013. Linking interactive modelica simulations to html5 using the functional mockup interface for the learnhpb platform. In: Proc. of the 13th IBPSA Conference, Chambery, France. pp. 2823–2829.

Parisi, T., 2012. WebGL: up and running. " O'Reilly Media, Inc.".

Phelps, W., 2002. Interface control document for the common image generator interface (cigi) version 2.0. Boeing Corporation, St. Louis, MO. Report TST02I015, available June 1009.

Postel, J., 1980. User datagram protocol. Tech. rep.

Rajlich, N., 2017. Java-websockets. `http://tootallnate.github.io/Java-WebSocket/`.

Reis, B., Teixeira, J. M., Kelner, J., 2011. An open-source tool for distributed viewing of kinect data on the web. In: VIII WORKSHOP DE REALIDADE VIRTUAL E AUMENTADA–2011. Disponível em:¡ https://www. gprt. ufpe. br/grvm/Publication/-FullPapers/201. Vol. 1.

Rizano, T., Abeni, L., Palopoli, L., 2013. Experimental evaluation of the real-time performance of publish-subscribe middlewares.

Rymaszewski, M., 2007. Second life: The official guide. Vol. 2. John Wiley & Sons.

Schollmeier, R., 2001. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: Peer-to-Peer Computing, 2001. Proceedings. First International Conference on. IEEE, pp. 101–102.

Schwaber, K., Beedle, M., 2002. Agile software development with Scrum. Vol. 1. Prentice Hall Upper Saddle River.

SISO, 2012. Standard for common image generator interface (cigi).

Walker, J. D., Chapra, S. C., 2014. A client-side web application for interactive environmental simulation modeling. Environmental Modelling & Software 55, 49–60.

Wiedemann, T., 2001. Simulation application service providing (sim-asp). In: Proceedings of the 33nd conference on Winter simulation. IEEE Computer Society, pp. 623–628.

Wirtz, D., 2016. Protobuf. js. `http://dcode.io/protobuf.js/`.

ZeroMQ, 2012. Broker vs. brokerless. `http://zeromq.org/whitepapers:brokerless`.

# Appendix

## Code fragments

### Protocol Buffer Schemas

**Listing 7.1:** CommonProto.proto

```
syntax = "proto3";

option csharp_namespace = "Vico.Unity";
option java_package = "no.ntnu.vico.proto";


message PBSimulationSetup {

    string name = 1;
    string uuid = 2;
    PBTime time = 3;
    PBEntity root = 4;

}

message PBSimulationUpdate {

    PBTime time = 1;
    PBEntityUpdate root = 2;

}

message PBTime {

    uint64 iterations = 1;
    float simulationTime = 2;
    float runTime = 3;
    float timeStep = 4;

}

message PBEntity {

    string uuid = 1;
    string name = 2;
    PBTransform transform = 3;
    repeated PBGeometry geometries = 4;
    repeated PBCurve curves = 5;
    repeated PBEntity children = 6;
    repeated PBComponent components = 7;

}

message PBEntityUpdate {

    string uuid = 1;
    PBTransform transform = 2;
    repeated PBCurveUpdate curves = 3;
    repeated PBEntityUpdate children = 4;
    repeated PBComponent components = 5;

}

message PBTransform {

    PBVec3 localPosition = 1;
    PBQuat localQuaternion = 2;

}
```

```
message PBCurve {

    string uuid = 1;
    float radius = 2;
    PBColor color = 3;
    repeated PBVec3 points = 4;

}

message PBCurveUpdate {

    string uuid = 1;
    repeated PBVec3 points = 2;

}


message PBEntityAdded {

    PBEntity entity = 1;

}

message PBEntityRemoved {

    string uuid = 1;

}

message PBComponent {

    string type = 1;
    string data = 2;

}


//////////////////MATH//////////////////////

message PBVec3 {

        float x = 1;
        float y = 2;
        float z = 3;

}

message PBQuat {

        float x = 1;
        float y = 2;
        float z = 3;
        float w = 4;

}

message PBColor {

    uint32 r = 1;
    uint32 g = 2;
    uint32 b = 3;
    uint32 a = 4;

}

message PBColors {

        repeated PBColor value = 1;

}

////////////////////GEOMETRY////////////////

message PBGeometry {

    oneof these {
                PBColor color = 1;
                PBExternalSource texture = 2;
        }

    PBVec3 offsetPosition = 3;
    PBQuat offsetQuaternion = 4;
        PBBoundingBox boundingBox = 5;
    bool isCollision = 6;
    bool isVisual = 7;

    oneof shape {
        PBBox box = 21;
        PBPlane plane = 22;
        PBHeightField heightField = 23;
```

```
        PBSphere sphere = 24;
        PBCylinder cylinder = 25;
        PBCapsule capsule = 26;
        PBMesh mesh = 27;
    }
}

message PBBoundingBox {

        PBVec3 center = 1;
    PBVec3 halfExtents = 2;

}

message PBLine {

        PBVec3 start = 1;
        PBVec3 end = 2;

}

message PBBox {

    float x = 1;
    float y = 2;
    float z = 3;

}

message PBPlane {

    float width = 1;
    float height = 2;

}

message PBHeightField {

    float width = 1;
    float height = 2;
    uint32 widthSegments = 3;
    uint32 heightSegments = 4;
    repeated float heights = 5;

}

message PBSphere {

    float radius = 1;

}

message PBCylinder {

    float radius = 1;
    float height = 2;

}

message PBCapsule {

    float radius = 1;
    float height = 2;

}

message PBPointCloud {

        repeated PBVec3 points = 1;

        oneof these {
                PBColor color = 2;
                PBColors colors = 3;
        }

}

message PBMesh {

    float scale = 1;

    oneof these {
        PBRawMeshData raw = 11;
        PBExternalSource source = 12;
    }

}

message PBRawMeshData {
```

```
    repeated uint32 indices = 1;
    repeated float vertices = 2;
    repeated float normals = 3;

}

message PBExternalSource {

        string extension = 1;
        string location = 2;
        string baseName = 3;
        string fullPath = 4;

}
```

**Listing 7.2:** RequestProto.proto

```
syntax = "proto3";

option csharp_namespace = "Vico.Unity";
option java_package = "no.ntnu.vico.proto";

import "CommonProto.proto";

message Request {
    RequestType type = 1;
}

enum RequestType {

    SIMULATION_SETUP = 0;
    SIMULATION_UPDATE = 1;

}

message Response {

    oneof these {
        PBSimulationSetup simulationSetup = 1;
        PBSimulationUpdate simulationUpdate = 2;
    }

}
```

**Listing 7.3:** SubscriptionProto.proto

```
syntax = "proto3";

option csharp_namespace = "Vico.Unity";
option java_package = "no.ntnu.vico.proto";

import "CommonProto.proto";

message Publication {

    oneof these {

        PBSimulationUpdate simulationUpdate = 1;
        PBEntityAdded entityAdded = 2;
        PBEntityRemoved entityRemoved = 3;

    }

}
```

**Listing 7.4:** PlotProto.proto

```
syntax = "proto3";

option csharp_namespace = "Vico.Unity";
option java_package = "no.ntnu.vico.proto";

message PlotRequest {

    oneof these {
        RequestAvailablePlots availableRequest = 1;
        RequestPlotSetup setupRequest = 2;
        RequestPlotUpdate updateRequest = 3;
```

```
    }

}

message PlotResponse {

    oneof these {
        AvailablePlots available = 1;
        PlotSetup setup = 2;
        PlotUpdate update = 3;
    }

}

message AvailablePlots {
    repeated PlotInfo plots = 1;
}

message PlotInfo {

    string uuid = 1;
    string name = 2;

}

message RequestAvailablePlots {
    //intentionally left empty
}

message PlotSetup {
    string xLabel = 1;
    string yLabel = 2;
    repeated string legend = 3;
    Range range = 4;
}

message PlotUpdate {
    bool hasData = 1;
    float x = 2;
    repeated float y = 3;
}

message RequestPlotSetup {
    string uuid = 1;
}

message RequestPlotUpdate {
    string uuid = 2;
}

message Range {
    float min = 1;
    float max = 2;
}
```

**Listing 7.5:** SpawnProto.proto

```
syntax = "proto3";

option csharp_namespace = "Vico.Unity";
option java_package = "no.ntnu.vico.proto";

import "CommonProto.proto";

message PBSpawnObject {

    string parentUuid = 1;
    PBVec3 position = 2;
    PBVec3 velocity = 3;
    PBQuat quaternion = 4;
    oneof shape {
        PBBox box = 10;
        PBSphere sphere = 11;
        PBCylinder cylinder = 12;
        PBCapsule capsule = 13;
    }

}
```

# Unity3d

**Listing 7.6:** Master.cs

```
using System.Threading;
using ZeroMQ;
using System;
using System.Text;
using Vico.Unity;
using Newtonsoft.Json;
using Google.Protobuf;

public class Master : MonoBehaviour
{

    struct MasterMsg {
        public Vector3 position;
        public Quaternion quaternion;
    }


    private Thread thread;
    private ZSocket publisher;
    private Queue<MasterMsg> queue = new Queue<MasterMsg>();


    public int port = -1;
    private bool stopThread;

    void Start()
    {

        thread = new Thread(PublicationThread);
        thread.Start();
        Debug.Log("Master started!");
    }

    // Update is called once per frame
    void Stop()
    {
        stopThread = true;
        thread.Join();
        Debug.Log("Master stopped!");
    }

    void FixedUpdate()
    {

        if (queue.Count == 0)
        {

                MasterMsg msg = new MasterMsg ();
                msg.position = transform.position;
                msg.quaternion = transform.rotation;

            queue.Enqueue(msg);
        }

    }

    private void PublicationThread()
    {

        using (var context = new ZContext())
        {
            using (publisher = new ZSocket(context, ZSocketType.PUB))
            {

                if (port == -1)
                {
                    publisher.Bind("tcp://*:*");

                    byte[] address = new byte[1024];

                    publisher.GetOption(ZSocketOption.LAST_ENDPOINT, out address);
                    char[] c = { ':' };
                    var split = Encoding.UTF8.GetString(address).Split(c);
                    Array.Reverse(split);
                    port = Convert.ToInt32(split[0]);
                }
                else
                {
                    publisher.Bind("tcp://*:" + port);
                }

                publisher.SetOption(ZSocketOption.SNDHWM, 1);
```

```
                while (!stopThread)
                {

                    if (queue.Count > 0)
                    {
                        MasterMsg msg = queue.Dequeue();

                        Vec3Msg v = new Vec3Msg();
                        v.X = msg.position.x;
                        v.Y = msg.position.y;
                        v.Z = msg.position.z;

                        QuatMsg q = new QuatMsg ();
                        q.X = msg.quaternion.x;
                        q.Y = msg.quaternion.y;
                        q.Z = msg.quaternion.z;
                        q.W = msg.quaternion.w;

                        publisher.SendMore(new ZFrame(v.ToByteArray()));
                        publisher.Send (new ZFrame (q.ToByteArray ()));

                    }

                    Thread.Sleep(5);

                }
            }
        }
    }

    void OnApplicationQuit()
    {
        Stop();

    }
}
```

**Listing 7.7:** Slave.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Threading;
using Vico.Unity;
using ZeroMQ;
using Google.Protobuf;
using System;

public class Slave : MonoBehaviour {

    public string hostAddress = "localhost";
    public int port = -1;

    private Thread thread;

    private Vector3 masterPos;
    private Quaternion masterQuat;

    private bool stopThread;


    void Start () {

       if (port != -1)
        {
            thread = new Thread(SubscriptionThread);
            thread.Start();
            Debug.Log("Slave started!");
        }

    }

    void Stop()
    {
        if (thread != null)
        {
            stopThread = true;
            thread.Join();
            Debug.Log("Slave stopped!");
        }
    }

    // Update is called once per frame
    void Update () {

        if (masterPos != null)
```

```
        {
            transform.position = masterPos;
                transform.rotation = masterQuat;
        }

    }

    void SubscriptionThread()
    {

        using (var context = new ZContext())
        {
            using (var subscriber = new ZSocket(context, ZSocketType.SUB))
            {

                subscriber.SetOption(ZSocketOption.RCVTIMEO, 100);
                subscriber.Connect("tcp://" + hostAddress + ":" + port);
                subscriber.SubscribeAll();


                while (!stopThread)
                {

                    try {
                        using (var msg = subscriber.ReceiveMessage())
                        {
                            Vec3Msg v =  Vec3Msg.Parser.ParseFrom(msg.Pop().Read());
                            masterPos = new Vector3(v.X, v.Y, v.Z);

                            QuatMsg q =  QuatMsg.Parser.ParseFrom(msg.Pop().Read());
                            masterQuat = new Quaternion(q.X, q.Y, q.Z, q.W);

                        }
                    } catch (Exception ex)
                    {
                        Debug.Log("RCVTIMEO?");
                    }
                }
            }

        }

    }

    void OnApplicationQuit()
    {
        Stop();
    }

}
```

**Listing 7.8:** STLLoader.cs

```
public class STLLoader {


    #if UNITY_EDITOR
    [MenuItem("GameObject/Import From STL")]
    static void StlLoadMenu()
    {
        string pth = UnityEditor.EditorUtility.OpenFilePanel("Import STL", "", "stl");
        if (!string.IsNullOrEmpty(pth))
        {
            LoadSTLFile(pth);
        }
    }
    #endif

    public static GameObject LoadSTLFile(string fn)
    {

        string meshName = Path.GetFileNameWithoutExtension(fn);
        byte[] data = File.ReadAllBytes(fn);

        return LoadSTL(meshName, data);

    }

    public static GameObject LoadSTL(string name, byte[] data)
    {
        int faces;
        List<Vector3> vertices = new List<Vector3>();
        List<Vector3> normals = new List<Vector3>();


        using (BinaryWriter bw = new BinaryWriter(new MemoryStream()))
        {
```

```
            bw.Write(data);

            using (BinaryReader br = new BinaryReader(bw.BaseStream))
            {
                br.BaseStream.Position = 80;
                faces = br.ReadInt32();

                int dataOffset = 84;
                int faceLength = 12 * 4 + 2;


                for (int face = 0; face < faces; face++)
                {

                    int start = dataOffset + face * faceLength;

                    br.BaseStream.Position = start;
                    float normalX = br.ReadSingle();
                    br.BaseStream.Position = start + 4;
                    float normalY = br.ReadSingle();
                    br.BaseStream.Position = start + 8;
                    float normalZ = br.ReadSingle();

                    for (int i = 1; i <= 3; i++)
                    {

                        int vertexstart = start + i * 12;
                        br.BaseStream.Position = vertexstart;
                        float x = br.ReadSingle();
                        br.BaseStream.Position = vertexstart + 4;
                        float y = br.ReadSingle();
                        br.BaseStream.Position = vertexstart + 8;
                        float z = br.ReadSingle();

                        vertices.Add(new Vector3(x, y, z));
                        normals.Add(new Vector3(normalX, normalY, normalZ));

                    }
                }

            }

        }

        int[] indices = new int[faces * 3];
        for (int i = 0; i < indices.Length; i++)
        {
            indices[i] = i;
        }

        GameObject gameObject = new GameObject();
        if (name != null)
        {
            gameObject.name = name;
        }

        MeshFilter mf = gameObject.AddComponent<MeshFilter>();
        MeshRenderer renderer = gameObject.AddComponent<MeshRenderer>();
        renderer.materials = new Material[] { new Material(Shader.Find("Standard (Specular setup)")) };
        Mesh mesh = new Mesh();
        mf.mesh = mesh;

        mesh.vertices = vertices.ToArray();
        mesh.normals = normals.ToArray();
        mesh.triangles = indices;

        return gameObject;
    }

    public static GameObject LoadSTLFromRemote(string host, int httpPort, Vico.Unity.PBExternalSource source) {

        GameObject gameObject = null;
        using (WebClient client = new WebClient ()) {

            try {
                byte[] data = client.DownloadData ("http://" + host + ":" + httpPort + "/servlet?source=" + source.
                    FullPath);
                return LoadSTL(source.BaseName, data);

            } catch (WebException ex) {
                Debug.LogError (ex);
            }

        }

        return gameObject;

        }

}
```

**Listing 7.9:** RemoteOBJLoader.cs

```
public class RemoteOBJLoader {

    public static GameObject LoadFromRemote(string host, int httpPort, Vico.Unity.ExternalSourceMsg source) {

        DirectoryInfo dir = Directory.CreateDirectory (Application.temporaryCachePath + "/obj_" + Guid.NewGuid().
              ToString());
        string objFile = dir.FullName + "/" + source.BaseName + "." + source.Extension;
        string mtlFile = dir.FullName + "/" + source.BaseName + ".mtl";

        GameObject gameObject = null;
        using (WebClient client = new WebClient ()) {

            try {
                byte[] data = client.DownloadData ("http://" + host + ":" + httpPort + "/servlet?source=" + source.
                      FullPath);
                File.WriteAllBytes (objFile , data);

                data = client.DownloadData(("http://" + host + ":" + httpPort + "/servlet?source=" + source.FullPath).
                      Replace(".obj", ".mtl"));
                if (data != null) {
                    File.WriteAllBytes (mtlFile , data);
                    List<string> textures = TextureLocations(mtlFile);
                    foreach (string tex in textures) {
                        data = client.DownloadData(("http://" + host + ":" + httpPort + "/servlet?source=" + source.
                              Location) + tex);
                        File.WriteAllBytes (dir.FullName + "/" + tex , data);
                    }
                }
            } catch (WebException ex) {
                Debug.Log(ex);
            }

            if (File.Exists(objFile)) {
                gameObject = OBJLoader.LoadOBJFile(objFile);
            }

        }

        Directory.Delete (dir.FullName, true);

        return gameObject;

    }


    private static List<string> TextureLocations(string mtl) {

        List<string> list = new List<string> ();
        if (File.Exists (mtl)) {
            foreach (var line in File.ReadAllLines(mtl)) {
                if (line.Contains ("map_Kd")) {
                    string[] split = line.Split (new char[]{' '});
                    list.Add (split [1]);
                }
            }
        }
        return list;

    }

}
```

**Listing 7.10:** AbstractClient.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using Vico.Unity;
using Google.Protobuf;

public abstract class AbstractClient {

        private Action<Publication> pubCallback;
        private Dictionary<string, Action<Response>> reqCallbacks;

        protected string hostAddress;
        protected int port;

        private bool isSubscribing;

        internal bool noParsing;

        public AbstractClient(string hostAddress, int port) {
                this.hostAddress = hostAddress;
```

```
        this.port = port;
        this.reqCallbacks = new Dictionary<string, Action<Response>> ();
}

public void Start() {
        OnStart ();
}

public void Stop() {
        if (isSubscribing) {
                Unsubscribe ();
        }
        OnStop ();
}

protected abstract void OnStart();

protected abstract void OnStop ();

protected abstract void Send(VicoMsg msg);

protected void onMsg (VicoMsg msg) {

try {

    using (msg) {

        byte type = msg.PopByte();
        switch(type) {
            case 0x000: {

                string guid = msg.PopString();
                Action<Response> callback;
                reqCallbacks.TryGetValue(guid, out callback);

                if (callback != null) {
                        reqCallbacks.Remove (guid);
                        if (!noParsing) {
                        byte format = msg.PopByte ();
                        Response response = null;
                        if (format == 0x000) {
                            response = Response.Parser.ParseFrom (msg.PopData ());
                        } else if (format == 0x001) {
                            string json = msg.PopString ();
                            response = Response.Parser.ParseJson (json);
                        }

                        if (response != null) {
                            callback (response);
                        }
                        } else {
                                callback (null);
                        }
                }
                }
                break;
            case 0x001: {
                if (pubCallback != null) {
                        byte format = msg.PopByte();
                        Publication pub = null;
                        if (format == 0x000) {
                                pub = (Publication.Parser.ParseFrom(msg.PopData()));
                        } else if (format == 0x001) {
                                pub = (Publication.Parser.ParseJson(msg.PopString()));
                        }

                        if (pub != null) {
                                pubCallback(pub);
                        }
                }
                }
                break;
        }

    }

} catch (Exception ex) {
        Debug.Log (ex);
}

}

public void Request(Request request, Action<Response> callback, bool binary = true) {

string guid = Guid.NewGuid ().ToString ();
reqCallbacks [guid] = callback;

VicoMsg msg = new VicoMsg ();
msg.Add (0x000);
msg.Add (guid);
```

```
        msg.Add ((byte) (binary ? 0x000 : 0x001));
        if (binary) {
            msg.Add (request.ToByteArray());
        } else {
            msg.Add (JsonFormatter.Default.Format (request));
        }
        Send (msg);

        }

        public void Subscribe(Action<Publication> callback, bool binary = true) {

                pubCallback = callback;
                isSubscribing = true;

                VicoMsg msg = new VicoMsg ();
                msg.Add (0x001);
                msg.Add ("subscribe");
                msg.Add ((byte) (binary ? 0x000 : 0x001));
                Send (msg);

        }


        public void Unsubscribe() {

                isSubscribing = false;

                VicoMsg msg = new VicoMsg ();
                msg.Add (0x001);
                msg.Add ("unsubscribe");
                Send (msg);

        }


    public void KeyPressed(bool binary = true)
    {
        List<Vico.Unity.PBKey> keys = UnityKeyToVicoKey.Get();
        if (keys != null && keys.Count > 0)
        {

            VicoMsg msg = new VicoMsg();
            msg.Add(0x002);
            msg.Add((byte)(binary ? 0x000 : 0x001));

            PBKeyPressed keyPressed = new PBKeyPressed();
            keyPressed.Keys.Add(keys);

            if (binary)
            {
                msg.Add(keyPressed.ToByteArray());
            } else
            {
                msg.Add(JsonFormatter.Default.Format(keyPressed));
            }

            Send(msg);

        }
    }
```

# Web page

**Listing 7.11:** Socket.js

```
var Socket = function (host, port) {

    var that = this;

    this.host = host;
    this.port = port;
    this.noParsing = false;

    this.reqCallbacks = {};
    this.subCallback = undefined;

    var openListeners = [];
    var closeListeners = [];


    var ws = new WebSocket("ws://" + host + ":" + port);
    ws.binaryType = "arraybuffer";

    this.onopen = function (func) {
        openListeners.push(func);
    };

    this.onClose = function (func) {
        closeListeners.push(func);
    };

    ws.onopen = function () {
        console.log('Connection open');
        for (var i = 0; i < openListeners.length; i++) {
            openListeners[i]();
        }
    };

    ws.onclose = function () {
        console.log('Connection closed');
        for (var i = 0; i < closeListeners.length; i++) {
            closeListeners[i]();
        }
    };


    ws.onmessage = function (evt) {

        var msg = new VicoMsg();
        msg.decode(evt.data);

        var type = msg.popByte();
        if (type === 0x000) {
            that.handle0x000(msg);
        } else if (type === 0x001) {
            that.handle0x001(msg);
        } else if (type === 0x003) {
            that.handle0x003(msg);
        } else if (type === 0x004) {
            that.handle0x004(msg);
        } else if (type === 0x009) {
            that.handle0x009(msg)
        }
        msg.destroy();


    };

    this.isConnecting = function () {
        return ws.readyState === 0;
    };

    this.isOpen = function () {
        return ws.readyState === 1;
    };

    this.isClosing = function () {
        return ws.readyState === 2;
    };

    this.isClosed = function () {
        return ws.readyState === 3;
    };

    this.close = function () {
        ws.close();
    };
```

```
    this.send = function (data) {
        ws.send(data);
    };

    this.start, this.bytes = 0, this.parseTime = 0;
    this.pStart;

};

Socket.prototype.generateGUID = function () {
    function s4() {
        return Math.floor((1 + Math.random()) * 0x10000)
                .toString(16)
                .substring(1);
    }
    return s4() + s4() + '-' + s4() + '-' + s4() + '-' +
            s4() + '-' + s4() + s4() + s4();
};


Socket.prototype.request = function (data, binary, callback) {
    var guid = this.generateGUID();
    this.reqCallbacks[guid] = callback;

    var format = binary === true ? 0x000 : 0x001;

    var msg = new VicoMsg();
    msg.addByte(0x000);
    msg.addString(guid);
    msg.addByte(format);

    if (format === 0x000) {
        msg.add(Proto.Request.encode(Proto.Request.create(data)).finish());
    } else if (format === 0x001) {
        msg.addString(JSON.stringify(data));
    } else {
        msg.destroy();
        return;
    }

    this.send(msg.encode());
};

Socket.prototype.subscribe = function (binary, callback) {

    this.subCallback = callback;

    var msg = new VicoMsg();
    msg.addByte(0x001);
    msg.addString('subscribe');
    msg.addByte(binary === true ? 0x000 : 0x001);

    this.send(msg.encode());
};

Socket.prototype.unsubscribe = function () {
    var msg = new VicoMsg();
    msg.addByte(0x001);
    msg.addString('unsubscribe');

    this.send(msg.encode());
};

Socket.prototype.jsonrpc_request = function (method, params, callback) {

    var guid = this.generateGUID();

    var payload = {
        jsonrpc: "2.0",
        method: method,
        params: params,
        id: guid
    };

    this.reqCallbacks[guid] = {callback: callback, payload: payload};
    var msg = new VicoMsg();
    msg.addByte(0x004);
    msg.addString(JSON.stringify(payload));
    this.send(msg.encode());

};

Socket.prototype.jsonrpc_notify = function (method, params) {
    var payload = {
        jsonrpc: "2.0",
        method: method,
        params: params
    };
    var msg = new VicoMsg();
    msg.addByte(0x004);
```

```
    msg.addString(JSON.stringify(payload));
    this.send(msg);
};

Socket.prototype.keyPressed = function (vicoKey, format) {

    var i = Proto.Key[vicoKey];
    if (i !== undefined) {
        var data = {keys: [i]};
        var msg = new VicoMsg();
        msg.addByte(0x002);
        msg.addByte(format);
        if (format === 0x000) {
            msg.add(Proto.KeyPressed.encode(Proto.KeyPressed.create(data)).finish());
        } else if (format === 0x001) {
            msg.addString(JSON.stringify(data));
        } else {
            msg.destroy();
            return;
        }

        this.send(msg.encode());
    }


};

Socket.prototype.plotRequest = function (data, binary, callback) {

    var guid = this.generateGUID();
    this.reqCallbacks[guid] = callback;

    var format = binary === true ? 0x000 : 0x001;

    var msg = new VicoMsg();
    msg.addByte(0x003);
    msg.addString(guid);
    msg.addByte(format);

    if (format === 0x000) {
        msg.add(Proto.PlotRequest.encode(Proto.PlotRequest.create(data)).finish());
    } else if (format === 0x001) {
        msg.addString(JSON.stringify(data));
    } else {
        msg.destroy();
        return;
    }
    this.send(msg.encode());

};

Socket.prototype.spawnObject = function (cam, binary) {

    var format = binary === true ? 0x000 : 0x001;

    var msg = new VicoMsg();
    msg.addByte(0x005);
    msg.addByte(format);

    var pos = cam.getWorldPosition();
    var vel = cam.getWorldDirection();
    var data = {
        position: {x: pos.x, y: -pos.z, z: pos.y},
        velocity: {x: vel.x * 20, y: -vel.z * 20, z: vel.y * 20},
        quaternion: {x: 0, y: 0, z: 0, w: 1},
        sphere: {radius: 0.1}
    };

    if (format === 0x000) {
        msg.add(Proto.Spawn.encode(Proto.Spawn.create(data)).finish());
    } else if (format === 0x001) {
        msg.addString(JSON.stringify(data));
    } else {
        msg.destroy();
        return;
    }
    this.send(msg.encode());

};


Socket.prototype.handle0x000 = function (msg) {
    var guid = msg.popString();

    var response = undefined;
    var callback = this.reqCallbacks[guid];
    if (callback !== undefined) {
        if (this.noParsing === false) {
            var format = msg.popByte();
            if (format === 0x000) {
```

```
                response = Proto.Response.decode(msg.popData());
            } else if (format === 0x001) {
                response = JSON.parse(msg.popString());
            }
        }
    }

    msg.destroy();


    callback(response);
    delete this.reqCallbacks[guid];

};

Socket.prototype.handle0x001 = function (msg) {

    if (this.subCallback !== undefined) {

        var data = undefined;
        var format = msg.popByte();
        if (format === 0x000) {
            data = Proto.Publication.decode(msg.popData());
        } else if (format === 0x001) {
            data = JSON.parse(msg.popString());
        }

        if (data !== undefined) {
            this.subCallback(data);
        }
    }
};

Socket.prototype.handle0x003 = function (msg) {
    var guid = msg.popString();
    var response = undefined;
    var callback = this.reqCallbacks[guid];
    if (callback !== undefined) {
        var format = msg.popByte();
        if (format === 0x000) {
            response = Proto.PlotResponse.decode(msg.popData());
        } else if (format === 0x001) {
            response = JSON.parse(msg.popString());
        }
    }

    msg.destroy();

    if (response !== undefined) {
        callback(response);
        delete this.reqCallbacks[guid];
    }
};

Socket.prototype.handle0x004 = function (msg) {
    var msg = JSON.parse(msg.popString());
    if (msg.id !== undefined) {
        var id = msg.id;

        if (this.reqCallbacks[id] !== undefined) {

            if (msg.error !== undefined) {
                var trace = {error: msg, payload: this.reqCallbacks[id].payload};
                console.error(trace);
            } else {
                this.reqCallbacks[id].callback(msg.result === null ? 'void' : JSON.parse(msg.result));
            }
        }

    } else {
        console.log(msg);
    }
};
```

# Java

**Listing 7.12:** VicoFrame.java

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.nio.charset.Charset;
import java.util.Arrays;

/**
 *
 * @author laht
 */
public class VicoFrame {

    private final static Charset CHARSET = Charset.forName("UTF-8");

    private byte[] data;

    public VicoFrame(byte[] data) {
        this.data = data;
    }

    public VicoFrame(String data) {
        if (data != null) {
            this.data = data.getBytes(CHARSET);
        }
    }

    public byte[] getData() {
        return data;
    }

    public byte getByte() {
        return data[0];
    }

    public String getString() {
        if (hasData()) {
            return new String(getData(), CHARSET);
        }
        return null;
    }

    public boolean hasData() {
        return data != null;
    }

    public int size() {
        if (hasData()) {
            return data.length;
        }
        return 0;
    }

    public void destroy() {
        if (hasData()) {
            data = null;
        }
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 61 * hash + Arrays.hashCode(this.data);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final VicoFrame other = (VicoFrame) obj;
        return Arrays.equals(this.data, other.data);
    }

}
```

**Listing 7.13:** VicoMsg.java

```java
import java.io.IOException;
import java.io.InputStream;
import java.nio.ByteBuffer;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;

/**
 *
 * @author laht
 */
public class VicoMsg implements Iterable<VicoFrame> {

    private Deque<VicoFrame> frames;

    public VicoMsg() {
        this.frames = new ArrayDeque<>();
    }

    public static VicoMsg decode(byte[] data) {
        return VicoMsg.decode(ByteBuffer.wrap(data));
    }

    public static VicoMsg decode(ByteBuffer buf) {

        VicoMsg msg = new VicoMsg();
        int contentSize = buf.getInt();
        for (int i = 0; i < contentSize;) {

            int frameSize = buf.getInt();
            byte[] data = new byte[frameSize];
            buf.get(data, 0, data.length);

            VicoFrame frame = new VicoFrame(data);
            msg.add(frame);

            i += frameSize + 4;
        }
        return msg;
    }

    public static VicoMsg decode(InputStream stream) {
        try {
            byte[] sizeBuf = new byte[4];
            stream.read(sizeBuf);
            VicoMsg msg = new VicoMsg();
            int contentSize = ByteBuffer.wrap(sizeBuf).getInt();
            for (int i = 0; i < contentSize;) {
                stream.read(sizeBuf);
                int frameSize = ByteBuffer.wrap(sizeBuf).getInt();
                byte[] data = new byte[frameSize];
                stream.read(data, 0, data.length);
                VicoFrame frame = new VicoFrame(data);
                msg.add(frame);
                i += frameSize + 4;
            }
            return msg;
        } catch (IOException ex) {
            return null;
        }
    }

    public int messageSize() {
        return contentSize() + 4;
    }

    public int contentSize() {
        int size = 0;
        for (VicoFrame f : frames) {
            size += f.size() + 4;
        }
        return size;
    }

    public byte[] encode() {

        int contentSize = contentSize();
        ByteBuffer data = ByteBuffer.allocate(contentSize + 4);
        data.putInt(contentSize);
        for (VicoFrame frame : frames) {
            int frameSize = frame.size();
            data.putInt(frameSize);
            data.put(frame.getData());
        }
        data.flip();

       return data.array();
```

```
    }

    public void destroy() {
        if (frames == null) {
            return;
        }
        for (VicoFrame f : frames) {
            f.destroy();
        }
        frames.clear();
        frames = null;
    }

    public VicoMsg add(String stringValue) {
        add(new VicoFrame(stringValue));
        return this;
    }

    public VicoMsg addByte(byte data) {
        add(new byte[]{data});
        return this;
    }

    public VicoMsg add(byte[] data) {
        add(new VicoFrame(data));
        return this;
    }

     public VicoMsg add(byte[] data, boolean zip) {
        add(new VicoFrame(data));

        if (zip) {

        }
        return this;
    }

    public VicoMsg addInt(int i) {
        ByteBuffer buf = ByteBuffer.allocate(4);
        buf.putInt(i);
        buf.flip();
        return add(buf.array());
    }

    public VicoMsg add(VicoFrame e) {
        if (frames == null) {
            frames = new ArrayDeque<>();
        }
        frames.add(e);
        return this;
    }

    public VicoMsg addFirst(VicoFrame e) {
        if (frames == null) {
            frames = new ArrayDeque<>();
        }
        frames.addFirst(e);
        return this;
    }

    @Override
    public Iterator<VicoFrame> iterator() {
        return frames.iterator();
    }

    public VicoFrame pop() {
        return frames.poll();
    }

    public String popString() {
        return pop().getString();
    }

    public byte[] popData() {
        return pop().getData();
    }

    public int popInt() {
        return ByteBuffer.wrap(popData()).getInt();
    }

    public byte popByte() {
        return pop().getByte();
    }

    public int size() {
        return frames.size();
    }
```

```
}
```

**Listing 7.14:** RemoteManager.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetAddress;
import java.net.URL;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.stream.Collectors;
import no.ntnu.mechlab.rpc.RpcService;
import no.ntnu.vico.core.sim.VicoSimulation;
import no.ntnu.vico.proto.connect.ConnectProto;
import no.ntnu.vico.remote.handlers.BandwithTestHandler;
import no.ntnu.vico.remote.handlers.KeyHandler;
import no.ntnu.vico.remote.handlers.PlotHandler;
import no.ntnu.vico.remote.handlers.RequestHandler;
import no.ntnu.vico.remote.handlers.ServiceHandler;
import no.ntnu.vico.remote.handlers.SpawnHandler;
import no.ntnu.vico.remote.handlers.SubscriptionHandler;
import no.ntnu.vico.remote.handlers.service.VicoService;
import no.ntnu.vico.remote.net.Heartbeat;
import no.ntnu.vico.remote.net.WritableConnection;
import no.ntnu.vico.remote.net.http.SimpleHTTPServer;
import no.ntnu.vico.remote.net.http.HttpRequestServer;
import no.ntnu.vico.remote.net.tcp.TCPServer;
import no.ntnu.vico.remote.net.udp.UDPServer;
import no.ntnu.vico.remote.net.ws.WSServer;
import no.ntnu.vico.remote.net.zmq.ZmqServer;
import org.slf4j.LoggerFactory;
import no.ntnu.vico.remote.net.VicoServer;

/**
 *
 * @author laht
 */
public final class RemoteManager {

    private final static org.slf4j.Logger LOG = LoggerFactory.getLogger(RemoteManager.class);

    private final Heartbeat heartbeat;
    private final ServiceHandler serviceHandler;
    private final List<VicoServer> servers;
    private final Map<Byte, MessageHandler> handlers;

    private boolean started;

    public RemoteManager(VicoSimulation sim) throws IOException {
        this.handlers = new HashMap<>();
        this.servers = new ArrayList<>(Arrays.asList(new VicoServer[]{
            new TCPServer(this),
            new ZmqServer(this),
            new WSServer(this),
            new UDPServer(this),
            new HttpRequestServer()

        }));

        this.heartbeat = new Heartbeat(createConnectMsg(sim));

        registerHandler(new RequestHandler(sim));
        registerHandler(new SubscriptionHandler(sim));
        registerHandler(new KeyHandler(sim));
        registerHandler(new PlotHandler(sim));
        registerHandler(serviceHandler = new ServiceHandler());
        registerHandler(new SpawnHandler(sim));
        registerHandler(new BandwithTestHandler(sim));

        registerService(new VicoService(sim));

    }

    public void addServer(VicoServer server) {
        this.servers.add(server);
        if (started) {
            server.start();
        }
    }
```

```
public void registerHandler(MessageHandler handler) {
    byte key = handler.getKey();

    if (handlers.containsKey(key)) {
        throw new IllegalStateException("Key currently in use by: '" + handlers.get(key).getClass().getSimpleName()
                + "'");
    }
    handlers.put(key, handler);
}

public void registerService(RpcService service) {
    serviceHandler.registerService(service);
}

public void handle(WritableConnection con, VicoMsg msg) {

    byte key = msg.popByte();
    MessageHandler handler = handlers.get(key);
    if (handler != null) {
        handler.handle(con, msg);
    } else {
        msg.destroy();
        LOG.warn("No handler registered for key: '{}'", (int) key);
    }

}

public void start() {

    servers.forEach(s -> s.start());
    heartbeat.start();
    started = true;

    LOG.info("RemoteManager started!");

}

public void stop() {

    servers.forEach(s -> s.stop());
    heartbeat.stop();

    LOG.info("RemoteManager stopped!");

}

private ConnectProto.ConnectMsg createConnectMsg(VicoSimulation sim) {

    String hostAddress;
    try {
        hostAddress = InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException ex) {
        hostAddress = "127.0.0.1";
        Logger.getLogger(RemoteManager.class.getName()).log(Level.SEVERE, null, ex);
    }

    ConnectProto.ConnectMsg.Builder builder = ConnectProto.ConnectMsg.newBuilder();
    builder.setUuid(sim.getUuid().toString())
            .setSimulationName(sim.getName())
            .setDescription("No desc..")
            .setHostAddress(hostAddress)
            .putAllPorts(servers.stream().collect(Collectors.toMap(VicoServer::getName, VicoServer::getPort)));

    return builder.build();
}
```