Utrecht University

Crowd Simulation as a Service: A scalable, real-time architecture.

Moving the crowd to the cloud.

Chrit Hameleers
ICA-3978125

supervised by:
Dr. R.J. Geraerts
Dr. ir. J.M.E.M. van der Werf

Master's Thesis in the Game and Media Technology program

in the
Faculty of Science
Department of Information and Computing Sciences

February 25, 2019

Abstract

In this thesis on real-time crowd simulation as a service, we explore how crowd simulation applications can benefit from cloud computing. The aim of this thesis is to provide a candidate architecture for real-time cloud-based simulation software that is scalable and highly performant. We also aim to document our design process to show how and why decisions were made.

First, we introduce the subject by motivating the need for cloud computing in simulation software. We then provide an overview of related work about both crowd simulation and computer and cloud architecture. Next, we describe the problems we are trying to solve and we list a collection of requirements that will help guide our design process. We then combine our knowledge into a candidate architecture for real-time Modeling and Simulation as a Service (MSaaS). After presenting our candidate architecture we perform research into the performance sensitive parts of MSaaS and discuss how we can apply our results.

Acknowledgements

I would like to thank my supervisors Dr. Roland Geraerts and Dr. ir. Jan Martijn van der Werf for all their guidance, support and advice. Furthermore, I would like to thank my loving and supporting girlfriend, family and friends which all supported me through the good times and the bad. Finally, I want to thank my fellow board members of USDV U Dance, who always remained patient and supportive while I finished this thesis.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**

2 Related work **3**
- 2.1 Crowd Simulation . . . 3
- 2.2 Cloud Architecture . . . 5
  - 2.2.1 MSaaS . . . 5
    - 2.2.1.1 Modeling . . . 6
    - 2.2.1.2 Simulation . . . 6
- 2.3 Software Architecture . . . 6
  - 2.3.1 Architectural patterns . . . 7
    - 2.3.1.1 Model View Controller . . . 7
    - 2.3.1.2 Broker Pattern . . . 7
    - 2.3.1.3 Enterprise Service Bus pattern . . . 9
    - 2.3.1.4 Service-Oriented Architecture Pattern . . . 11
- 2.4 Architecture Evaluation through Architecture Mining . . . 12

3 Problem description **13**
- 3.1 Modeling Pipeline . . . 13
- 3.2 Simulation . . . 14
- 3.3 Architecture design . . . 15

4 Requirements **16**
- 4.1 Quality attributes . . . 16
  - 4.1.1 Performance . . . 16
  - 4.1.2 Security . . . 17
  - 4.1.3 Maintainability, Modifiability and Interoperability . . . 18
- 4.2 Functional requirements . . . 18
  - 4.2.1 Front end . . . 19
  - 4.2.2 Back end . . . 19

5 Candidate architecture **20**
- 5.1 Client with MVC . . . 20
- 5.2 Client-Server model . . . 21

5.3 Message bus 22
5.4 Defining the service 26
5.4.1 Modeling service 27
5.4.2 Simulation service 27
5.4.3 Data management and authorization 28
5.5 The complete picture 28

6 Architecture analysis **30**
6.1 Experiment introduction 30
6.1.1 software used 31
6.1.2 Hardware used 31
6.1.3 Variables 31
6.1.3.1 Processing time 31
6.1.3.2 Data size 32
6.1.3.3 Independent variables 32
6.1.4 Hypotheses 32
6.2 Experiment setup 33
6.2.1 Experiment details 33
6.2.2 The Scenarios 34
6.2.2.1 Scenario 1 34
6.2.2.2 Scenario 2 34
6.2.2.3 Scenario 3 34

7 Results **37**
7.1 Data size **D** results 37
7.1.1 Network throughput 37
7.1.2 Network Protocol 38
7.2 Processing time **P** results 39
7.2.1 Scenario 1 39
7.2.1.1 Outliers 40
7.2.2 Scenario 2 40
7.2.3 Scenario 3 41

8 Conclusion **44**
8.1 Research contribution 44
8.2 Discussion of experiments 45
8.3 Discussion of research method 45
8.4 Future work 46
8.4.1 Implementation 46
8.4.2 Security 46
8.4.3 Non-agent based systems 46

References **47**

Chapter 1

Introduction

Crowd simulation software has many applications across a large number of different domains. Urban planners can use simulations to make our environments safer and more efficient. Architects can use these models to test evacuation scenarios and local governments can plan large events without worrying about congestion and the crowd disasters associated [1]. Furthermore, a large variety of games from entertainment to educational content can make good use of realistic large scale real-time simulation software as well. To better facilitate these demands, we aim to bring crowd simulation technology to the users in a way that is scalable, highly performant and affordable. The modern solution is to take advantage of the steady increase of performance and affordability in cloud hosting. Turning simulation into an online service in the form of a SaaS model (Software as a Service) or even a PaaS model (Platform as a Service) seems like a logical move. This practise has been called MSaaS [2] (Modeling and Simulation as a Service) or SimSaaS [3] (Simulation Software as a Service).
Since we are looking at both the creation of the environment (model) and the execution of the simulation in a service-oriented architecture we will go with calling it MSaaS.

We want to achieve an architecture for the modeling and simulation of crowds which can be scaled to serve many users simultaneously. State-of-the-art crowd simulation software is still expensive and requires powerful hardware to run. Allowing the software to run as a cloud-based application makes this technology more accessible.
In this thesis we look at the issues encountered when scaling up crowd simulation applications and the best ways of dealing with them.
This thesis intends to show an overview of available literature on MSaaS and to document the architectural decisions involved in designing such a system. An elaborate view into this process is valuable to future architects of cloud based crowd simulation systems.

First, we delve into the current state of cloud architecture and crowd simulation in the related work section in Chapter 2. Chapter 3 outlines the problems this thesis is trying to solve in more detail. We show the requirements related to MSaaS architecture in Chapter 4. Next we propose a candidate architecture for an MSaaS system in Chapter 5. We perform practical analysis of this architecture in chapter 6 where we discuss the reasoning behind the architecture proposed in Chapter 5. Finally, Chapter 7 shows the results of the analysis and Chapter 8 concludes the thesis.

Chapter 2

Related work

This chapter provides an overview of topics relevant to MSaaS. The first section provides an understanding of the crowd simulation research field. The second part of the chapter is dedicated to architectural patterns used in the design of cloud-based applications. The final part discusses software architecture evaluation methods.

2.1 Crowd Simulation

Crowd simulation is the science of simulating the movements of a large number of virtual people (agents) simultaneously. The agents should respond to the environment and each other as accurately and realistically as possible.
Crowd simulation combines knowledge of various disciplines to achieve this goal: computer science, artificial intelligence, mathematics and behavioural studies being the most prominent. Challenges from these disciplines include: dealing with complex environments, creating intelligent agent behaviour, creating realistic agent behaviour and optimizing for large crowds.

Path planning occurs on multiple levels within a simulation. High-level planning determines the global route agents follow through the environment, while low-level planning handles more local challenges like avoiding collisions with the environment and other agents. Path planning behaviours for agents originate from algorithms like A* and boids flocking [4] but have become a lot more advanced. For example, a method exists using forces to create certain social behaviours [5]. Another method makes agent control more global by using particle systems that guide agents [6]. Some even take inspiration from fluid dynamics by observing similarities between the flow of liquids and the flow of crowds [7]. These methods, whether agent-based, particle-based or flow-based, have their own strengths and limitations that need to be considered. Agent-based systems can create a wide variety of specialized behaviour at the cost of processing power per agent. Particle- and flow-based systems can handle a large number of agents better but utilize assumptions about common agent goals and behaviours. These advanced methods exploit the steady increase of hardware capabilities, often using large amounts of processing power and memory. Sufficient processing power is needed to update large quantities of agents every simulation step. For simulations of large scale, memory becomes a bottleneck as well. This bottleneck depends on the complexity of the environment and how dynamic the agent behaviour is.

Crowd simulation software has many real-life applications: Designing efficient escape routes for a large building, preventing crowd disasters and improving people flow at large events, simulating pedestrian behaviour for urban planning and even creating convincing simulated crowds in movies and video games [8].

Because of this wide range of useful applications and the steep investment in expensive commercial software and state-of-the-art hardware, there exists an opportunity to capitalize on the advances and popularity of cloud computing. Section 2.2 introduces this topic in more detail. By distributing the workload of the software across multiple powerful machines, crowd simulation applications can be made accessible to smaller organizations that could benefit from it. It also allows the agent based approach to crowd simulation to be scaled up to much larger simulations without running into hardware limits. Applications that perform crowd simulation and modeling in the cloud are called MSaaS (modeling and Simulation as a Service), a derivation of the more popular Software as a Service, or SaaS.

Previous research into MSaaS shows promising results in the scalability of cloud-based crowd simulation. As shown in Malinowski, Artur, et al.(2017) [9], MSaaS applications can drastically improve performance by exploiting powerful hardware and clever parallelization algorithms. Their implementation is not a cloud application, but it does utilize parallel computing and powerful computer clusters to simulate an area of 6 square kilometers for 100.000 agents in under 3 hours. Research by Wagoum, Armel Ulrich Kemloh, et al. [10] shows that parallel computing can facilitate real-time crowd simulation applications in practical scenarios. Our goal is to find ways to translate these methods to a cloud architecture to facilitate real-time Simulation as a Service. MSaaS is further discussed in Section 2.2.1.

2.2 Cloud Architecture

Cloud architecture is a broad collection of software and hardware combinations by using distributed computing [11]. Three main branches are commonly identified. SaaS focuses on making a traditional application available through cloud computing, keeping all control with the developer and allowing the users to use the application and their data from any device with an internet connection. An example of this is Google maps [12].

PaaS describes a service that makes it easier for their users to create their own specialized applications. Examples of these are application hosting services such as Microsoft Azure, Amazon Web Services and Google App Engine [12].

The third variant is IaaS, which automates the distribution of hardware resources. Examples are again Microsoft Azure and Amazon Web Services with Google providing Google Compute Engine [12].

For SaaS applications, cloud computing is a set of enabling technologies which create a platform for SaaS developers to build their Service Oriented Architecture (Section 2.3.1.4) solutions [12]. Cloud computing can be used to make simulation software more available and portable, but bringing simulation and modeling to the cloud also introduces a new set of issues. All data needs to travel over a networked connection which forces us to think about the effect on performance caused by the additional overhead and the limited bandwidth. Additionally, the data is often privacy sensitive so security is a big concern as well for environment models and simulation results. Finally, reliability is important as we have to guarantee access to the service for all clients with minimal downtime and optimal performance during peak hours. A complete specification of quality attributes and functional requirements can be found in Chapter 4.

2.2.1 MSaaS

MSaaS stands for Modeling and Simulation as a Service [13]. It is essentially the SaaS model applied to the modeling and simulation of environments. For this thesis we are specifically looking into real-time modeling and simulation on a large scale.
MSaaS has been gaining more attention from researchers and institutions for it promises to be an important tool in many areas of application. These areas were discussed in a NATO publication [2] about the advantages, disadvantages and requirements for future MSaaS implementations as well, together with an analysis of some existing ones.

2.2.1.1 Modeling

Modeling in this context refers to the designing and building of a 2D or 3D environment and scenario for use with a crowd simulation engine. This could be as complex as a Full 3D model that needs to be converted into a navigation mesh [14] or as rudimentary as a drag and drop editor with basic shapes. Designing simulation scenarios is a form of modeling as well. This entails setting the number of agents for the scenario, where they enter and leave the environment and the agent behaviour or other environmental triggers and events. Modeling in the cloud allows users to model from any connected device without needing specialized hardware. It also allows users to collaborate on designing an environment model from multiple devices. Information contained within these models often include classified information such as floor plans, crowd density data or even military secrets. Because of this, it is important the models are stored and shared in a secure way.

2.2.1.2 Simulation

Simulation refers to the using of a model of an environment and analyzing the behaviour of agents interacting with the environment and each other.
Simulation is the most performance intensive part of the MSaaS model and brings the greatest challenges when it comes to efficient use of network bandwidth, processing power and memory.
Because MSaaS makes crowd simulation much more portable, scalable and available, it is important it works reliably under most network conditions. Latency, unreliable connections and bandwidth restraints are all factors to take into account when designing a robust architecture for MSaaS.

2.3 Software Architecture

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." [15].

As stated in the above definition, we need a set of structures to reason about the system. This section provides such structures and explains their properties. We want to find an abstracted view of the MSaaS software, identifying the most important components and analyzing and mapping the interactions between them. To do this, we analyze a set of architectural patterns that help solve architectural problems related to the MSaaS domain.

2.3.1 Architectural patterns

Architectural patterns in software engineering are template architectures that can be applied to solve a specific commonly occurring architectural problem. They are similar to software patterns but on a bigger scale. Architectural patterns are used to improve certain aspects of software. The patterns in this section are mainly communication patterns.

2.3.1.1 Model View Controller

The model-view-controller pattern is used to separate application functionality into three components: *the model*, which stores the application data, *the view*, which displays relevant data and handles user interaction, and *the controller*, which ties the model and the view together by communicating state changes between them. The MVC pattern introduces complexity to the architecture in exchange for modularity and looser coupling of application logic.

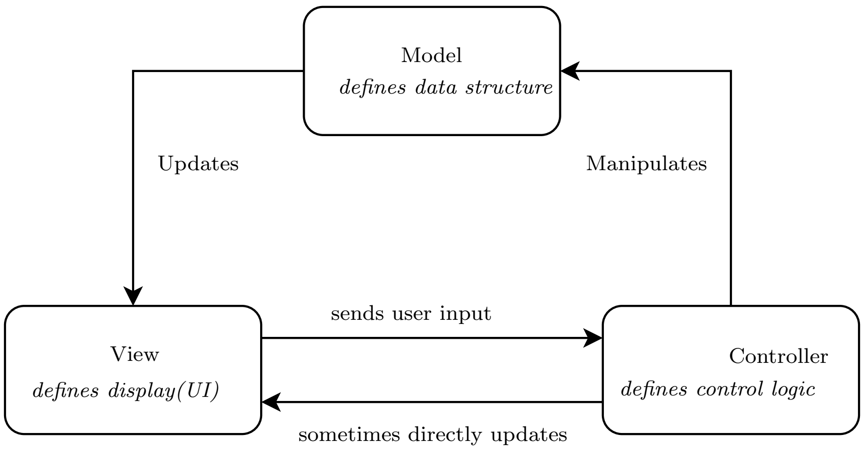

FIGURE 2.1: The model view controller pattern[15][16]

2.3.1.2 Broker Pattern

A broker pattern is used to connect multiple clients to multiple servers while obfuscating the actual details of the architecture to the front-end user [15]. It effectively separates the *client* and *server* by introducing an intermediary 'broker' component. The broker

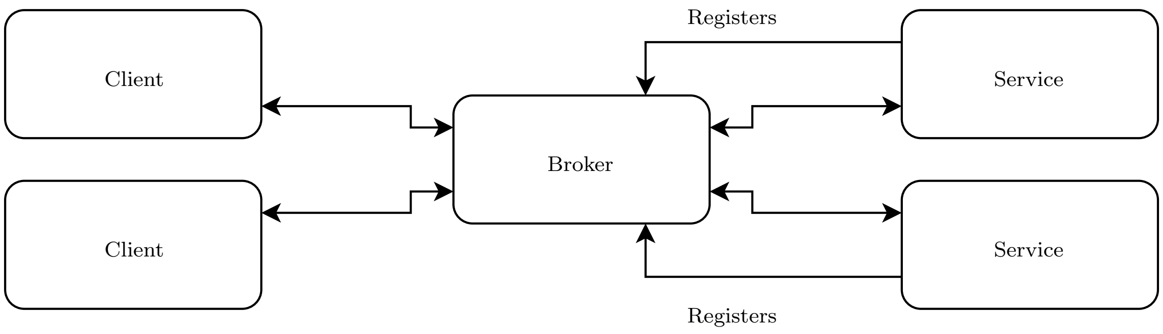

Figure 2.2: The broker pattern. [18][19]

pattern brings some advantages and disadvantages we need to consider. We list these below, starting with the advantages.

The clients do not know the identity and location of the server. This makes it easy to alter or update the back-end components without requiring front-end updates. It allows us to develop the individual components of the architecture side by side more easily because of this separation. Finally, it allows us to deal with the inevitable complexity that comes with a distributed, parallel execution of the simulation over multiple (virtual) machines [17].

Next, we look into the disadvantages of the broker pattern. The front-end clients usually only communicate with the single broker. The introduction of multiple brokers, for example when scaling the architecture, introduces extra complexity. This makes the broker a bad candidate when scaling up the architecture to support many clients and services. We look into the Enterprise Service Bus pattern in Section 2.3.1.3, which trades some control for scalability to solve some of these issues.

The extra network hop the broker introduces adds more overhead and latency. Obfuscating information behind a broker decreases opportunities for optimization as well, because of the loss of contextual knowledge about the data being sent [18]. Finally, a scenario with a single broker is a performance bottleneck for communication as well as a security and reliability risk: If the broker goes down, the entire system will cease to function. In the context of real-time crowd simulation, we have to find a way to maximize performance and minimize security risk.

The broker pattern can be implemented as an intermediary where all the communication between all clients and servers is processed and forwarded by the broker. In the most basic scenario it acts as a hub connecting multiple clients and services together. In this scenario, which is shown in Figure 2.3, the client never directly communicates with the server: All the communication happens through the intermediary broker. The benefit

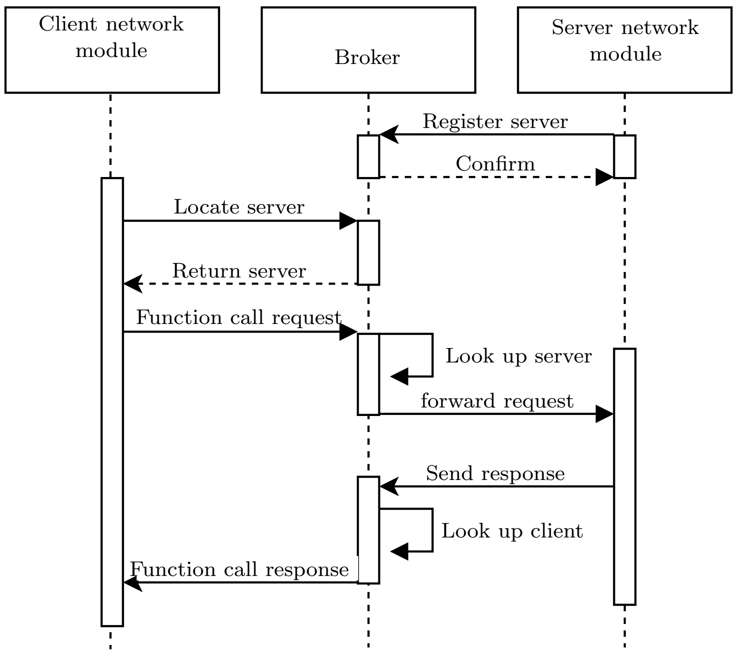

FIGURE 2.3: The broker pattern as intermediary [18]

of this is that the broker can apply additional logic to the request sent and the client doesn't need to know what services exist in the back-end.

A broker can also connect a client to a server so they can communicate directly. This is called a broker with server look-up or a registry. In this scenario, the client sends a message to the broker asking for the address of a server that can handle its request. The broker then provides this address from a list of servers that previously registered to the broker. The client can then send its request to the server which can then process the request and respond to the client. This is shown in Figure 2.4.

2.3.1.3 Enterprise Service Bus pattern

The Enterprise Service Bus (ESB) pattern (Figure 2.5) evolved as a solution to increasingly large and hard to maintain collections of software in enterprise architectures. It consists mainly of a messaging service (the bus) which enables communication between services and clients. The data transferred over the bus is standardized to a format shared by all clients and servers. This way every connected component can subscribe to, and publish to the message bus. This can be achieved by designing all components to use the same message format or by retroactively fitting the components with an adapter to translate messages to and from a format the bus understands. Alternatively, the ESB can have built-in translations to support the most common message types, or the message types most used in the domain it is applied in.

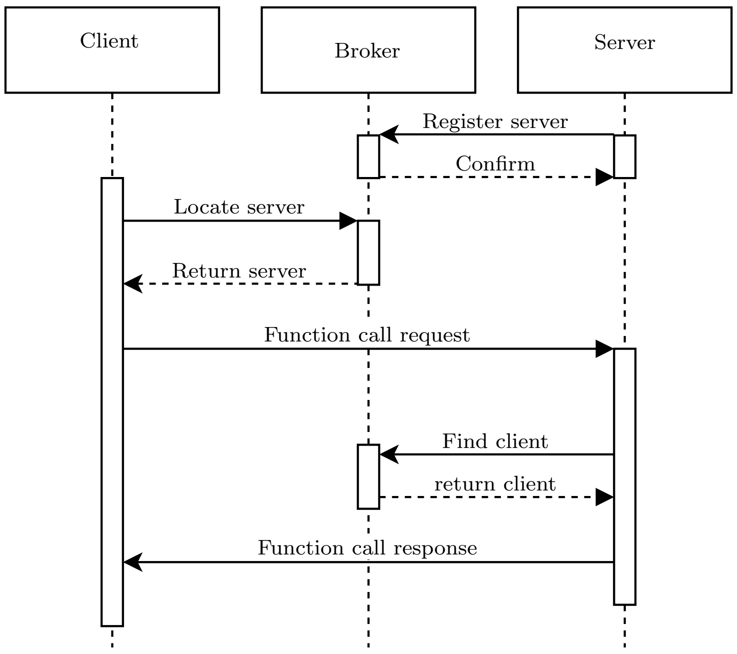

FIGURE 2.4: The broker pattern with server look-up [18]

The ESB differs from a broker pattern by not interfering with the messages sent on the bus. In a broker pattern, the broker manages every message received and sent. It obfuscates to a component what other components exist in the network and decides what message goes where. This traditional broker is called the hub-and-spokes pattern and is useful for smaller applications but does not scale well due to the larger amount of overhead added with each component that needs to us the hub.

The ESB however, sends all communication through the bus and all connected components may listen to it. It applies little to no logic to the messages it broadcasts to all connected components besides very simple translation between data formats and publish-subscribe functionality.

The advantages of the ESB pattern are its scalability and its loose coupling. Because the EBS contains no program logic, it is relatively easy to scale it up horizontally. The ESB also allows services to be very loosely coupled as long as they share a common format of communication.

The downsides or liabilities of the ESB are found in security, flexibility and functionality. Security is a point of concern in the ESB because a weakness in one connected app can expose all connected systems to attacks. The ESB can also be inflexible to work with when it is not possible to adapt an application to work with the desired message format. Expanding the functionality of the ESB is tricky because a change in the way messages are handled impacts all connected components. Finally, it cannot support certain functionality a broker might: complex message routing, message aggregation and message splitting, for example.

The ESB pattern supports a Service-Oriented Architecture, which we dicuss in the next section (Section 2.3.1.4).

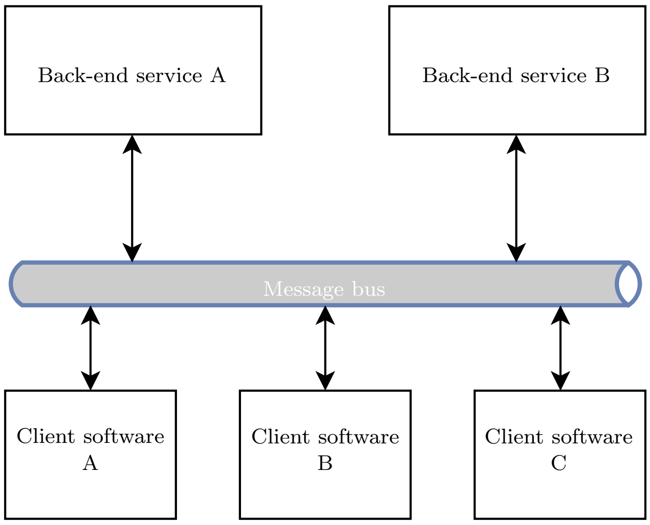

FIGURE 2.5: The ESB pattern. [19]

2.3.1.4 Service-Oriented Architecture Pattern

The Service Oriented Architecture (SOA) pattern is a high-level pattern that facilitates cloud applications by building on top of a broker or ESB pattern [15][12]. Its purpose is to connect service providers and service consumers. It adds a layer of abstraction between the providers and the consumers to make it easy for the consumers to use the services without needing to know the finer details. It combines different services that may be written in different languages and may be located in different physical locations. SOA is a solution for these problems that takes into consideration security, performance and availability. The SOA pattern often includes a registry or ESB as middle-ware between client services and back-end services. Because of this it shares the advantages and problems of these patterns. Other additions can be made to this pattern to expand its functionality, for example a Registry of services that tracks all available services and makes them easily discoverable by clients. Another addition is an orchestration server which uses set work flows to guide the communication between service provider and consumer.

2.4 Architecture Evaluation through Architecture Mining

To motivate the design choices made for the candidate architecture, we need to be able to evaluate and analyze our architecture.

One such evaluation method is the Architecture Tradeoff Analysis Method (ATAM) [20]. ATAM is a way to analyze a proposed or existing architecture on multiple functional and quality attributes. It does this by identifying risks, sensitivity points and tradeoff points. where risks are important decisions that have not been made. Sensitivity points are parameters in the architecture which have a high correlation with a measurable quality attribute and tradeoff points are parameters where multiple sensitivity points are involved and where changing one of them influences the others. It also evaluates the architecture based on quality attribute characterizations. Where they look at a quality attribute, such as performance, and identify external stimuli, architectural decisions, and responses. External stimuli are events that cause the system to respond or change. Architectural decisions are the aspects of an architecture that directly impact the chosen quality attribute. Responses are the measurable outcomes of architectural decisions such as latency and throughput. Finally, ATAM also makes use of scenario's. These scenarios are grouped into three categories: User scenarios, developer scenarios and customer scenarios. These scenarios document the requirements and demands of these three groups of stakeholders. More on requirements and quality attributes can be found in Chapter 4.

Architecture mining is about the evaluation of software architecture. Analyzing and visualizing the architecture as software is being developed is helpful in ensuring the implemented architecture stays as close as possible to the intended architecture. It can also reveal bottlenecks, flaws and strengths of an architecture which can be used to further evolve the design and implementation. Architecture mining combines Architecture compliance checking [21] and visualization. By extracting useful information such as design patterns [22], and using this information to make visualizations and logs about the architecture, we can make statements about the suitability of proposed architectures and analyze if the implemented architecture holds up to the design and if it fulfills our set requirements.

Chapter 3

Problem description

The goal is to design an architecture for a cloud based real-time crowd simulation application. Besides the final architecture design produced, this thesis aims to document and research the important decisions that led to the final design and to show alternatives. In addition prototypes will be built and further testing and research will be performed.

3.1 Modeling Pipeline

First, we look into the design of an architecture to support the modeling pipeline. This pipeline is responsible for the processing of 3D models into navigation meshes that represent the traversable areas [23]. In this pipeline, the user will upload a 3D model on the front end of the system. The front end will then make a connection to the back end, possibly through a broker or bus managing the connection process. The front end can then upload the 3D model to the modeling server for processing and will receive the processed navigation mesh when it has been fully processed. The 3D data of the environment can come from a variety of sources. The most straightforward being an uploaded model file. But it could also be a more complex client side environment editor which generates 3D data to be processed by the modeling pipeline.

The processing of this data is a complex process that can take a while depending on the environment. It has to perform a number of steps which can be hardware intensive. More specifically, they can be memory intensive because of the loading of many polygons for complex geometrical calculations. The 3D environment has to be flattened into multiple navigable polygons. Any spaces which are too small or too steep to be traversed and spaces that are obstructed need to be filtered [23].

We need to distribute workloads efficiently so that none of the components are kept waiting unnecessarily. Because of the complexity of the processing of 3D data, it is difficult to optimize the back end. A solid distributed implementation for the back end is necessary to ensure scalability while keeping the physical machines required low. Further load balancing and concurrent execution can then be used here to improve performance.

Because this process is usually only run a couple times per simulation project, performance is not the most important attribute here. The data involved could be classified or privacy sensitive, taking into account security risks is more important here. Depending on the scenario the requirements of the modeling service may vary. Can we support a fast service which is also secure enough for sensitive data?

We need to find out what the most efficient way is to set up this service. What gives us the best options for meeting the requirements set? These requirements are formulated and discussed in Chapter 4.

Problem 3.1. What architectural decisions satisfy our set requirements when processing environment models, and why?

3.2 Simulation

The simulation component of the MSaaS environment is responsible for instantiating a simulation given an environment and a set of parameters. This simulation then has to stream real-time results back to the client while listening for changes to the model or parameters provided in real-time by the user. The results also have to be recorded in a way to be able to perform posterior analysis.

To make sure multiple simulations for different users can run without issue, the workload will have to be balanced appropriately across multiple simulation servers. Realizing this to operate smoothly for all use cases is a challenge. Performance is key for this service (as shown in Section 4.1.1) as every performance increase will allow for more agents to be simulated in real-time with more complex environments. Many factors can potentially have a negative impact on performance. Scaling up horizontally (by adding more machines) adds more processing power but will increase overhead. Horizontal scaling will be an important tool when running multiple simulations side by side for different users or when running different parts of the same simulation in parallel. The architecture needs to minimize the overhead this creates. Keeping data streams secure and separated between clients is a requirement which can negatively impact performance as well. These are factors to take into account when designing an architecture for the simulation service.

The main use case for the simulation service has an authorized user connecting to the service, requesting a simulation given a set of parameters and then starting a stream of results from the service back to the client. This stream is then viewed in the client which can then react and alter the parameters. The service needs to be able to calculate about 15 simulated minutes worth of scenario in advance at the minimum so the data can be used to prevent crowd disasters.

Problem 3.2. What architectural decisions satisfy our set requirements when running a crowd simulation in real-time and why?

3.3 Architecture design

Problem 3.3. What architecture for an MSaaS application satisfies the most of our set requirements?

The designed architecture will have to be evaluated against a non-cloud single machine version. Techniques from architecture compliance checking and architecture mining can be applied to formulate well-motivated statements about the performance of the proposed architecture.

Problem 3.4. How can we evaluate our proposed architecture in a meaningful way?

Chapter 4

Requirements

In this chapter, we set a number of requirements that we want our MSaaS architecture to fulfill. The most important ones are listed here and are grouped under the two categories below: Functional requirements and Quality attributes (non functional requirements). These requirements and attributes will provide guidelines and criteria to design a suitable architecture. Requirements will sometimes directly or indirectly influence another requirement which makes requirements engineering a careful balancing act. For our architecture we will highlight the following quality attributes: performance, security and maintainability. In the next section we will go into the functional requirements of the system. Note that these are not the only requirements and attributes that apply, but they are the ones that are the most interesting for this architecture.

4.1 Quality attributes

Quality attributes, or non functional requirements, are used to evaluate the system as a whole. They cover functional stability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability [24]. The attributes that apply most to MSaaS are discussed in this section.

4.1.1 Performance

Performance is a big concern when making a latency-sensitive service available through an MSaaS architecture. The architecture for simulation software in the cloud has to support a real-time simulation mode where the simulation can be monitored and altered as it is being calculated. This allows users to interact quickly to environmental changes and to efficiently work with the software. To accomplish this the architecture needs to facilitate many connections which send or receive many instances of large data packets. Real-time simulation can support up to 60.000 agents when using fast, modern hardware before calculation times exceed the real-time criterion of 10 updates per second [25]. The complexity of the environment is also a limiting factor on the number of agents supported. Thousands of positions and velocities need to be sent through the network at least 10 times per second. To meet this benchmark, the architecture needs to support multiple users working with or viewing the same simulations which means the amount of network bandwidth needed goes up for every client connected.

The performance requirements are as follows:

- *P1*: The system must support the simulation of tens of thousands of agents in real-time (with 10 simulation step updates per second).
- *P2*: The system needs to support many users viewing the same or different simulations simultaneously.
- *P3*: The data streamed through the network must use as little bandwidth as possible and cannot exceed the capacity of a state-of-the-art, high-end internet connection.
- *P4*: The user needs to be able to react to environmental changes in the simulation within 15 minutes. Typically, you have 3 minutes to make a prediction for the coming 15 minutes. This gives enough time to intervene if necessary.

4.1.2 Security

Simulations are often run with sensitive data. Simulation environments, parameters and data can contain sensitive information and the architecture must ensure the customer data is kept secure from anyone but the intended users. Users of simulation software include government organizations such as the military [2]. The data from these clients could even be classified and subject to strict additional security demands. Encryption of data streams might need to be supported at critical points of data transferal and a proper balance needs to be found between security and performance where security takes priority over performance. Alternatively, the architecture could support different levels of security, where lower levels of security provide better performance. This way the architecture can adapt to different areas of application and their security needs.

The security requirements are as follows:

- *S1*: The system must guarantee the safety of environment models.
- *S2*: The system must guarantee the safety of simulation results.
- *S3*: The system must guarantee the safety of user information.
- *S4*: The level of security should be adaptable.

4.1.3 Maintainability, Modifiability and Interoperability

Because real-time crowd simulation is a state-of-the-art research field components are prone to change as frameworks and engines are released or updated with new discoveries in the field.
It is thus crucial to think about how to design the architecture to be flexible to change, expandable and maintainable. It also needs to be able to facilitate communication between a number of different components. Creating well defined and properly documented APIs will help to realize this. This practise is called "Open architecture".

The Maintainability, Modifiability and Interoperability requirements are as follows:

- *M1* Different types of front-end views need to be supported.
- *M2* The used simulation engine or framework could be changed or updated regularly and the architecture needs to support this.
- *M3* The architecture needs to support complex simulation engines and frameworks, especially when parallel computing is utilized.
- *M4* Communication protocols used need to support communication with many types of implementations of both front-end and back-end.

Proper interfaces, wrappers, and adapters need to be designed to meet these requirements and to mitigate the impact of these components being changed as development on them continues.

4.2 Functional requirements

Functional requirements are the requirements that dictate the core functionality the architecture must support.

4.2.1 Front end

The front end is a flexible part of the architecture which can take different shapes depending on the situation. For example: different front-end views can include a view to view the simulation in real-time, a view to set up a simulation, and a view to manage a running simulation. The following functionality needs to be supported to facilitate most of these use cases.

- *F1* Submit an existing 3D model to the server for conversion into a navigation mesh of the traversable environment.
- *F2* Creating and editing a simulation environment in an editor.
- *F3* Viewing simulations and simulation results.
- *F4* Viewing a simulation while making changes in real-time.
- *F5* Accessing stored simulation parameters, results and environments for later use.
- *F6* Viewing statistics for further analysis.

4.2.2 Back end

The back end is where all of the processing happens. It is responsible for running hardware-intensive tasks like the conversion of 3D models into navigation meshes and the running of responsive simulations in real time.

The back end needs to include services that support at least the following functionality:

- *F7* Converting existing 3D models into a navigation mesh of the traversable environment.
- *F8* Running simulations and broadcasting simulation results.
- *F9* Processing client input in real time.
- *F10* Storing simulation parameters, results and environments for later use.
- *F11* Gathering and storing statistics for further analysis.
- *F12* Providing insights, e.g. to prevent crowd disasters.

Chapter 5

Candidate architecture

This section shows the design process the candidate architecture went through and how we arrived at the final architecture design. The architecture patterns and domain knowledge introduced in Chapter 2 are used to lay a foundation for our architecture. This foundation is then examined,altered and extended to conform to the requirements set in Chapter 4.

5.1 Client with MVC

The architecture starts with the MSaaS client. We base this client on an MVC (Section 2.3.1.1) pattern. The user interacts with the client through a view. This view could simply be a view of the simulation, but usually also lets the user interact with the simulation or model an environment. The inputs the user performs are processed in the controller and the controller then updates the internal client model. The model then finally processes the changes and calculates the next steps of the simulation which the view can then display. This completes the MVC cycle. The MVC pattern allows us to encapsulate the client into three well defined components and we can identify the components that are going to be involved in turning this simulation client into a SaaS application. The functionality of the model needs to be moved out of the front-end if we want to turn it into a SaaS application. We want to do this because we want to take advantage of a powerful back-end to do the expensive calculations the model needs to do. The controller is then going to have to communicate over a networked connection to reach the model.

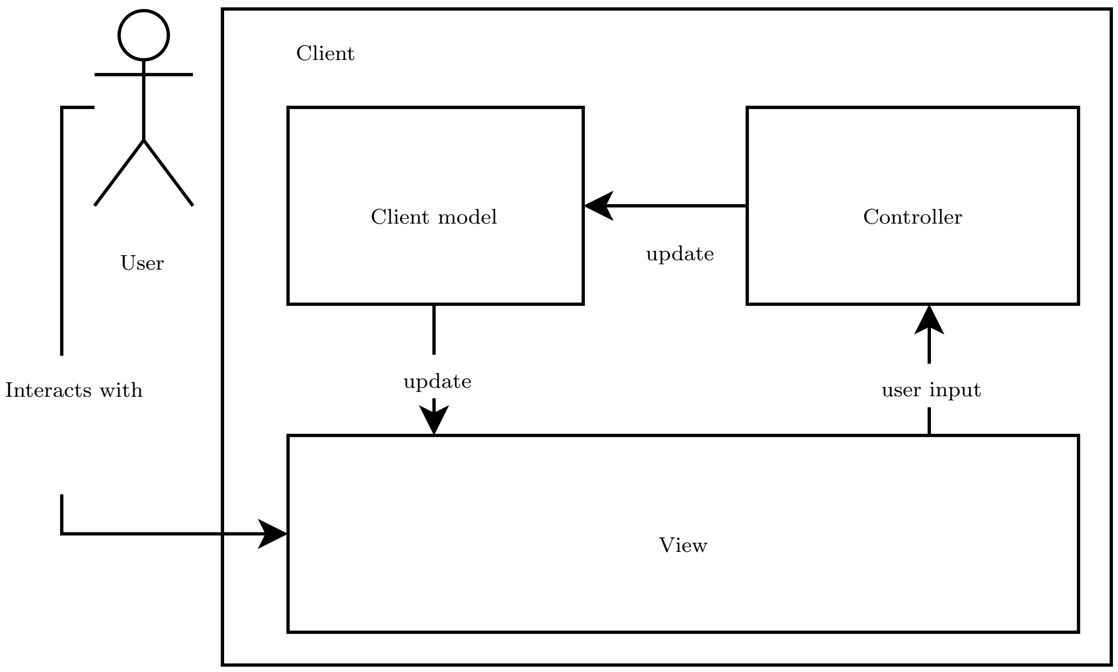

FIGURE 5.1: A basic MVC client

5.2 Client-Server model

To move the calculation-heavy model to the cloud we need to separate it from the client. We do this by taking the MVC model and splitting the controller into two parts that communicate over a networked connection. It may look like we did not move the model to the server because the client model is still in place. This client model is very basic however, and it is updated with already-processed data that the controller receives from the server model. The main reason we still have it here is because the client needs to understand certain data structures to process the updates from the server model. This setup is shown in figure 5.2. The next problem that we need to deal with is the way we are going to facilitate the communication between the front-end controller and the back-end controller when we scale up the architecture horizontally.

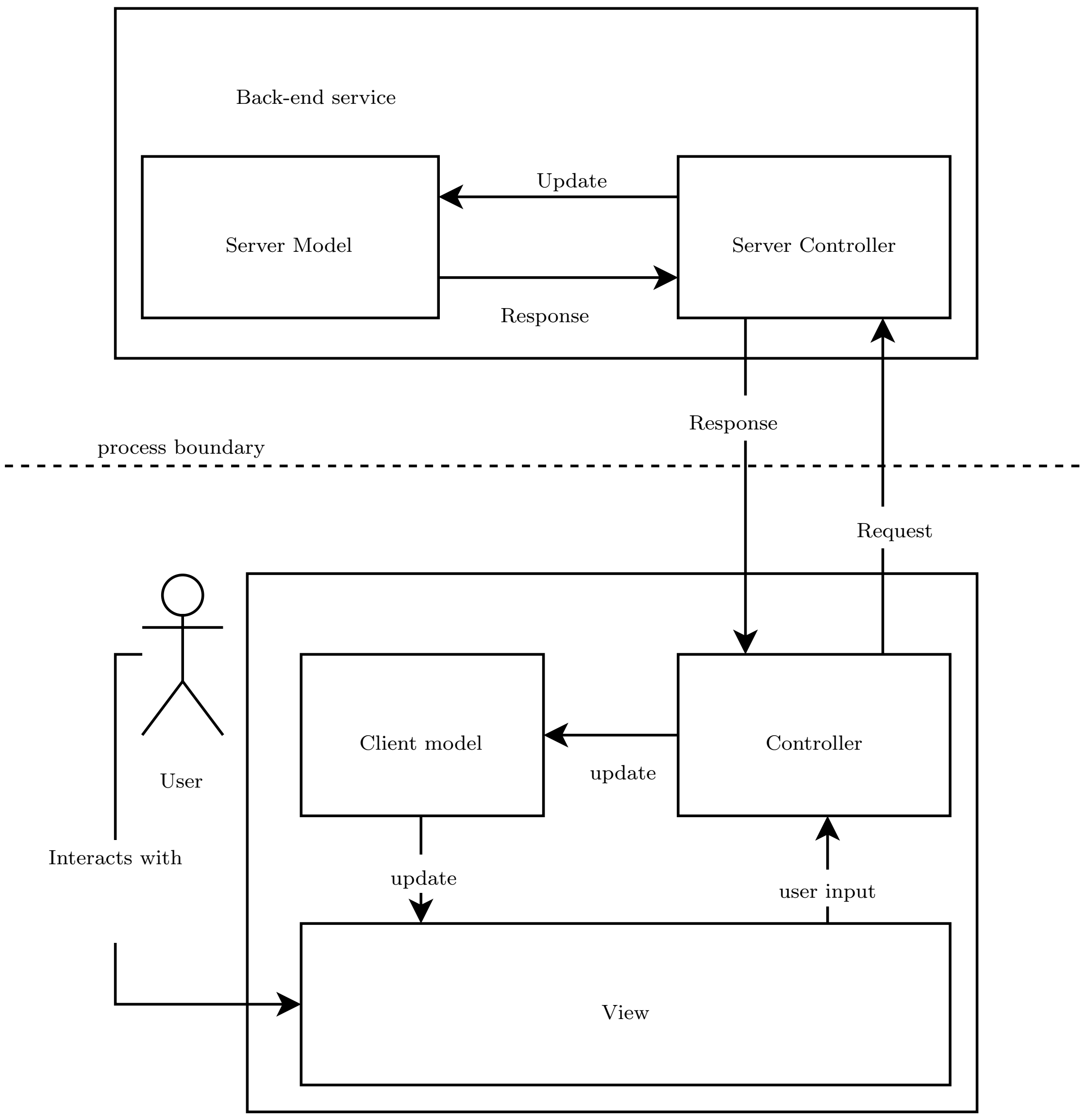

FIGURE 5.2: The MVC pattern split up into client and server.

5.3 Message bus

To enable horizontal scaling of the architecture we use the message bus pattern (Section 2.3.1.3). We choose the message bus in favor of the broker pattern because of the disadvantages discussed in Section 2.3.1.2: the broker does not scale enough because of its complexity. This message bus was also chosen because of its scalability and strongly defined responsibility within the application. Its only responsibility is to provide an interface to the clients which allow them to communicate with back-end services. It can be scaled up horizontally by adding additional messaging servers to handle message requests and API calls.

The message bus is common in architectures for SaaS applications because of its presence in the Service Oriented Architecture pattern (Section 2.3.1.4). The design shown in Figure 5.3 allows for the modular implementation of all the services and makes it relatively easy to add additional services if needed. It allows for the creation of clients for different use cases by cherry-picking the services needed for the application.

The security risks of the ESB mentioned in Section 2.3.1.3 can be mitigated by using the bus mostly as a look-up to connect clients to services. Where possible the data can be encrypted and by using a publish-subscribe protocol that monitors all subscribed components we can keep an eye out for malicious software listening in on the ESB.

The message bus functions as the communication backbone of the architecture. It is a messaging server that uses a single format to distribute messages from clients to service and back. The main responsibility of the bus is communication: This is why it does minimal processing on the messages it transfers. All clients and services communicating with the bus need to make sure to convert their data to a format the message bus understands. In our architecture the bus is mainly used to establish connections and can then be bypassed (Figure 5.4) to allow for a faster more direct connection to enhance performance where necessary. This performance increase comes from less overhead and less network hops.

The message bus now allows us to add multiple services and to make these services available to multiple clients. But what happens when a single service becomes very complex?

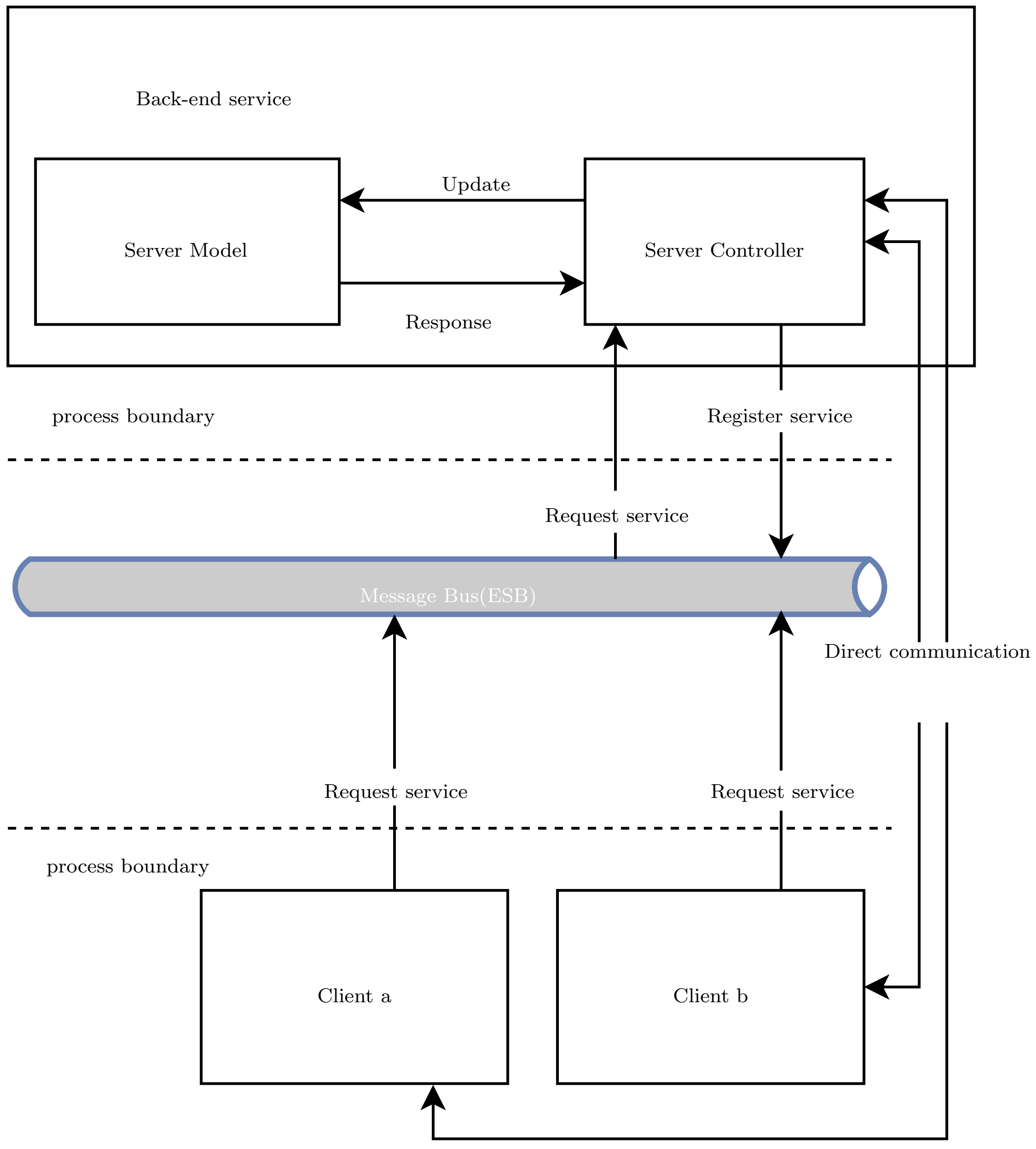

FIGURE 5.3: A message bus is added in between client and server.

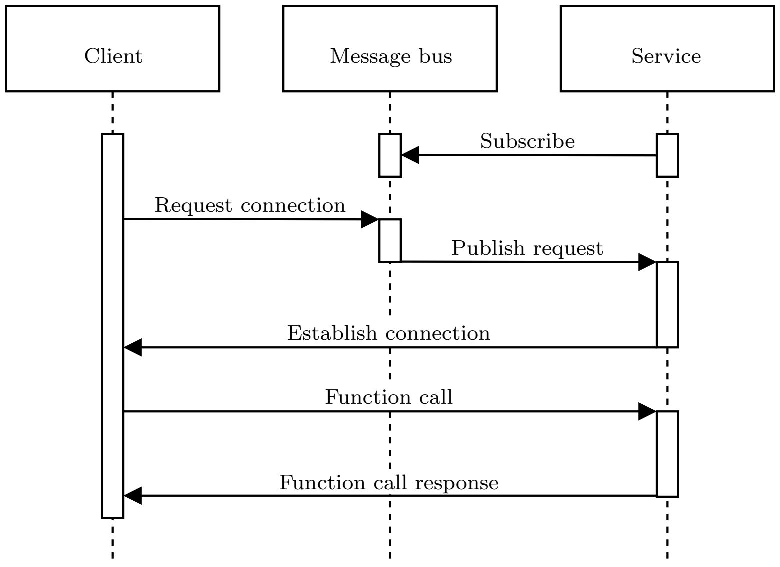

FIGURE 5.4: A client connecting to a service

5.4 Defining the service

Because the architecture requires multiple different services we introduce a new service pattern. We will use this pattern to define a service within our architecture. This pattern is used in all back-end services in the candidate architecture like the modeling service and the simulation service. Figure 5.5 shows what this service pattern looks like.

The server model has been scaled up horizontally by introducing the option to have additional servers. These servers are managed by the server controller which performs load balancing and which creates and deletes server resources as needed. The reason this happens within the service itself is to keep the message bus load as low as possible by delegating the task of load balancing to the specific service required. When a client has been assigned a server by the server controller it can proceed to communicate with this server directly. This process is shown in Figure 5.6.

Next we look into the three services we need for the MSaaS architecture. These services are all using the service pattern defined in this section.

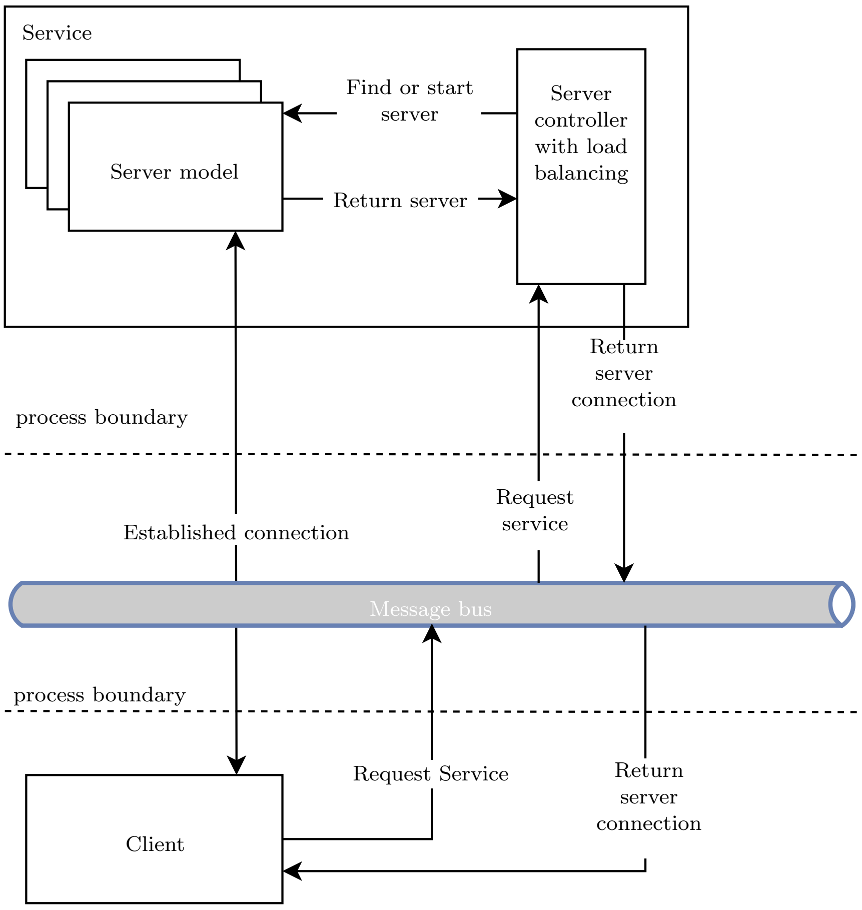

FIGURE 5.5: Scaling the back-end horizontally.

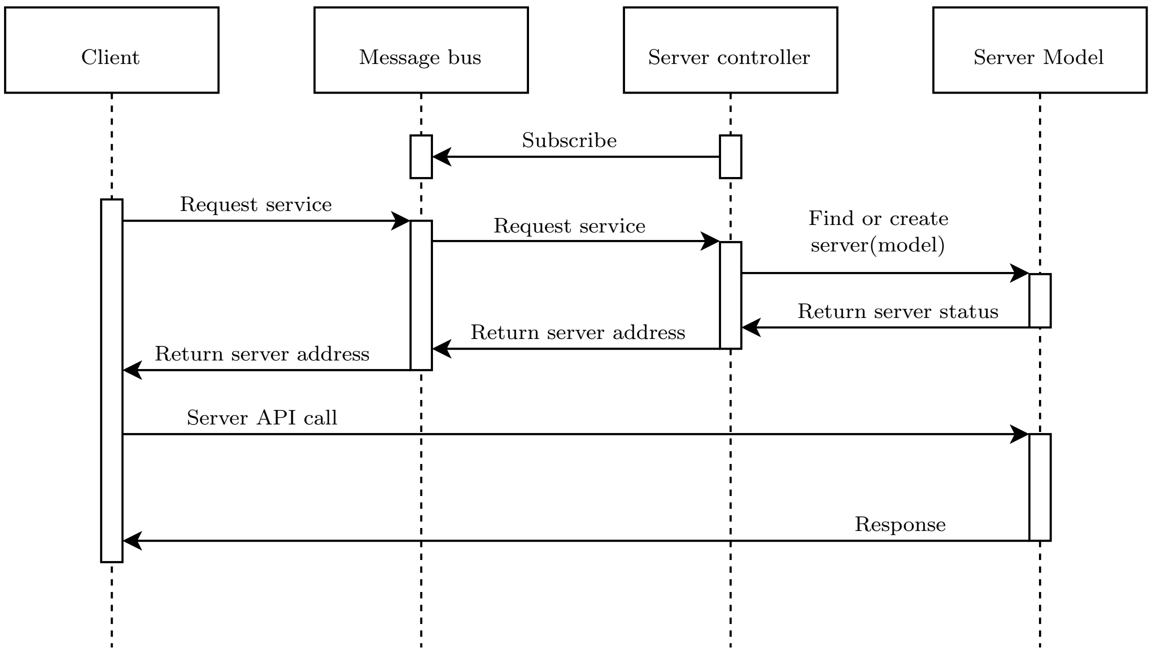

Figure 5.6: Sequence diagram of a client requesting a service.

5.4.1 Modeling service

The modeling service has the responsibility of processing 3D-environments into navigation meshes that can be used in a simulation. Because we want a simulation to be adjustable while it is running in real-time, we need to have a real-time modeling service available. The client can request a modeling server through the messaging bus API. The messaging bus then passes the request to the modeling controller which will then create or assign a modeling server using the load balancer. After processing has finished the modeling server outputs the processed file which is sent back to the client.

Because the process of converting 3D data into navigation meshes can be very computationally expensive, the load balancer could distribute a single request over multiple servers if this is more efficient and the implementation of the service allows it.

5.4.2 Simulation service

The simulation service works similarly to the modeling service described in the previous section. The main difference is that the service does not work using a request and response protocol. Instead, it establishes a connection between server and client which will push simulation data to the client viewer in a steady stream. The client will still use the message bus API when pushing changes to the simulation parameters or the model, because these might need additional info from other services or additional processing by other services.

5.4.3 Data management and authorization

The data management and authorization service is responsible for most of the administrative tasks of the MSaaS application. It handles authentication of users and maintains session data. This service maintains a database of offline data for the users so they can have persistent data storage between sessions. This is were results of past simulations or preprocessed navigation meshes will be stored for later use.

5.5 The complete picture

The final picture can be found in Figure 5.7. The three services are connected to the bus and have the option to engage in a direct connection with a client after authorization and being linked through the message bus API.

The architecture is now scalable and modular. We can increase processing power vertically and horizontally on all layers and adding another service only requires updates in the client and the message bus to support the new functionality.

In the next chapter we analyze this architecture further to see if it can satisfy certain requirements from Chapter 4.

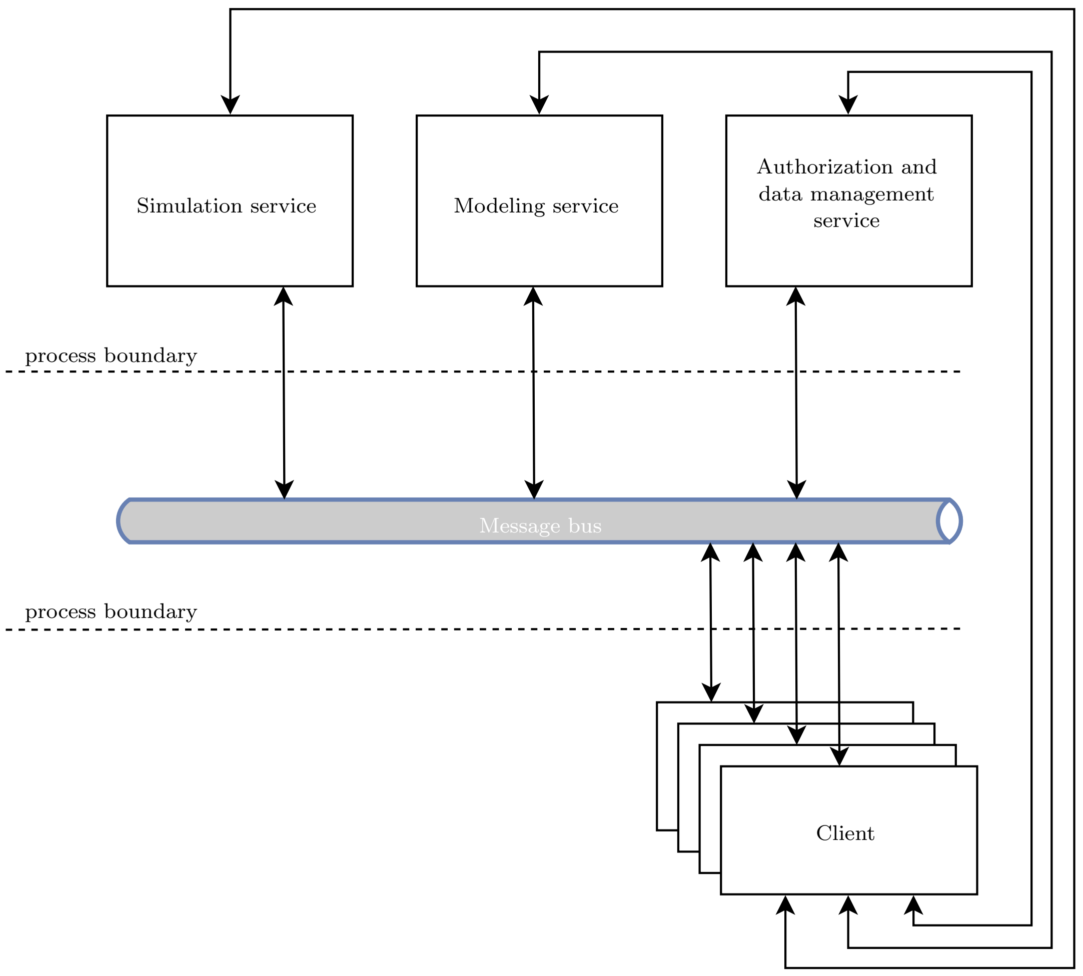

FIGURE 5.7: The complete candidate architecture.

Chapter 6

Architecture analysis

So far our look into the different quality attributes and functional requirements of the MSaaS architecture has shown us that performance is the most important attribute for our architecture. Our candidate architecture from Chapter 5 needs to deliver on this requirement because it is crucial for a SaaS implementation of simulation software to be preferable over a standalone application. The simulation part of MSaaS is the most performance sensitive which is why we test the performance of the simulation service in this chapter.

To find out if the performance of the candidate architecture holds up in practise we perform a series of tests. In these tests we analyze the amount of network traffic a simulation would generate: We look at the number of messages that need to be passed from the client to the server and back. Furthermore, we look at the size of these messages and ways to optimize this process. The tests are performed on a practical implementation of a standalone crowd simulation framework. Even though no actual networked code is involved here we can still get a good idea of the bandwidth and latency constraints by measuring the communication to an offline crowd simulation library.

6.1 Experiment introduction

In this section we specify the used software and hardware for this section and formulate variables and hypotheses.

6.1.1 software used

The software used for these experiments is a version of the crowd simulation framework by van Toll, W., Jaklin, N., & Geraerts, R. [25] which was modified to provide the information needed for this experiment. This is an agent-based simulation framework. The simulation is visualized by a front-end Unity3D implementation which uses the above framework in the form of a dynamic-link library. The used Unity3D version was 2018.3.5f1 (64 bit) for Windows.

6.1.2 Hardware used

The experiments were performed on a Lenovo Ideapad Y700 Laptop with 8 gigabytes of memory, an Intel core i7-6700HQ CPU (4 cores,8 threads) and an NVIDIA GeForce GTX 960M video card. This laptop runs Windows 10 home edition, version 10.0.17134 build 17134.

6.1.3 Variables

Two dependant variables are declared in this section: Processing time **P** and Data size **D**. After that, we discuss how we manage the independent variables.

6.1.3.1 Processing time

We quantify the Processing time **P** of a system by looking at the amount of time it takes to perform a single step of the simulation. This time is measured in system ticks which are 100 nanoseconds long on the test computer (Section 6.1.2). A single simulation step cannot last longer than 1 million ticks, or 100 milliseconds to satisfy our real-time requirement. To make certain calculations or definitions simpler, we may use 100 milliseconds instead of 1 million ticks when discussing **P**. We can get a measure of this variable using the Stopwatch class from the diagnostics namespace of the .NET framework. **P** is the total time in ticks between calling the simulation step updates and the receiving and processing of those updates. These step updates are ideally executed 10 times per second and serve as the framerate of the simulation. Note that the actual framerate of the visualization is usually higher to provide the user a smooth visual experience.

6.1.3.2 Data size

The data size variable **D** is a measure of the amount of sent data in bytes per simulation step. Variable **D** is measured using the Marshal.SizeOf() method from the System.Runtime.InteropServices namespace in C#. This method does not give us the exact size of the data at runtime. But it gives us the unmanaged memory size of the data structures used to store all the variables that need to be sent. We look at the structs that are used to send information to the simulation library and find the amount of data sent per agent.

6.1.3.3 Independent variables

The following independent variables are present as well:

The hardware and operating system used are variables in this setup which we keep consistent throughout all experiments to minimize its impact. Overhead introduced by Unity3D needs to be kept at a minimum as this could scale poorly and disrupt the experiment results at larger agent quantities. We disable any graphical rendering, but the way Unity3D structures objects still adds considerable overhead which sadly we can not exclude from the results. We have to thus keep in mind a more optimized client implementation might perform differently.

The number of agents simulated and the environment model are variables that we keep consistent through three distinct simulation scenarios so we can find out how our architecture scales to larger and more complex simulations. We call this variable for the number of agents **A**. Scenario 1 is a very basic control scenario. Scenario 2 is a more extensive look into the relationship between **A** and **P**. Scenario 3 is a stress test to provide insight into large simulations.

6.1.4 Hypotheses

To be able to reason about the effect the number of agents simulated will have on our architecture, we formulate the following hypotheses, starting with the null hypothesis $\boldsymbol{H_0}$.

$\boldsymbol{H_0}$: There is no relationship between the number of agents **A** in a scenario and **P**. $\rho_{AP} = 0$

Rejecting this null hypothesis would allow us to specify how well our modeling service can scale up with the number of agents.

The alternative hypothesis, $\boldsymbol{H_1}$ then becomes:

$\boldsymbol{H_1}$: There is a positive relation between the number of agents **A** in a scenario and **P**. $\rho_{AP} > 0$

We choose $\boldsymbol{H_1}$ to be this way because we expect the amount of time needed to increase when we simulate more agents. To get a better insight in the correctness of this hypothesis, an experiment is performed. As described in the next section.

6.2 Experiment setup

The experiment consists of measuring Processing time **P** and Data size **D** in a standalone crowd simulation implementation. We measure **P** by collecting the time a simulation step takes to complete in ticks (Section 6.1.3). We measure **D** by measuring the size of the data sent during one simulation step per agent. These variables are collected by measuring between point A and point B in Figure 6.1. For **P**, we start the timer at point A and end it at point B. For **D**, we measure the size of the data sent at A and the size of the data received at B.

We measure variable **P** for all simulation steps during 25 seconds of simulation. We can then perform statistical analysis with Pearson's correlation test to get an idea of the correlation between the number of agents and the data size. The relation between the number of agents and the processing time **D** is also analyzed by looking into the data structures used to store the data that would be sent over the network in a networked implementation. These results can be found in the next chapter.

6.2.1 Experiment details

The experiment is a standalone implementation of the Unity3D client. It contains a library that handles the simulation logic and we use calls to this library to evaluate the processing time of API calls and data sizes of traffic that would go over the network in an MSaaS version of this application. This library has an API that is exposed through an adapter to allow access to the simulation framework. To test if Requirement P1 from Chapter 4 is met, we have to time how long it takes to calculate a single simulation step. Next, we analyze the library call data sizes which we can use to validate requirement P3 from Chapter 4.

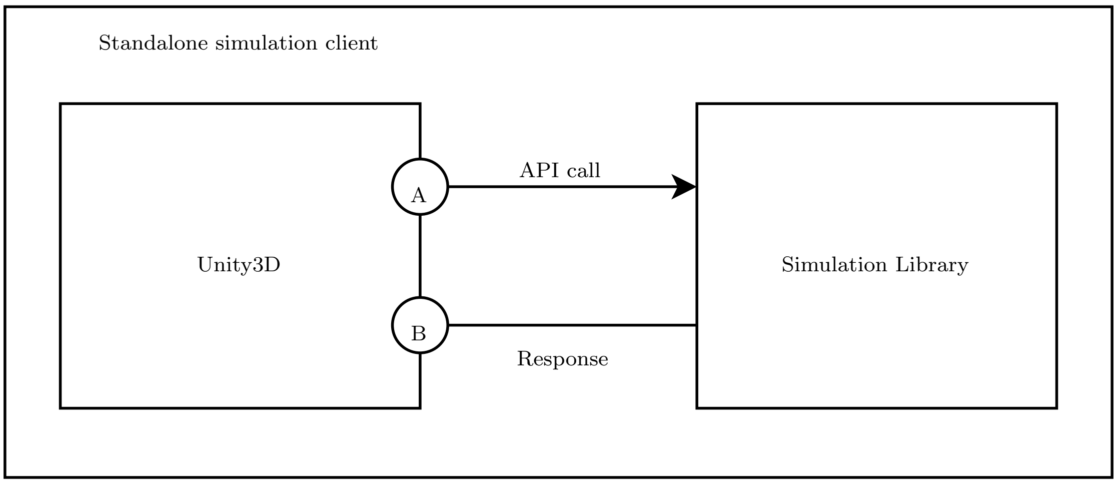

FIGURE 6.1: The experiment setup.

6.2.2 The Scenarios

The three scenarios we are testing are of increasing complexity and the experiment is performed for all three scenarios. The numbers of agents were chosen because they cover the range the used hardware can process nicely. Scenario two contains multiple measurements to provide insight into the type of relationship between **A** and **P**. Scenario 3 contains 2 stress test measurements for larger groups of agents.

6.2.2.1 Scenario 1

Scenario 1 (S1) is the least complex and contains 10 agents and 5 obstacles. The agents navigate to the position of the agent directly in front of them, swapping places. There are only 5 rectangular obstacles in the middle to make the global and local pathfinding not completely straightforward. This scenario is shown in Figure 6.2.

6.2.2.2 Scenario 2

Scenario 2 (S2) contains 10 tests where we test agent numbers of 100 agents to 1000 agents in increments of 100 agents. In this scenario the agents try to make their way to the other side of a field of 62 obstacles. This is a medium stress test for the Unity3D client. This scenario is shown in Figure 6.3.

6.2.2.3 Scenario 3

Scenario 3 (S3) uses the same environment model as S2 and contains 2 tests. The first test simulates 4800 agents and the second one simulates 14400. This scenario tests the

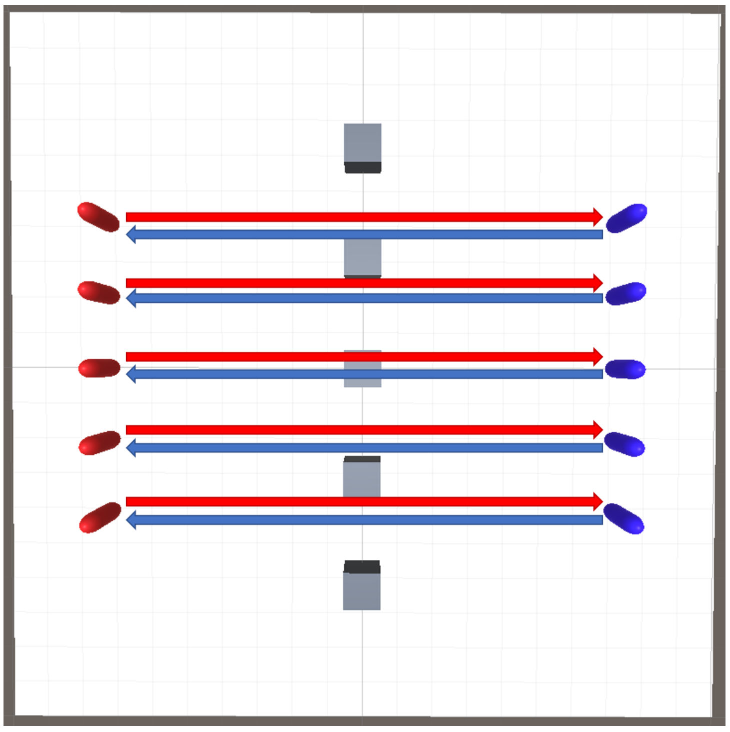

Figure 6.2: Simulation scenario 1

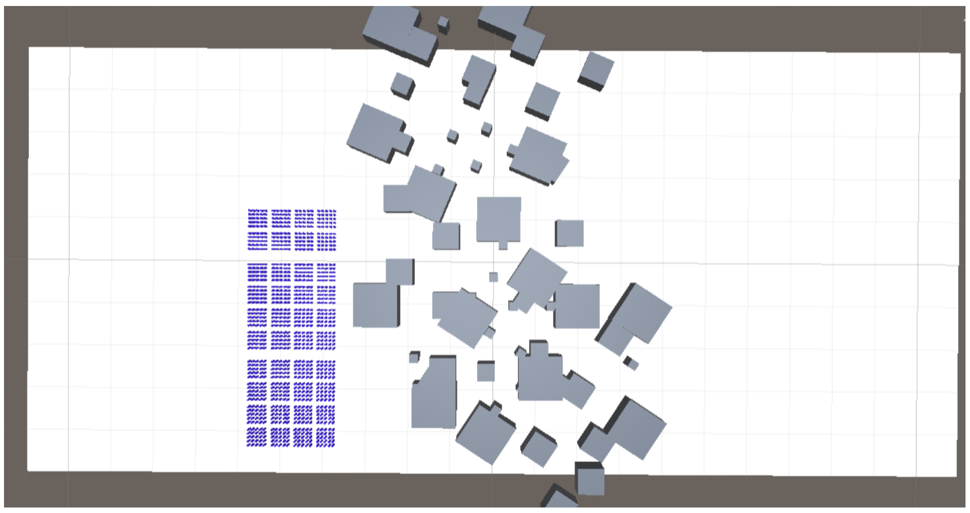

Figure 6.3: Simulation scenario 2

largest number of agents. This is a high stress test for the Unity3D client and at the limit of what the testing hardware can handle. This scenario is shown in Figure 6.4.

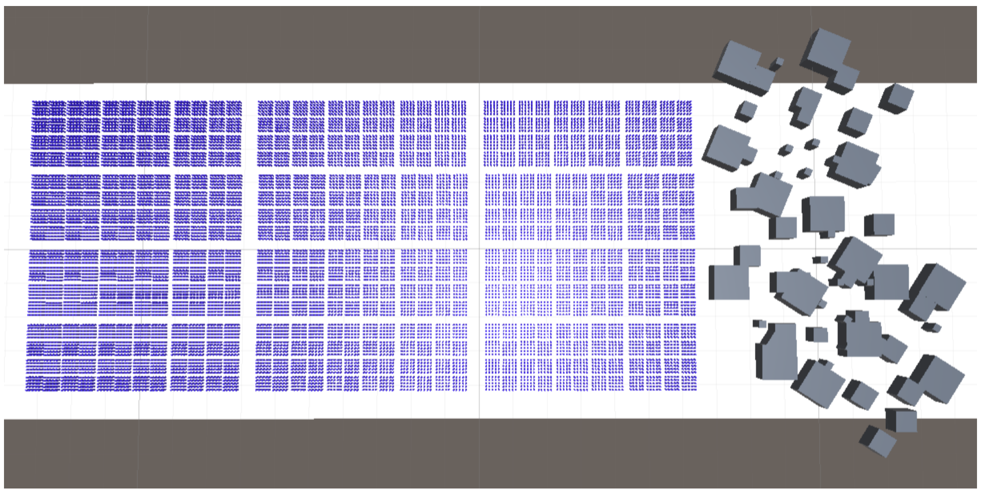

Figure 6.4: Simulation scenario 3

Chapter 7

Results

In this chapter, we discuss the results of the experiments defined in the previous chapter. First, we analyze the data usage of a hypothetical simulation service. Next, we discuss our hypotheses and the results of the processing time analysis.

7.1 Data size D results

We analyzed the data size using Marshal.SizeOf() from the System.Runtime.InteropServices namespace and by analyzing the data structures used for storing agent positions and velocities. We found that the data structures used require 104 bytes per agent. This number consists of 2 integers, 8 doubles, 2 integer16's (smaller integer) and 2 enumerable types. Note, in its normal, non-marshalled form, that this data would take up 148 bytes.

7.1.1 Network throughput

Network throughput is crucial for our MSaaS application. It is the number of bits per second that are received by the recipient of a transmission over the internet.

Assuming these 104 bytes are what we need to transfer a single agent's data, we can, in an ideal scenario, support over 12.000 agents on a 100 mbps (megabits per second) connection at a rate of 10 simulation steps per second. We get this number by getting the number of bytes in 100 megabits (12.500.000 bytes) and dividing this number by the 104 bytes of our data structure. We then divide by 10 to gives us the number of agents we can simulate at a simulation step framerate of 10 updates per second. It is possible to make use of gigabit network connections to support more agents in an MSaaS application but if we want the client software to be flexible and thus not always require a network connection of 1 gbps (gigabit per second) or more this is not an option. The throughput bottleneck in this scenario is most likely going to be the client bandwidth. This is why we assume a maximum bandwidth of 100mbps. In reality, this throughput is not going to be possible because of the overhead introduced by the network transmissions. In the next section we discuss what the impact of this overhead is and how to deal with it.

7.1.2 Network Protocol

Commonly a network connection is set up using a TCP (Transfer Control Protocol) connection. TCP is a reliable transport protocol that guarantees that data arrives at the destination without errors. This is useful when reliability is crucial for example when loading web pages (HTTP uses TCP). But when streaming data TCP also performs congestion control which means it limits the maximum throughput when packets are lost. UDP (User Datagram Protocol) can stream at the maximum bandwidth and does not limit itself. The drawback of this is that it does not guarantee packages to be correct or even that packages arrive at their destination at all.

How does this apply to our MSaaS architecture? Both UDP and TCP add network overhead to the packages they send. Additionally, TCP adds another bottleneck for our throughput because of its congestion control. UDP adds 20-60 bytes of IP address information in the header and it always adds 8 more bytes UDP header. The Maximum Segment Size (MSS) on the ethernet is 1500 bytes [26] which means a single package cannot be smaller than 1500 bytes minus the 68 byte header. The MSS is limited further by the Maximum Transmission Unit (MTU) which can be as low as 576 bytes per packet which is the minimum number of bytes for the IPv4 packet any device must be able to receive [27]. This means, that to be sure the packet is not fragmented, which causes additional unwanted overhead, the safest segment size becomes 508 bytes after subtracting the UDP and IP header size. Taking into account UDP overhead and agent data size this gives us 10.600 agents that we can transmit data for in 100 miliseconds.

If we want to simulate bigger numbers of agents in real-time our best bet is to decrease the amount of data being sent for each agent. The 104 bytes needs to go down to 22 bytes per agent to simulate 50.000 agents in real-time for example. Using smaller more efficient data types where possible is advised. For example, in the tested implementation the double values could be replaced by float values or integers. An example bandwidth-friendly approach would be: 4 floats (for the positions and velocities in 2D), and 2 integers (agent identifier and layer number). Which would be 24 bytes per agent unmarshalled. Another way to optimize the bandwidth is to cull the agents that are not being viewed in the client from the step updates. Lastly, we could send a stream of a video that was generated on the server side instead of the large number of agent velocity and position updates.

UDP gives us performance improvements if well implemented and optimized for crowd simulation data structures. The tradeoff is the unreliability of UDP packets: they can get lost, corrupted, delayed or desynchronized [26]. The unreliability can be masked by animations on the client side to a certain degree, and can be further mitigated by clever predictions if the client hardware can handle it.

If unreliability is not an option, TCP can be used at the cost of performance. Because of the throughput throttling this will be less performant than UDP. It is difficult to predict the exact number of agents this can support because of this throttling. To get the best of both worlds one could use TCP until the throughput becomes so small 10 updates per second cannot be guaranteed anymore. At which point the user has to choose between switching to UDP or running the simulation at a slower step frequency.

For any communication that is not as throughput sensitive as the real-time simulation step updates it is recommended to use TCP because of its reliability.

7.2 Processing time P results

The results of the experiment described in the previous chapter provide us with a benchmark of the framework we tested. Agent based simulation approaches tend to get slower as more and more agents are added to the simulation. The results provided in this section give us a better understanding.

7.2.1 Scenario 1

From our basic control scenario with 10 agents we learn that in the 25 second test run the simulation library was called 1057 times. Most of these calls come from three methods that are called every simulation step. This means they make up 750 of those calls (3 times 25 calls per 100 milliseconds). These are the calls of which we have already analyzed the data size in Section 7.1; they contain positions and velocities of agents. The rest are functions that are run at the start of the simulation to set up the environment and the simulation parameters. An example is setting the walkable areas and initiating the agents.

7.2.1.1 Outliers

These setup functions make the first simulation step take a little bit longer than usual which is to be expected. We omit this first simulation step from further analysis as it is an outlier. Another outlier we find is a very periodic and consistent spike in **P** as seen in Figure 7.1. These consistent spikes can also be observed in the Unity3D profiler. The exact cause of this is unknown because it happens in the simulation library. Because the server ideally optimizes these spikes we treat these spikes as outliers as well. All outliers were omitted from further analysis.

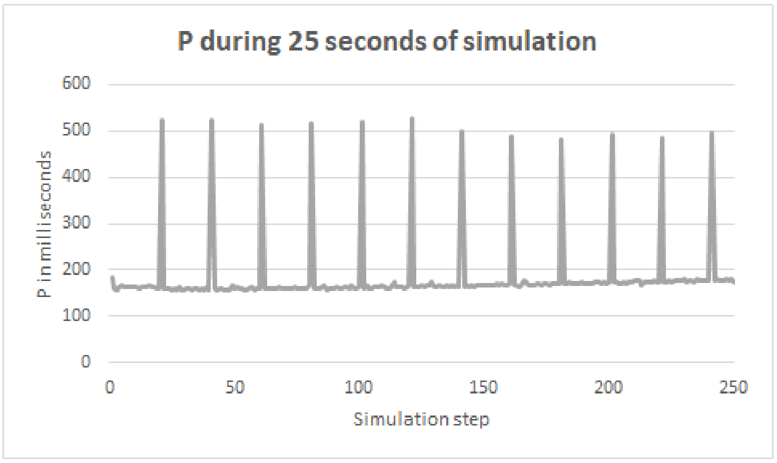

FIGURE 7.1: Spikes observed in P for 14400 agents.

7.2.2 Scenario 2

For scenario 2 we gathered data from 10 different agent numbers: 100, 200, 300, 400, 500, 600, 700, 800, 900 and 1000 agents.

Agents	Median P(milliseconds)
100	0,2054
200	4,5717
300	6,287
400	8,2047
500	10,0239
600	11,9581
700	13,4194
800	14,7018
900	17,3329
1000	18,2811

TABLE 7.1: Median **P** values for different agent numbers.

Applying Pearson's correlation test on our data gives us $\rho = 0,991733927$, which is a very strong positive linear correlation. This linear correlation becomes apparent in

Figure 7.2. The processing time for 100 agents is much lower than expected. From 200 agents upward, **P** follows a linear pattern.

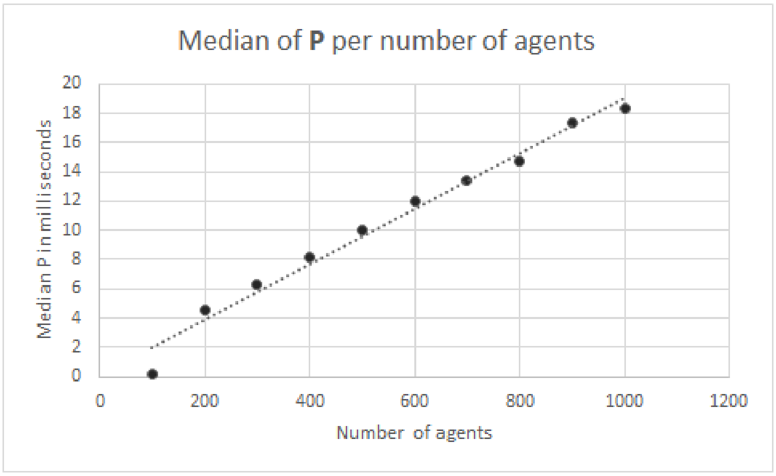

FIGURE 7.2: Median **P** measurements presented in a graph.

This linear correlation is helpful for our architecture design because it lets us very precisely pinpoint the maximum agent capacity that can still run in real-time. In the next section we look at some bigger agent numbers to see if this correlation holds up.

7.2.3 Scenario 3

Two larger quantities of agents were tested in Scenario 3; one with 4800 agents and one with 14400. Figure 7.3 shows these tests in relation to the control test. This graph is presented by using a logarithmic scale. We can see that 4800 agents simulate in just under our 100 millisecond benchmark. The 14400 agent test violates our benchmark by taking consistently longer than 100 milliseconds to process all agents. In Figure 7.4, we zoom in on the 14400 agent scenario where we see that processing time **P** climbs during the simulation. The agents interacting with each other and the environment (collision avoidance) might be causing this. But that cannot be claimed with certainty based on these results alone. It is however a pattern that exists for all test data and is worth looking into more as there could be room for optimization.

Agents	Median P(milliseconds)
4800	59,6152
14400	164,7193

TABLE 7.2: Median **P** values for larger agent amounts.

As seen in Figure 7.5, the data for larger agent numbers seems to follow our linear trend. Pearson's test confirms this with a positive correlation of $\rho = 0,998753983$.

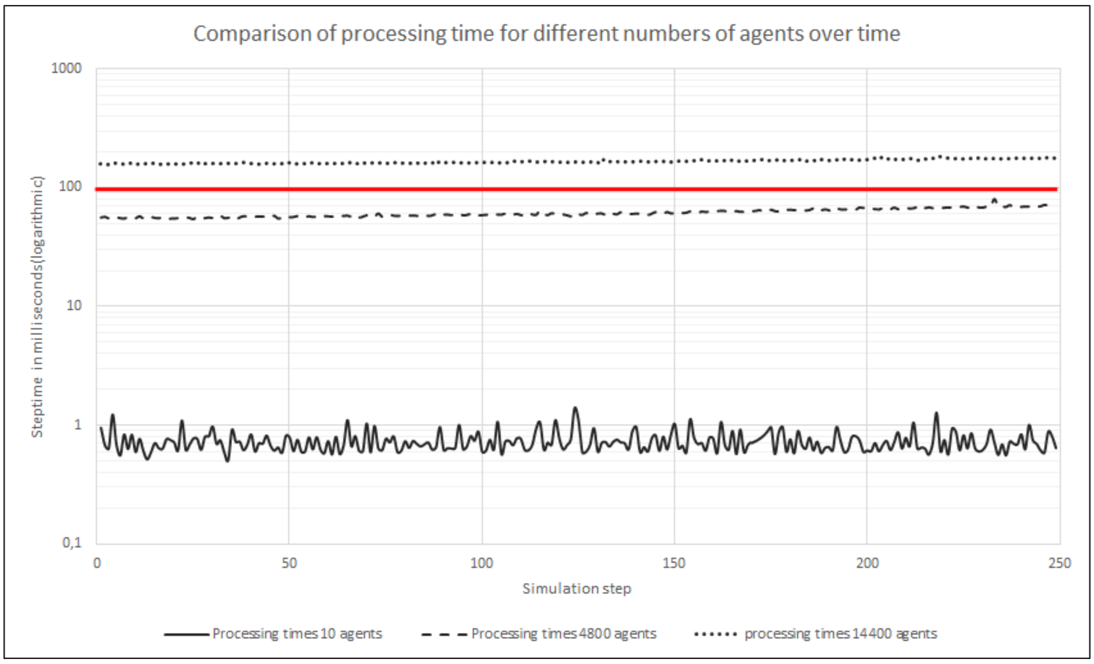

FIGURE 7.3: **P** measurements over time for three agent numbers presented in a graph.

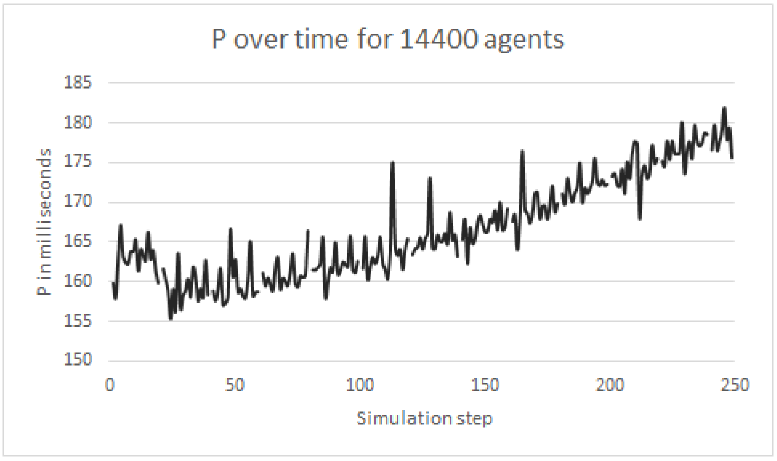

FIGURE 7.4: **P** measurements over time for 14400 agents.

With this evidence we feel safe to reject $\boldsymbol{H_0}$ and accept $\boldsymbol{H_1}$. This means there is a very strong linear correlation between agent numbers and the time the simulation library takes to process a single simulation step. Figure 7.5 shows that we can safely assume the maximum number of agents our test hardware can compute in real-time is slightly over 8000 agents.

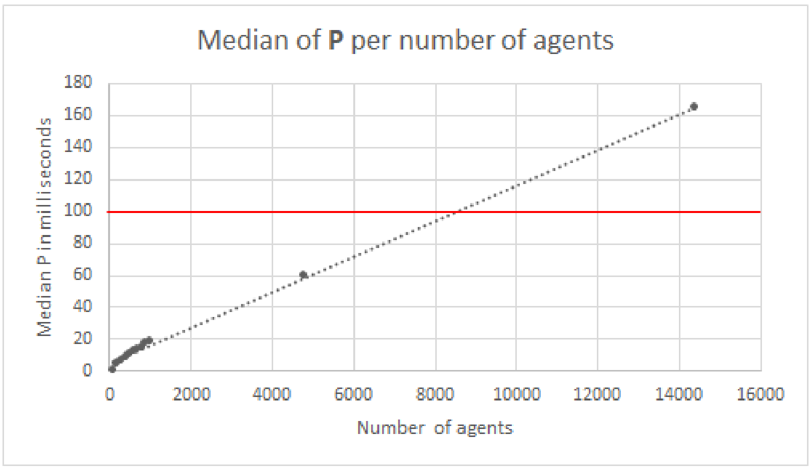

FIGURE 7.5: Median **P** measurements presented in a graph.

Chapter 8

Conclusion

This chapter concludes the thesis by discussing the implications of our results, by summarizing what we have presented and by looking forward towards possible future research.

8.1 Research contribution

In this thesis we have provided an overview of literature for both crowd simulation and computer architecture. We used this knowledge to set requirements for an MSaaS system which resulted in the design of a candidate architecture (Chapter 5). We then analyzed this architecture to learn more about its most important quality attribute: Performance.

In Chapter 3 we formulated a few problem statements that we intended to answer. Here they are listed once more:

Problem 8.1. What architectural decisions satisfy our set requirements when processing environment models and why?

Problem 8.2. What architectural decisions satisfy our set requirements when running a crowd simulation in real-time and why?

Problem 8.3. What architecture for an MSaaS application satisfies the most of our set requirements?

We presented a candidate architecture in Chapter 5. This architecture was constructed using well known and documented architectural patterns which were chosen based on their strengths and weaknesses. The resulting design provides a solid guideline for future MSaaS implementations. Furthermore we analyzed the most important Performance issues in Chapter 6 and 7.

Problem 8.4. How can we evaluate our proposed architecture in a meaningful way?

Evaluating architectures is a lot more abstract than evaluating implementations. In this thesis we intended to provide reasoning and alternatives where possible so that a critical reader can make their own informed judgments based on their own specific needs and requirements.

8.2 Discussion of experiments

Our analysis of an existing crowd simulation library and client provided us with a few important insights. It showed us that the maximum number of simulated agents, for an agent-based crowd simulation system, is limited largely by the network throughput. This means it is critical to keep data sent per agent to a minimum for real-time simulations if very large quantities of simulated agents need to be supported. This also validates our assumption that these real-time simulation data streams should bypass the central messaging bus (ESB) because the overhead and massive amounts of traffic this would introduce would result in very poor network performance. Alternative ways of streaming the agent data can also be considered. Instead of streaming all the positions and velocities for the client to animate, a subset of this data could be selected based on the needs of the client view. Another option would be to do all animation on the server side and to stream only a rendered video of the simulation output.

When looking into the relation between the number of agents and the processing time required we found a positive linear correlation. By improving the hardware by running these calculations on a powerful cloud service we can predict that this will drastically improve the maximum number of agents simulated for clients run on less powerful machines as long as the internet connection is decent.

8.3 Discussion of research method

For our analysis it was not feasible to test a (partial) networked implementation of our MSaaS architecture due to scope constraints. We only analyzed a non-networked standalone version of a crowd simulation application. We hope the performance analysis can help future software architects make more informed decisions regarding MSaaS architectures.

8.4 Future work

8.4.1 Implementation

Given our candidate architecture the next logical step is implementing the architecture and analyzing the more practical problems faced in MSaaS development. An implementation provides an environment where our design choices can be validated and researched.

8.4.2 Security

Another area of interest is studying the other quality attributes that we did not analyze as thoroughly as a part of this thesis. The most prominent one is security. How does security impact performance? Which parts of the architecture are most vulnerable? It is an interesting challenge to find a way to secure the large data streams required for real-time simulation without compromising the performance too much.

8.4.3 Non-agent based systems

Our analysis focused on an agent based crowd simulation system. How do other systems perform in an MSaaS setting? Can other systems handle more agents in real-time? Can other systems provide the same functionality as agent-based systems? Flow-based and particle-based systems are only two examples of simulation algorithms that are not agent based. These systems excel at huge simulations but lack complex agent behaviours.

References

[1] Dirk Helbing and Anders Johansson. *Pedestrian, crowd and evacuation dynamics.* Springer, 2009.

[2] Team MSG-131. Modelling and simulation as a service: New concepts and service-oriented architectures. 2015.

[3] Tiago Azevedo, Rosaldo JF Rossetti, and Jorge G Barbosa. Densifying the sparse cloud simsaas: The need of a synergy among agent-directed simulation, simsaas and hla. *arXiv preprint arXiv:1601.08116*, 2016.

[4] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH computer graphics*, volume 21, pages 25–34. ACM, 1987.

[5] Dirk Helbing and Peter Molnr. Social force model for pedestrian dynamics. *Physical review E*, 51(5):4282, 1995.

[6] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Transactions on Graphics (TOG)*, 25(3):1160–1168, 2006.

[7] Roger L Hughes. The flow of human crowds. *Annual review of fluid mechanics*, 35 (1):169–182, 2003.

[8] Suiping Zhou, Dan Chen, Wentong Cai, Linbo Luo, Malcolm Yoke Hean Low, Feng Tian, Victor Su-Han Tay, Darren Wee Sze Ong, and Benjamin D Hamilton. Crowd modeling and simulation technologies. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(4):20, 2010.

[9] Artur Malinowski, Paweł Czarnul, Krzysztof Czuryo, Maciej Maciejewski, and Paweł Skowron. Multi-agent large-scale parallel crowd simulation. *Procedia Computer Science*, 108:917–926, 2017.

[10] Armel Ulrich Kemloh Wagoum, Bernhard Steffen, Armin Seyfried, and Mohcine Chraibi. Parallel real time computation of large scale pedestrian evacuations. *Advances in Engineering Software*, 60:98–103, 2013.

[11] Santosh Kumar and RH Goudar. Cloud computing-research issues, challenges, architecture, platforms and applications: a survey. *International Journal of Future Computer and Communication*, 1(4):356, 2012.

[12] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-oriented cloud computing architecture. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 684–689. IEEE, 2010.

[13] Robert Siegfried, Tom van den Berg, Anthony Cramp, and Wim Huiskamp. *M&S as a Service: Expectations and Challenges.* Orlando, FL: SISO, 2014.

[14] Wouter Van Toll, Atlas F Cook, and Roland Geraerts. Navigation meshes for realistic multi-layered environments. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3526–3532. IEEE, 2011.

[15] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in Practice.* Addison-wesley, 2014.

[16] Mozilla. Mvc architecture, Jun 2018. URL https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture.

[17] Siqi Shen, Shun-Yun Hu, Alexandru Iosup, and Dick Epema. Area of simulation: Mechanism and architecture for multi-avatar virtual environments. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 12 (1):8, 2015.

[18] Microsoft. Broker, Mar 2014. URL https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648096(v=pandp.10).

[19] David Chappell. *Enterprise service bus.* O'Reilly Media, Inc., 2004.

[20] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000.

[21] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*, pages 12–12. IEEE, 2007.

[22] Jason McColm Smith. *SPQR: formal foundations and practical support for the automated detection of design patterns from source code.* University of North Carolina at Chapel Hill, 2005.

[23] Arne Hillebrand, Marjan van den Akker, Roland Geraerts, and Han Hoogeveen. Separating a walkable environment into layers. In *Proceedings of the 9th International Conference on Motion in Games*, pages 101–106. ACM, 2016.

[24] Iso/iec 25010. URL `http://iso25000.com/index.php/en/iso-25000-standards/iso-25010`.

[25] Wouter van Toll, Norman Jaklin, and Roland Geraerts. Towards believable crowds: A generic multi-level framework for agent navigation. 2015.

[26] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*. Addison Wesley, 2011.

[27] Jon Postel et al. Rfc 791: Internet protocol. 1981.