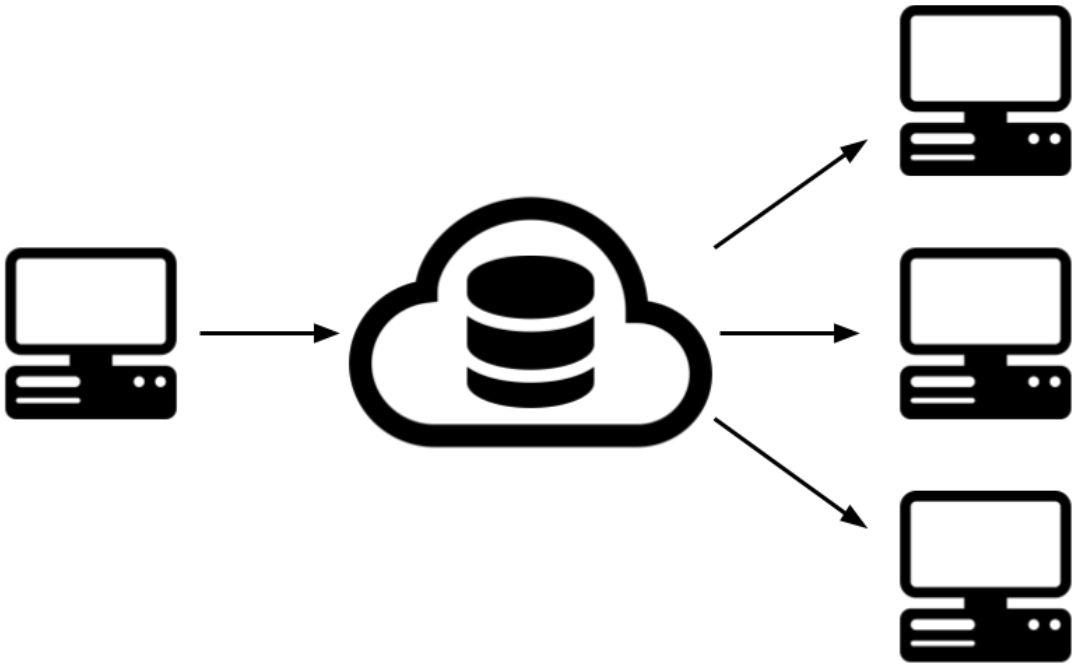


Cost Model for Scalable Containerized Relay Game Servers

The change in cost as a server needs more containers to meet client demands.

Master's thesis in Computer science and engineering

SABINE RANDOW

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Cost Model for Scalable Containerized Relay Game Servers

The change in cost as a server needs more containers to meet client demands.

SABINE RANDOW

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Cost Model for Scalable Containerized Relay Game Servers
The change in cost as a server needs more containers to meet client demands.
SABINE RANDOW

Supervisor: Muhammad Waqar Azhar, CSE
Advisor: Erik Möller, Opera Software
Examiner: Pedro Petersen Moura Trancoso, CSE

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Depiction of clients communicating via a relay server running in the cloud. Created using bootstrap icons.

Typeset in LaTeX
Gothenburg, Sweden 2023

Cost Model for Scalable Containerized Relay Game Servers
The change in cost as a server needs more containers to meet client demands.
SABINE RANDOW
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Previous research has looked into how to dynamically scale containerized applications with consideration to clients Quality of Experience (QoE), but there is a lack of knowledge on how relay server's, used for multiplayer games, scale. Through investigating the largest cloud platform providers, three key metrics were identified: Network traffic, memory allocation and CPU utilization. These metrics were investigated depending on several parameters: the clients perceived frequency of messages, the number of clients connected to a server, and the number of clients playing the same game. To do this, a client-simulator program was expanded to work with a pre-existing server developed by Opera Software. The server and client-simulators were used in different environments, both bare-metal machines and containers from Amazon Web Services. Upon analyzation it was found that network traffic and memory allocation scales linearly, while the CPU utilization can only be interpolated withing a range used to train a third degree regression model. The error of all models were fairly low at a maximum of 2,658%.

Keywords: Computer, science, computer science, engineering, project, thesis.

Acknowledgements

Thank you Erik Möller and Opera Software for providing supervision and a server to work with. Thank you to Pedro and Waqar at Chalmers for your academic support. Thank you to all my friends, especially thoe also writing their thesis (Erik, Love, William and Victor) for the times we did not work and instead took some well deserved breaks. Thank you Simon for cooking food when I've been the most stressed and to my family (Åsa, Mark and Max) for your support during my life until now, without you I would not have completed my education.

Sabine Randow, Gothenburg, 2023-06-16

Contents

Glossary

AWS Amazon Web Services. vi, 1, 4, 7–10, 14, 16, 22, 23, 29, 32, 38–42, 44, 45

CPU% CPU Utilization. xviii, 10, 11, 13, 17, 18, 23, 25, 34–36, 38, 41–44, XIII

E2 Elastic Compute Cloud. 8, 10, 38, 44

EC2 Elastic Compute Cloud. 8, 9, 42, 44

GCP Google Cloud Platform. 8, 10, 38, 39, 44, 45

IaaS Infrastructure as a Service. 7

OS Operating System. 10, 15, 16, 22

PaaS Platform as a Service. 7

PID Process Identification. 16, 30

QoE Quality of Experience. vi, xvii, xviii, 1–4, 11–14, 18–22, 26–28, 34, 35, 37, 39, 41, 43–45, XIII

QoS Quality of Service. 1, 11

r2 R-Squared value. xvii, xviii, 34, 35, XIII

SaaS Software as a Service. 7

vCPU Virtual Central Processing Unit. 9, 10, 17, 23, 39, 42

VPS Virtual Private Servers. 7

WSL Windows Subsystem for Linux. 18, 23, 42

List of Figures

List of Tables

1

Introduction

A client-server model is an architecture made up of (at least) one client and (at least) one server, where the client sends requests and the server responds [1]. Some examples of client-server systems are the Mail Transfer Protocol and Hypertext Transfer Protocol, that many people use on a daily basis. Therefore, an efficient server structure is needed for good performance, and tools to utilize it fully.

One specific area of client-server system is used for multiplayer games. It requires tracking of multiple things like what clients are connected and which clients are connected to each other, for e.g., a room. There also exist specific Quality of Service (QoS) constraints associated with multiplayer games, like frame age and the rate of communication, which impact the players Quality of Experience (QoE) [2].

A quick and performant multiplayer game server model is essential to many people today. This can be seen when observing Steam, a platform for playing computer games used by millions of clients. They recently reached a new record of 30,795,186 simultaneous online users (reached 11/6/2022), and their top four played games are online multiplayer games [3].

In order to reduce costs, multiple gaming companies choose to use cloud providers to host their game servers. These cloud providers rent out networks of computational resources. According to Amazon, the studio Epic Games, who created the popular multiplayer game Fortnite, have since 2018 exclusively used Amazon Web Services (AWS) for hosting its multiplayer servers (which has had as many as 15.3 million concurrent players) [4]. Another big studio who use AWS is Ubisoft who have published many games including: Assassin's Creed, Watch Dogs, Farcry and many more. Smaller game studios like the Swedish game studio Fatshark, who have published the game Warhammer 40,000: Darktide, also use AWS. We can see that cloud providers are a popular choice for hosting multiplayer game servers.

But why do servers need to be scalable? Well, the number of players in a game does not remain constant over time. This can be seen by using Steam Charts, which displays player traffic over a set period of time for different games. When looking at the game *PUBG: BATTLEGROUNDS*, between the dates of April 9 – April 17 in the year 2023 [5], the chart in figure 1.1 (with time on the x-axis and player-activity on the y-axis) is provided. It becomes evident that player traffic is not necessarily constant over time. And it is understandable that more resources are needed during peaks than troughs.

Figure 1.1: Player activity over a week for the game *PUBG: BATTLEGROUNDS*.

The easy solution to cater to fluctuating client demand would be to constantly have resources able to cater to peak hours. But this would lead to many computing resources going unused during troughs hours. Thus, it would be logical to scale down the resources allocated to the game during troughs to save money, and to scale up resources during peaks to maintain client QoE.

Using cloud providers is not only possibly good for saving money, they can also be great for the environment. The companies that provide cloud services (amongst many, Google, Amazon, and Microsoft) individually work towards minimizing their data centers environmental impact [6]. Their motivation comes partially from that cooling data centers can amount to as much as 70% of the total expenses of running their services. Decreasing the energy needed for cooling data centers would not only save money for the cloud provider companies, it would also help decrease the negative environmental impact of data centers, which currently are accountable for about 2% of the world's total energy consumption.

Previous work has been done to create scalability-models for different types of containerized applications, which is discussed further in section 2.5.1. But so far there exists no model focusing on multiplayer relay servers, at least I can not find any prior work for this specific server structure. The main purpose of a relay server scalability model, for multiplayer games, could be to enable early development testing of game servers on smaller scales. A developer could use the model to determine what constellation on cloud provider containers to use for a certain test case, before scaling their server for production. The point is that the model would make it easier for game developers to evaluate their server-hosting options while spending less time on researching cloud providers.

This thesis firstly discusses the limitations and goals of the thesis in this chapter. Then chapter 2 covers the necessary theory and background required to understand the problem and its solution. Some background topics are multiplayer relay game servers, containerized applications and previous related research. This is followed by chapter 3 which describes the developed tool for testing, environments for testing and metrics that have been measured to create a scalability-cost model. Then the results are provided in chapter 4 followed by thoughts on future work required to

improve on the model in chapter 5, and the conclusion follows in chapter 6.

1.1 Aim and Objectives

Several papers describe scalability models for containerized applications, but none specific to the area of relay game server. This area could be interesting given its specific structure of rooms and QoE parameters specific to gaming. It could be used by smaller and larger multiplayer games. Thus, the aim of this thesis is to provide the basis of a scalability-cost model that can predict future needs of a server with a set of parameters, for a given set of container-resource-configurations.

The model can not consider every possible parameter in the time-frame of this thesis. Thus, the aim is for the developed scalability-cost model to be modular so that future work can be done to expand it and increase its accuracy. As an initial demand, the developed scalability model should depend majorly on the frame rate that clients connected to a game expect. Key metrics connected to the server structure are identified, as well as the demand on resources that different frame rates require.

To emphasize the modular aspect of the scalability-cost model, sub-models should be created that define requirements for the final model. These sub-model should cater to different metrics of importance when choosing between cloud provider containers. Thus, the thesis has to find and motivate what metrics are of importance. Then the sub models should be able to combine in different ways to cater to different cloud providers.

1.2 Limitations

The time for this thesis was limited. Thus, there was a need to develop a scalability-cost model specifically for relay game servers that is modular and can easily be built on in the future. The challenge with this was that no two relay game servers function in the same way, and there exist many options of cloud computing resource configurations. Other limitations were the following:

- The tools used to gather data were written in C++20 since it is the version that existing code uses, which was adapted to the test-setup described in section 3.2 and used to gather data for the scalability-cost model. Although, the logic of the setup will still be explained so that it can be replicated in other programming languages.
- UDP was used for communication between clients and the server. Thus, it did not matter if datagrams are lost in transmission. If losses happen irregularly, they should not be noticeable to the clients and thus not affect their QoE. Using TCP might give other results since, for example, the two protocols have different header lengths, thus they affect network traffic differently, which discussed in section 3.1.1.
- Not all cloud platform providers can be tested, neither can all containers from

one provider be tested. The thesis is thereby limited to a selection of instances provided by AWS. It was chosen specifically because the thesis was done in collaboration with Opera, who will use AWS for their servers in the future. It offers cheap containers, and even has some resources for free for a limited time.

- At a larger scale, there would be multiple instances of a server in service, and clients would go through additional steps to connect to a room (discussed in section 3.2.2. This thesis only used a single server to which clients connected. But multiple instances had to be used to simulate clients, since one client-simulator could not create an infinite amount of clients before it crashed (presumably due to clients sharing resources).
- All functionality of the game server were not considered or tested, because the focus was solely on maintaining a frame rate, QoE, for all clients. There are many services that a server can provide, but the scope would become too large for this thesis if every service was to be included. Thus, the modularity of the final scalability-cost model is important to enable others to add their additional features, parameters, and metrics.

2

Theory

This thesis looks into the scalability of containerized relay game servers. To understand what has been done, a basic understanding of containerized applications, different cloud computing providers, the network traffic created by relay servers and importance of avoiding frame delays is needed and therefor discussed in this chapter.

2.1 The Basics of a Relay Server

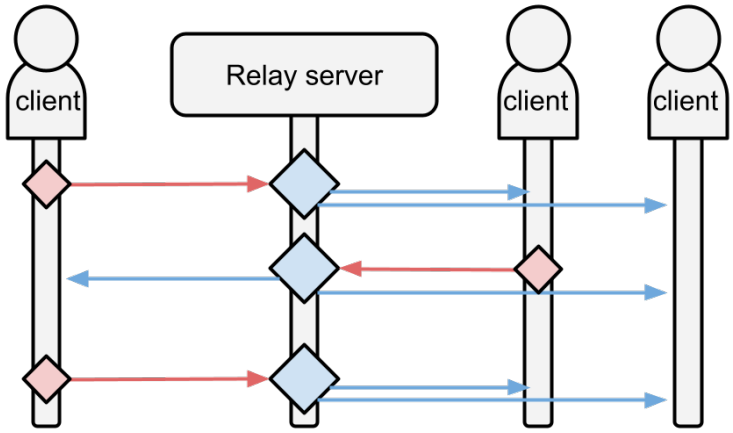


Figure 2.1: Time diagram of interaction between the relay server and clients using the same room.

This thesis was done in collaboration with Opera and therefore adapted to their relay server structure. To describe the structure, some intertwining terminology needs to be established:

- **Room**: A room is created for every instance of a multiplayer game. Clients can create rooms and connect to them using an invitation from other clients. Packets (frame reports) sent from a client are distributed to all clients in a room.
- **Client**: A client is a computer used to play a multiplayer game. Every client simulates the input of other clients (trying to anticipate their next move/step) until they receive the actual input from the relay server. A client sends its frame reports, packets containing the player input, to the relay server. And it

expects to get other players frame reports from the relay server on a regular basis.

- **Relay Server**: The relay server receives all frame reports from clients, identify the room they should be distributed to, and distributes them to all other clients in that room. It uses UDP for communication.

The purpose of the relay server is to relay packets between clients that share a room. Packets contain information about a client's movement in the game (frame reports), which all other clients use to correctly render the game state locally. For example, if there is a room of size 3, seen in figure 2.1, whenever a client sends their frame report to the server (red arrows in figure) it is forwarded to the other two clients connected to the same room (blue arrows in figure).

Through intuition, it is understandable that with a room size of n, the total number of packets sent from all clients to the server, for a single frame, would be n. Similarly, the total number of outgoing packets from the server per client, for a single frame, would be $n - 1$. Comparing these numbers, it becomes clear that the outgoing packets from the server greatly surpass the number of incoming packets. This relation between incoming and outgoing packets, and how it scales, is discussed further in section 3.1 as it greatly affects the network traffic generated by the relay server.

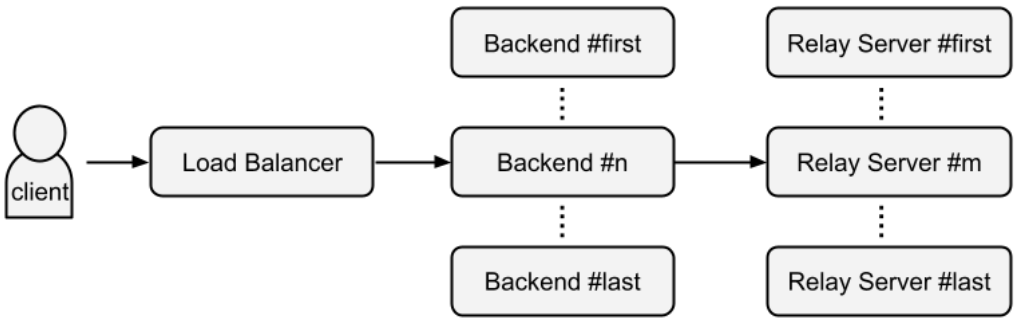


Figure 2.2: Flowchart showing how one client connects to a multiplayer relay server.

When a client wishes to create a new multiplayer game room, they are usually first directed to the load balancer and follow the flow in figure 2.2. The load balancer determines which backend-instance should service the request, and aims at creating an equal workload for all backend servers. The backend server verifies the request, that the client is real and that they have an account valid for the platform. It then asks one instance of the relay servers to create the new multiplayer room, or let the client join an existing one. After this, the backend creates a URL to join the game if the game and platform supports this. Lastly, the player is directed to the relay server and joins the room. Any other player that wants to join the same room are first verified by a backend instance, and then directed to the relay server instance that is hosting the multiplayer game room. This is how a large scale server structure would manage clients, but in this thesis clients go directly to one server.

2.2 Server Hosting

There exist many ways of hosting servers which are suited for different purposes. This thesis can not list all ways, but instead highlight some relevant to game server hosting.

Firstly, those that want smaller servers usually play with a group of friends on a smaller scale. Examples of companies offering this service are Survival Servers and Gameservers.com [7], [8]. A customer pays a monthly rate to get access to a server IP to which they and their friends can connect and play together. They can not control the way the server works, and usually there exist multiple kinds of servers tailored towards different games.

This is not suited for large-scale server hosting since costs will skyrocket quickly. It's also not possible to dynamically change the number of player slots per month, since they are charged per month. So if suddenly half the player-base disappears in the middle of a month, you are forced to still pay for the resources required to host them.

Another type of server hosting is self-server hosting, which requires that a person sets up their own server-computer and host the server on it. This requires that the user knows how to set up a server as well as maintain it. Several guides exist online to instruct on how to set up a server, one example is provided by Ubuntu [9]. Of course, the server capacity is limited by the hardware the user is able to purchase, so scaling the server is limited to the physical space and hardware available to the self-hosting user.

Virtual Private Servers (VPS) are virtual servers that take up a small space on a machine, which are best suited for smaller servers that do not require a full computer to operate. But at the same time, it gives the user full control of the server software setup. VPS can be self-hosted or use a VPS provider, e.g., Digital Ocean [10].

The last way of hosting servers is to use cloud providers, for example AWS, who provide a network of resources for different purposes [11]. Cloud computing can be used in many ways, and almost everyone comes in contact with it daily. Examples would be hosting email services or social media. If using cloud providers to set up email services for e.g., a company, this means that the company does not have to allocate money to hardware, storage space and maintenance of the email service [12].

Typically, there are three areas when talking about cloud computing: Software as a Service (SaaS), Platform as a Service (PaaS) or Infrastructure as a Service (IaaS) [11]. They all build on each other as seen in figure 2.3. SaaS is when a client can access a software using the internet (like email), instead of running it on their computer. Platform as a Service provides developers with the tools and resources necessary to produce a cloud-based SaaS software. IaaS would be the resources to host the SaaS and keep it running, and considers things such as network traffic capacity, storage, computing resources etc.

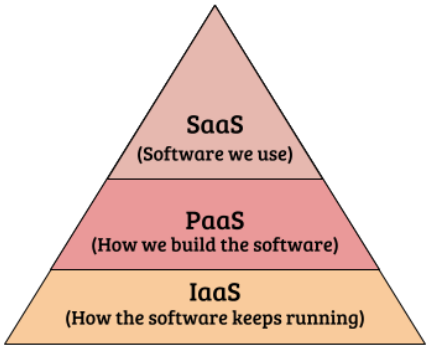


Figure 2.3: The pyramid formed by SaaS, PaaS, and IaaS.

2.3 Containerized applications

To containerize an application is a software deployment process in which the application, its code, files, and libraries are bundled together [13]. This way, the container can run on different devices using different operating systems. Containers can be considered similar to virtual machines, but the virtualization happens at different levels [14]. Virtual machines are an abstraction of the computing resources, while containers only virtualize the operating system. Essentially, this difference makes a container both smaller (in terms of memory) and faster than a virtual machine.

2.4 Services for Containerized Applications on the Cloud

The cloud infrastructure services market is a big industry that, according to Statista, spent $217 billion over 12 months (in Q3 of 2022) [15]. They continue to report that two thirds of the market revenue went to three companies: Amazon with AWS at 34%, Microsoft with Azure at 21%, and Google with Google Cloud Platform (GCP) at 11%. In total, these companies occupied 56% of the market in 2022.

AWS is the largest cloud platform provider for containerized applications that offers a range of services, and with many data-centers spread across many regions [16]. The main ideas of AWS, and other cloud providers, is to provide easy access to cloud computing resources and let users pay only for the resources they require. And since Amazon have data-centers in multiple regions, users can quickly deploy their products globally.

There exists different types of containers, but according to AWS, one usually used for game-servers is the Elastic Compute Cloud (EC2) [17]. The container type is available from GCP as well but called E2 [18]. There is no source telling that Azure provides a similar instance [19]. Instead, they seem to charge resources separately for each unit used.

Many features are associated with EC2 services. The simplest summary of them is the ability to set up virtual computing environments (often called instances or

containers), and the ability to chose from instances with different configurations for vCPU, memory and networking capacity [20]. This thesis mainly focuses on the T3 instance type provided by AWS since their resources are low, manageable, and free to use at the lowest level of resources; Using the T3 instances keeps the costs of the thesis low.

Table 2.1: Different AWS EC2 instances, their number of cores, memory, and cost per hour. The Network Bandwidth is 5Gbps for all machine types.

Machine type	vCPU	CPU Credits per Hour	Mem (GB)	On-Demand hourly rate ($)
T3.nano	2	6	0,5	0,0054
T3.micro	2	12	1	0,0108
T3.small	2	24	2	0,0216
T3.medium	2	24	4	0,0432
T3.large	2	36	8	0,0864
T3.xlarge	4	96	16	0,1728
T3.2xlarge	8	192	32	0,3456

To get an understanding of the payment-scheme for containerized application, the plan for AWS T3 instances is described (although there exist many instance options of EC2 containers). There are multiple ways of paying for AWS instances [21]. Two relevant ways are:

- On-Demand, which lets users pay per hour or second and only for the resources they need. Access is nonexclusive which might affect performance, and each vCPU is a thread of a real CPU. It is best used for applications with unpredictable workloads. Available configurations are listed in table 2.1.
- Dedicated Host, which gives the user a physical EC2 server with exclusive access to its resources. You can either pay per hour for a committed host instance, or for reserving a consistent amount of compute usage [22].

Prices differ depending on the region used, this thesis uses Stockholm as the region for all cost calculations and comparisons. The on-demand pricing for a dedicated T3 host costs $9.124 per hour ($78'831.36 per year), which allows a user to start up to 192 instances that in total consume 48 cores [22], available instance types for the T3 host can be seen in table 2.1. All instances have a Network Performance of 5Gbps [23], and the On-Demand pricing is for the Stockholm region [24].

CPU credits are earned by the hour, and each allows the instance to compute at the power of a physical CPU for one minute (burst computing). If one instead pays upfront for a year of reserving a T3 host it would cost $46'996. These are large sums of money, and why developers might consider paying On-Demand for single instances instead of dedicated hosts.

Table 2.2: Different GCP EC2 standard machines, their number of cores, memory, and cost per hour (Using the europe-north1 region).

Machine type	vCPU	Memory (GB)	Price/hour ($)	Outbound network traffic limit (Gbps)
e2-standard-2	2	8	0,02324	4
e2-standard-4	4	16	0,046479	8
e2-standard-8	8	32	0,092958	16
e2-standard-16	16	64	0,185917	16
e2-standard-32	32	128	0,371834	16

GCP provides services similar to AWS with the same E2 service [18]. These can be seen to a smaller extent in table 2.2 which covers the standard machine types, and more in depth in appendix A.1 which shows pricing for 3 categories of E2 containers. The first is the standard, the second figure is considered the *high CPU* versions, and the third is the *high memory* containers. When comparing these prices with those from AWS in table 2.1, it is evident that the price of the container is directly related to the amount of memory. When the memory doubles, the price also doubles. This is a simple relation that could be used for the scalability model. But contrary to AWS E2 containers, network traffic is priced separately per GB [25]. Although each container has a limit to the outbound traffic of a container, while inbound traffic is free of charge.

Pricing for Azure containers seems to be structured in another way, with no mention of elastic containers on their pricing page [19]. Instead, what seems to affect the price is three separate factors: the Operating System (OS) of the container, the memory allocation per GB, and the number of vCPUs. This would mean that a pricing model that is adapted to the set of container-types, as provided by AWS or GCP, would not work for Azure containers. But since this thesis aims to construct formulas for estimating the need of each metric (network traffic, memory allocation and CPU%) it should still be possible to estimate the cost of using Azure containers.

2.5 Previous Work on Scaling Containerized Applications

Many previous work focus on the dynamic scaling of cloud computing resources from a set of parameters. This i different from the approach in this thesis, which considers a static approach to provisioning; Resources are statically handed out based on a set of parameters that are not expected to change.

2.5.1 Scaling Containerized Applications

A key characteristic of containerized applications running on the cloud is their ability to dynamically scale their resources to fit the workload at different points in time [26]. By dynamically provisioning and de-provisioning resources, according to the current workload, users of containerized applications can pay for exactly the resources they need.

Multiple models have been created to achieve this, but none is targeted specifically towards a relay game server. Although, they do often consider a QoS. One example is Yu Sun et al. [27]. who aims at atomizing the allocation and de-allocation of resources while maintaining a web-applications QoS. They state that this tool is needed because it is difficult to determine which combination of servers are needed from looking at their hardware description, examples of this can be seen in table 2.1. And making an uneducated guess of what containers are needed could be costly for the user.

Regarding different QoS parameters, Said El Kafhali et al. [26] have gathered some QoE/QoS parameters used in other papers. Some of them are: Response time, rejection probability, CPU Utilization (CPU%), the average number of tasks and memory consumption. This shows that several researchers consider them important at that some of them are considered for this thesis, if they are related to relay server performance.

2.5.2 Resource Provisioning Techniques

A survey paper by Bhavani, B. H., and H. S. Guruprasad [28] tell that there are different ways to provision resources from cloud providers: Static, dynamic and user self-provisioning. With efficient provisioning, the profits from the hosted service can be maximized through minimizing the cost of resting resources. Although, the writers of the survey focus on other parameters that may affect resources, such as fault tolerance and reduced power consumption. The type of applications is also different, more geared towards scientific purposes. This could further solidify the lack of methods to provision resources for relay multiplayer game servers.

Another paper by Vlad Nae et al. [29] focus on resource provisioning in massively multiplayer online games. It is closer to the direction of this thesis, but not exactly the same. In this situation a server (or several) track the state of a gigantic ever-changing virtual world with, possibly, thousands of simultaneous players in the same room. But they do consider the same metrics for their model as this thesis will do: processing, memory and network. Although their models for the three metrics are more complex than they are expected to be for a relay server, due to the complexity of managing MMORPGs. However, they reach further in their scope, i.e., considering thousands of computers in a distributed network, and using multiple regions for their computing resources. And the complexity continues into discussions about, amongst many, dynamic predictions using neural networks. This reaches outside the scope of this thesis but gives guidance on possible future work.

2.6 QoE and Frame Rate

Frame delay is a problem in multiplayer games caused by clients locally rendering the game state, making them reliant on frame reports of all other clients in the game [30]. The problem is caused when data packages from other players are delayed, which causes an instant correction and can, for example, cause characters to jump on the screen which breaks an otherwise continuous animation.

If timing is important to the game, delayed frame reports can cause unfair disadvantages for some clients. This is a major problem considering that it affects the players QoE [2]. A proposed solution to this is to locally speculatively execute other players' inputs based on their previous actions [30]. When the data packages arrive, the simulation rewinds to the first incorrect frame.

It is individual to each human what frame rate is acceptable to them, but a study on video frame-rate has found that the lowest threshold for viewer satisfaction is 15 frames per second (fps) [31]. Meanwhile, a study on gaming and QoE dependent on frame-rate and bit-rate determines that the gaming community desire a frame rate of 60 fps, but that there was no significant difference in quality rating between 25 fps and 60 fps [32].

2.7 Length of Multiplayer Game Messages

This thesis aims to create a scalability model for the network traffic of relay servers (to be used as a sub-model to the final scalability-cost model, discussed more in section 3.1). One of the biggest contributors to network traffic is the size of the messages sent. Multiple papers and surveys have investigated and discussed the length of message-packets for different multiplayer games. S. Zander and G. Armitage created a traffic model for Halo 2, where they consider multiple players on the same client and how it affected client-server packet sizes [33]. They found that for 1 player per client, the client-server packet size was about 75 bytes. Meanwhile, T. Lang et al. [34] investigate Quake 3 and found that the map size affected the client-to-server packet size, but would usually be in the range 53–74 bytes. Finally, S. Ratti et al. [35] constructed a survey in 2005 that concluded that client-to-server packets are between 24 and 90 bytes long. From thees papers, it would be reasonable to assume that packet lengths withing the relay server system are between 24 and 90 bytes long.

3

Methods

This section covers the methodology of the thesis. Firstly it defines what metrics are considered important for the scalability-cost model, and reasoning on how they might affect the model, hence they are called sub-models. Secondly, the test-setup developed for the thesis is described to give insight on how it affects collection of data on the metrics. Possible faults and limitations of the test-setup will also be discussed.

3.1 The Scalability-Cost Model

The goals it for the scalability-cost model to consist of sub-models which produce requirement for different metrics. The thought is that in the future, many other sub-models can be constructed and used to fine-tune the scalability-cost model. This thesis starts with the metrics: Network traffic, CPU% and memory allocation. This is because these metrics are often displayed for different containers, as seen in section 2.4.

In the case of undeveloped sub-models, the requirement has to be given as a parameter to the scalability mode. A future goal would then be to replace each requirement-parameters with their own sub-model. So in the future, the scalability-cost model would calculate all requirements from a simple set of parameters. But this is the starting-point of the model, so many requirements have to stay as parameters. The one sub-model that exist is the network traffic scalability sub-model from section 4.1.

As seen in previous sections 2.6 and 2.1, the frequency at which players can render their game is important to their QoE, and the QoE is heavily reliant on that data-packets from the relay server are delivered on time. Thus, it is appropriate that the initial QoE parameter is set to be the frequency at which players want their game to update, and the size of the game rooms (since players want to play together, and the size greatly affect the number of packets the server has to send).

But packet size might vary between client-to-server and server-to-client packets, which has already been discussed in section 2.7. Therefore, the three variable inputs will be the total number of clients connected to the server, the size of rooms, an expected frame rate (QoE), and the size of client-to-server and server-to-client packet sizes.

Presumably, when a relation between the input parameters and the metrics, discussed

in the following section, is found the model will be able to determine the least number of containers needed by comparing the needs of each metric. The thought is that the model should, for each metric, firstly determine how many resources the metric needs and the cheapest combination of cloud containers necessary to sustain the resource demand. These steps will be called sub-models, where each sub-model defines a requirement for the scalability-cost model.

Then the scalability-cost model needs to work around the requirement that requires the most containers (equivalent to the highest cost) to satisfy the other requirements, and then continue in order of most demanding metric. But it is necessary to consider the differences between cloud providers, as exemplified in section 2.4. This section delves deeper on each sub-model and how to reason about them.

AWS was used during the thesis to measure metrics, and does provide a resource-usage monitoring tool called CloudWatch. It is an application and infrastructure monitoring tool that gathers metrics of used instances in real-time[36]. But it gives less control than manually investigating resource usage. Since this thesis is based on using Linux-containers from AWS, several useful Linux commands are discussed in the thesis starting with the following section.

3.1.1 Sub-Model: Network Traffic

Containers from some cloud-computing providers have limits on allowed volumes of network traffic per second, as discussed in 2.4. Or it is charged per used unit. Either way, it is vital to understand what amount of network traffic a set of parameters will cause in order to run enough containers to fulfill the network demand, and determine the cost of it.

It is important to note that it is not certain that the packets handled by the server will match the measured network traffic, since there might be activity in the background related to the cloud provider. For example, AWS uses their program CloudWatch to regularly gather data. But it is possible to calculate the bytes sent and received by the server and compare to the measured network traffic. To calculate the data received and sent by the server it is necessary to establish some parameters, which are:

- Packets from client, of size x
- Packets from server, of size y
- The total number of rooms r
- The size of a room n
- The expected frame rate (frequency of packets) f, derived from the QoE

Calculating the data handled by the server requires two separate parts. Firstly, if all clients send a packet to the server on a frequency, it would result in the bytes handled per second by the server described by equation 3.1.

$$r \cdot n \cdot x \cdot f \tag{3.1}$$

Secondly, if the server were to relay all those client packets on a frequency, it would generate data of the size seen in equation 3.2, assuming that every room is filled. Therefore, the total data handled is the two combined.

$$r \cdot n \cdot (n-1) \cdot y \cdot f \tag{3.2}$$

This traffic is generated on a frequency f, thus the total data received and sent per second to handle all connected clients would be equation 3.3.

$$(r \cdot n \cdot x + r \cdot n \cdot (n-1) \cdot y) \cdot f \tag{3.3}$$

The incoming and outgoing data of the server can be measured by itself through using counters and variables to count the number of packages, further discussed in section 3.2.1.

The sizes of network packets can vary due to several reasons. The first is that different games use different sizes, as seen in section 2.7. The second depends on the used communication protocol, UDP or TCP. The protocols operate in different ways; TCP is a streaming protocol, while UDP is a datagram protocol. They use headers of different sizes, and UDP has a packet size limit of 65,507 bytes for IPv4. In this case, it is assumed that clients and the server only send one packet per frame. Further, it is also assumed that UDP is the protocol used by the server. This results in packets being at least 8 bytes long, since this is the header size.

To actually measure the network traffic of the server running on a cloud instance, relevant tools are required. Firstly, the user has to connect to the server. This should be possible to do through ssh-clients for all cloud computing platform providers. But it is important to note that these ssh-clients contribute to the total network traffic of the server instance. And the tools to measure network traffic will differ depending on the OS used for the server.

How to measure network traffic depends on factors such as the used OS and how the connection to the server is set up. To measure the total network traffic (incoming and outgoing) on a Linux or Mac OS, the command `nload` is an option. It displays the incoming and outgoing data in bits per second (bit/s), as well as the average throughput and the total measured data. It is also possible to use the test-setup code to count, from the server's perspective, received and sent packages during an interval of time, which allows more control over how network traffic is measured and the ability to check the relation between packets in transit with measured network traffic.

Measuring the size of network packets in a program depends on the language used to develop the clients and server. In C++ the functions `recv()` and `send()` can be used by the server to get the length of received client packets, and the size of packets sent from the server. All UDP headers are included in the size, and thus the measurement should be correct. Equivalent solutions for other popular languages for creating servers should be available.

3.1.2 Sub-Model: Memory Allocation

The allowed memory per container instance typically varies more than the allowed network traffic, at least for AWS T3 instances, as seen in section 2.4. Assumptions can be made from section 2.1 that the relay server is not memory-heavy. But this could differ between different relay servers. Some might gather loads of data for various reasons, while some might just keep track of which clients reside in the same room. Either way, there is a need to first find the memory allocated to the idle server, if other memory is reserved, and then how much memory is used per client and room.

There is a minimum memory allocation required for the basic operations of a relay server, which are: The memory required to store the server code, the memory required per connected client and the memory required per active room. The size of the server has to be measured to understand its size. But the two other parts depend on the datatype used for storing them.

A growing memory requirement would also assume that the server dynamically allocates space for clients and rooms using, e.g., a C++ `std::vector` object. But the memory could also be statically allocated if using a predetermined C++ `std::array`. So the requirement produced by the memory sub-model will consists of two parts: The memory required for the server, which is fixed, and the memory required to track connected clients and their rooms, which is fixed or dynamic. And it is assumed that the storage for clients and rooms is similar to a list structure, because this gives us an easy way of looking at the scalability of the required memory for clients and rooms.

The commands that can be used to investigate what amount of memory is allocated by a process vary by OS. For Linux, the command `pmap` works when given the Process Identification (PID) of a process [37]. The PID can be obtained using the command `pidof` given the name of a process [38].

The parts that contribute to the cost (read memory allocation) of a generic list structure for a list called *Rooms* is visualized in figure 3.1. Firstly, there is a cost associated with the creation and head of the list-structure. Secondly, there is a cost to the structure of each room (which in themselves act like smaller list-structures since they store a set of information related to clients). Each room at least track the IP of its clients in order to relay packets to them.

$$rooms_{mem}(c, o, r, n) = n \cdot o + (n \cdot r \cdot c) \tag{3.4}$$

With this structure, constructing a formula for the memory needed to store rooms becomes relatively easy. It is assumed that every spot available to store a room is used. Then it is possible to establish some variables describing equation 3.4:

- The size of a client structure: c,
- the size of a room with zero clients (but an empty structure to store them in: o,

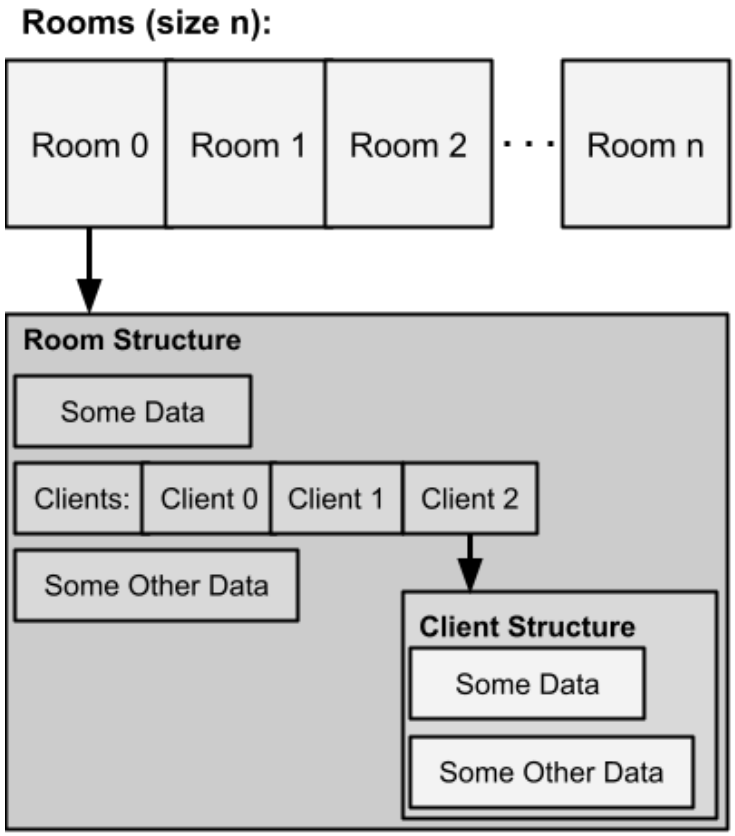


Figure 3.1: Showing a generic structure of what memory is required to store information about clients and rooms.

- the number of clients per room (room size): r,
- the number of rooms stored in the server: n,
- and the function that returns the size required for a set of rooms: $rooms_{mem}(c, o, r, n)$.

3.1.3 Sub-Model: CPU Utilization

There could be other processes running on the cloud-instance than the server. In order to correctly determine the needed CPU capacity. Firstly, there is a need to consider what percentage of CPU% every client requires (with consideration to room size or other variables) as well as what is already used.

CPU% will vary from server to server due to their implementations and client demand of service. Thus, this thesis aims to present a methodology of predicting the future CPU% for a server setup. To do this, the server has to be run on a bare-metal machine using physical CPUs, not vCPUs. This is because early testing showed inconsistent results for CPU% measurements on vCPUs for the same test-case, and were therefor deemed unreliable. This may be because each vCPU is a thread of an actual CPU.

Measuring the CPU% can be done in many ways. This thesis used the command

`top` with the flag `-l` which allows the user to capture the state of the machine for a set number of intervals. This command works for both Linux and macOS machines. The results were written to a text file by writing `$ top -l 15 > results.txt`. And the CPU% data can be gathered for a certain program by using the command `$ cat results.txt` grep < program_name >|.

Test cases were grouped by their QoE, testing 10Hz, 20Hz, 30Hz and 40Hz. For each QoE, room-sizes of three to seven were used for a number of rooms from the pool of {2, 4, 6, 8, 10}. The hope was to from this data find a formula that both describes how CPU% scales with increased number of rooms, changing room sizes and changing QoE. Alas, this was not possible and discussed further in 4.3. This is when ChatGPT was consulted on how to find a relation between the data points. ChatGPT suggested creating polynomial regression models on each test group (based on QoE), this is what the result in section 4.3 rely on.

There is a limitation to this method, the values for untested QoE-groups can not be predicted, since the models are based on values for a certain QoE. This would require a user to determine a QoE at which their server should operate, and from that they can gain knowledge on how CPU% scales with room-size and the number of rooms.

The server was set up on a MacOS-laptop and clients spawned on a Windows machine in Windows Subsystem for Linux WSL. Both machines operated on the same network and could thus easily communicate.

3.2 Test-Setup Development

The first part of the thesis was to develop a test-setup, consisting of a client-simulation program and an already created relay server written in C++20. Each client-simulation program can spawn multiple clients, and multiple instances of the program can be run. Each instance can connect to the same server. The development of the client-simulation and changes to the server are described further in this section.

Since the test-setup was used to collect data for the scalability-cost model, any possible faults with the model could stem from the test-setup. So the goal of this section is to describe how to create a client-simulation program, how to use it for testing various metrics of a relay server and start discussing faults that might occur from the implementation. The described setup has been used in this thesis and hopes to inspire other version should the model be expanded upon.

3.2.1 Client-Simulation Program

The goal of this program was to spawn multiple clients that connect to a given IP and port, then repeatedly send packets while also receiving them. This was needed in order to measure if all clients kept receiving packets from the server at the expected QoE, since this rate had to be continuously upheld for any test to be valid.

The client-simulation program consists of three different types of threads. The first one is the *main* thread, it spawns clients according to the program arguments. Each

client consists of two threads, a *client-sender* thread that sends datagrams while the other, the *client-receiver* receives datagrams. The client threads are described further in the next section.

The different threads use signals and future-objects to communicate in the following order: *main* to *client-sending*, *client-sending* to *client-receiving*. Thus, for example, *the client-sending* thread can tell the *client-receiving* thread to stop its execution once the client stops sending packets. Then the main thread join all *client-receiving* threads before exiting the program.

3.2.1.1 The Structure of One Client

One client has 4 tasks to perform: get access to a room on the server, repeatedly send packets to it, receive packets from the server and track that the given QoE is maintained. To do this, each client spawns two different threads. The first thread is the thread spawned by the main process, the *client-sender* thread. The second is spawned by the *client-sender* thread and called the *client-receiver* thread.

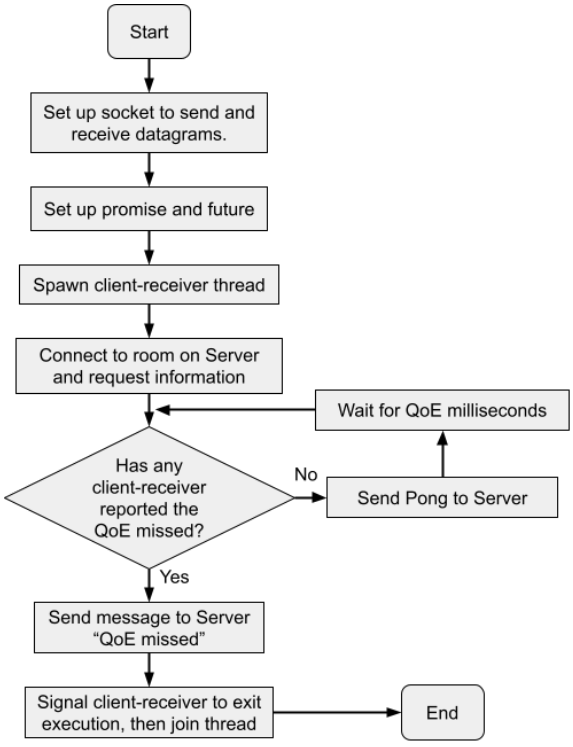


Figure 3.2: Flowchart of the client-sender thread behavior.

The *client-sender* thread structure can be seen in figure 3.2. It is responsible for connecting to a room on the server. After this, it repeatedly sends packets to the server on a frequency given by the QoE parameter. Its execution can be stopped by a signal from any *client-receiver* thread, which tells the *client-sender* that the QoE is not meet. Thereafter, the *client-sender* signals the *client-receiver* to exit its execution, then waits for the *client-receiver* thread to join, then finally ends its own execution so that the main program can join all *client-sender* threads.

The *client-receiver* thread structure can be seen in figure B.1. It firstly determines how many packets should be received per time frame (determined by the QoE), which is the room size minus one (further referred to as packet-group). The thread receives packets from the server and tracks the time between received packet-groups, and it tracks several pieces of data: The difference in time between the previously and currently received packet-group, a list of the 10 latest such times and the number of times the QoE parameter has been missed. With this data, the *client-receiver* can determine if the simulation should be shut down or not.

The *client-receiver* operates in a loop that starts by checking if its *client-sender* has told it to shut down. Then it checks if any packets have arrived from the server. If not, the loop continues from the first step. Otherwise, a counter is incremented. Once it reaches the size of the packet-group, the clock is stopped and the counter is reset. The time since the last received packet-group is calculated and stored in the list. If the QoE is not meet, it is logged to a file for the user to examine. Otherwise, the average of the list is calculated and compared to the expected QoE. If the average is the largest value of the two, the simulation will stop. After these checks, and if the simulation should continue, the clock will start again to measure the time until the next received packet-group.

3.2.1.2 Spawning Many Clients

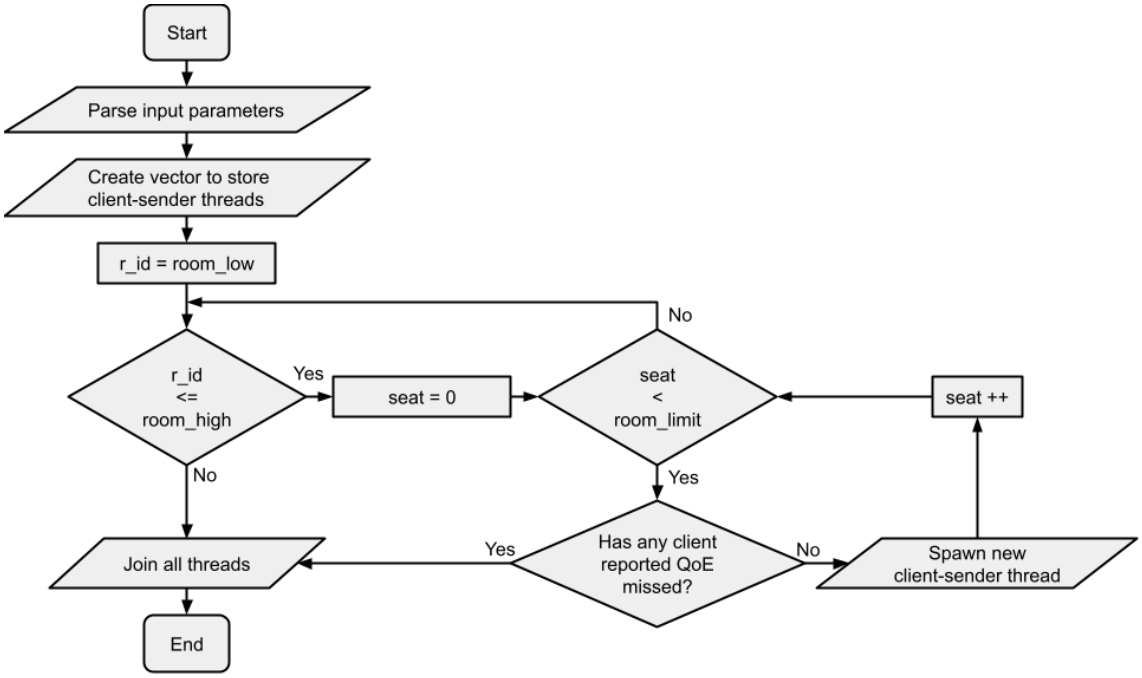


Figure 3.3: Flowchart of the client-simulation main thread.

The *main* thread of the client-simulation program follows the flow in figure 3.3. The number of clients spawned by it depend on the input arguments of the program. It will look to the range of available room-IDs, and how many clients fit per room (which assumes that every room is of equal size). The *main* thread loops over the number of rooms and create as many clients as available spots. And each *client-sender* thread

is pushed back into a vector, so that the *main* thread can track all active clients. On program shutdown, the *main* thread joins all clients in the vector.

The *main* thread is also used to time the execution of the simulation. Once all clients have been created, it sends a packet to the server which tells it to start collecting data such as received and sent packets. This is used to investigate the network traffic in 3.1.1. Once a set time span has passed, the *main* thread sends another message to the server that the simulation should end, and signal all *client-sender* threads to exit their execution.

3.2.2 The Relay Server

At the start of the thesis, there existed a simple single-threaded server developed by Opera. It is a relay server described partially in section 2.1. This was used as the basis of the server used in the thesis. Some changes were made to use it explicitly for testing:

- The garbage collector was turned off. Its purpose was to reset rooms that were no longer used. But clients remain connected for the whole simulation, and thus it is redundant to create this thread.
- The main loop of the server, receiving and managing incoming datagrams, was split out of the main program. This way, it could be passed as a function to new threads, enabling the server to be multithreaded in its main function.
- A command-line argument has been added to determine how many threads should run in the server. This is to easily tailor to different instances with different cores, seen in table 2.1.
- A new packet type was added, when received the server will send the same packet to all connected clients. This is what ends the whole simulation, and makes all clients shut down.
- Different counters were added to keep track of how many packets have been sent and received by the server, and the total size of these packets.

3.2.3 Issues with the Test Setup

Issues with the test setup are provided below for complete transparency, as these faults may affect the measurements done with it.

- When many clients run on the same machine, they share computational resources and memory. At some point, there could be so many clients spawned that they do not get access to the resources in time of the QoE, and thus determine that it has been missed. To determine if this is the case for a set of parameters that cause a QoE miss, run the parameters with lower number of clients on two separate machines, which connect to a server on a third machine, and see if the simulation exits at the same number of clients.

- With many clients running on their own thread, there will be many context switches to change between them. This could also affect the perceived latency in receiving packets.
- Creating a thread takes time, which could cause other threads to miss their QoE. To somewhat avoid this from happening, clients spawn at an interval of one second to spread out their creation.

A bigger problem for the whole setup is correctly timing a testing-period. The main thread signals the server once all clients have to be created, and the server has to receive and read this message before it starts collecting data. The client receiver sends another packets once the testing window has passed and the server will shut down. On shutdown, there might be incoming packets that have not been read or outgoing that have not been sent, which affects the readings of the simulation. This was investigated further in section 4.1

3.2.4 Test-Environments

The thesis used three different environments. The first one, a Mac M1 machine, was used to host both clients and the server to develop the test-setup and ensure that it worked as expected. No usable data was gathered, since it is, for example, improbable for clients and a server to share computational resources. And sharing resources will worsen the performance of the server.

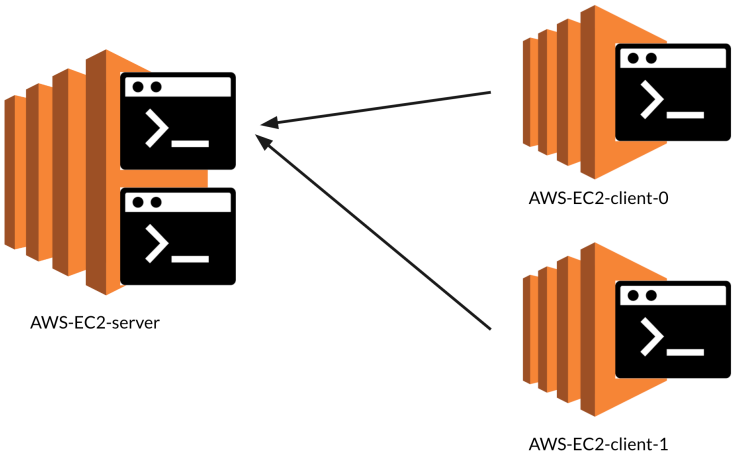


Figure 3.4: Showing 3 AWS T3.micro instances, their purpose, and number of connected ssh clients.

That is why the second environment used a number of AWS instances, as displayed in figure 3.4. AWS instances can run using several OSs. Ubuntu was chosen for this thesis since the provided server-code already worked well on this OS. The server was isolated on one of the possible instances listed in section 2.5.1, while clients were simulated from others. By placing the instances in the same AWS security group, the instances were able to communicate with each other using private IP addresses. In this environment, data for the network traffic and memory sub-models

was gathered. Initially it was used to test CPU% as well, but the results were found to be inconsistent.

The third environment, used to gather data on CPU%, used a MacOS-laptop (using Intel processors) and a Windows-machine running WSL. It was deemed necessary to run the server on a bare-metal machine and use actual CPUs instead of vCPUs, because early testing showed that the CPU% for vCPUs were inconsistent for the same test-cases. So a MacOS-laptop ran the server while the Windows machine produced clients using WSL, since the test-program only compiled for Linux and Mac. Ideally the clients would have been able to run on AWS instances, since it would have allowed more simulated clients, but it was not possible to get UDP packets to be delivered from AWS to the MacOS-laptop.

4

Results

This chapter focuses on the gathering and analyzation of data for the previously mentioned metrics: Network traffic, CPU% and memory allocation. The goal was to, from the gathered data, find in what way the metrics scale with changing parameters. This enables the creation of sub-models, which will produce requirements for the scalability-cost model.

4.1 Network Traffic Sub-Model

To gather data, two T3.micro machines were used. One ran the server and the other the client-simulation program. Communication with the machines was established using ssh-clients, of which the server needed two: one to run the server program, and one to use the Linux command `nload` to view network traffic. Since there were shh-connection to the server, the network traffic was presumably affected. Thus, the analyzing of network traffic started with figuring out how much traffic was caused by the two ssh-clients.

Table 4.1: Incoming and outgoing network traffic of idle server, observed over 24 minutes using *nload*.

Incoming network traffic		Outgoing network traffic	
Avg:	1,06 kbit/s	Avg:	10,16 kbit/s
Min:	816,00 Bit/s	Min:	5,73 kbit/s
Max:	7,6 kbit/s	Max:	25,55 kbit/s
Total:	55,63 Mbyte	Total:	3,12 Mbyte

The command `nload` was used in one ssh-client and left running for 24 minutes. The results displayed in table 4.1 show that the previous assumption was true: There exists network traffic without a running server on an AWS instance. During this time it was observed that there were always incoming and outgoing data, that it fluctuated greatly between the min and max values, and that the average stabilized. The total shown in the table considers traffic from before the command was run and can thus not be used to make assumptions, and it was impossible to manually start all clients, the server, and the monitoring of network traffic at the same time,

thus this metric (total network traffic, for incoming and outgoing traffic) were not considered going forward.

Table 4.2: Comparison measured and calculated bytes going to and from the server, for 30 clients in rooms of 3 and different QoE. Showing the difference compared to the calculated values.

QoE	Calculated bytes in	Calculated bytes out	Measured bytes in	Measured bytes out	Difference in (Byte)	Difference out (Byte)
1	7200	14400	7285	14000	+85 1,18%	-400 2,78%
10	72000	144000	71204	142272	-796 1,11%	+2272 1,58%
30	216000	432000	210514	420960	-5486 2,54%	-11040 2,56%
60	432000	864000	417702	835200	-14298 3,31%	-28800 3.33%

Several tests were run for different purposes. Firstly, the proposed equation 3.3, for calculating the total bytes sent and received by the server per second, was tested. Data is presented in table 4.2 where 30 clients in rooms of 3 started with different QoE frequencies and packages of 24 bytes, and the simulation ran for about 10 seconds. The time could not be exact due to the issues described in 3.2.3 The table shows that the calculated expected value and the measured values differ by a small percentage.

Firstly, the expectations from equations 3.1, 3.2 and 3.3 were compared to the measured values. Looking at the results for QoE 60, there were 14298 less received bytes than the expected 432000 bytes. That is equal to 595.75 packets missing for a packet with a size of 24 bytes. With 30 clients, that would be 19.86 packets missing per client. At a frequency of 60Hz, 20 packets missing is equal to a simulation time of 9,7 seconds. So if the simulation ended 0.3 seconds earlier than expected.

Next, a similar comparison was made for outgoing packets for the QoE 60 case. 28800 bytes equals 1200 packets. This means that each client received 40 packets less than expected during 10 seconds. 40 Packets received by a client is equivalent to the server receiving 20 (at room size 3, 2 packets are sent by the server for every received packet), which leads to a 0,3 second time difference if packets are sent at a frequency of 30Hz, just like the difference of incoming packets.

The result of applying the previous logic to all cases in table 4.2 is the time differences shown in table 4.3. It shows that for a set of 30 clients in rooms of 3 with changing frequency, the time difference that could be the cause of too many or too few packets being sent or received is less than half a second. This half second could be caused by the issues with the test-setup described in section 3.2, since thread switching,

Table 4.3: Time difference for various frequencies causing to many or to few packets to be sent or received.

QoE	Difference in (Byte)	Time equivalent	Difference out (Byte)	Time equivalent
1	+85	0,12s	-400	0,19s
10	-796	0,11s	+2272	0,16s
30	-5486	0,25s	-11040	0,26s
60	-1429	0,3s	-28800	0,3s

packet sending and receiving etc. can cause counters to not immediately take effect, processes to immediately stop etc. Some time-skew is expected unless the test setup is improved and its threads controlled on a lower level. It could be believed that these time-skews are within reason and that the results in table 4.2 show that the equation 3.3 holds for changing frequencies.

Table 4.4: Comparison measured and calculated bytes going to and from the server, for 5 rooms at 30Hz with varying room sizes. Showing the difference compared to the calculated values.

Room size	Calculated Bytes in	Calculated Bytes out	Measured Bytes in	Measured Bytes out	Difference in	Difference out
3	108000	216000	105154	210240	-2846 2,54%	-5760 2,67%
4	144000	432000	140211	420480	-3789 2,63%	-11520 2,67%
5	180000	720000	175217	700800	-4783 2,66%	-19200 2,67%
6	216000	1080000	210541	1052280	-5459 2,53%	-27720 2,57%

To show that equation 3.3 holds for a changing number of clients and room size, but a constant QoE and packet size with allowances in time difference from table 4.3, two tables have been constructed. Table 4.4 shows the results for varying room sizes, with a fixed set of 5 rooms and QoE of 30Hz. Table 4.5 shows the respective time differences which can be deduced from the differences in packets received or sent by the server. The times in table 4.5 are about the same as the corresponding QoE row in table 4.3, which suggest that these results are to be trusted on the same basis.

With a correlation between the parameters and the number of bytes received and sent by the server, it was time to investigate how network traffic (in bits/s) scales with different parameters. But the previously discussed timing errors had to be

Table 4.5: Time difference for various room sizes causing to many or to few packets to be sent or received.

Room size	Difference in	Time equivalent	Difference out	Time equivalent
3	-2846	0,26s	-5760	0,27s
4	-3789	0,26s	-11520	0,27s
5	-4783	0,27s	-19200	0,27s
6	-5459	0,25s	-27720	0,26s

considered. So the remaining test were performed as follows: The simulator was set to run for 10 seconds. Then `nload` was used for 5 seconds within the simulation time. The average network traffic measured by `nload` was used as the result for the upcoming test.

Several measurements based on varying parameter values, except a constant QoE of 30hz, are shown in appendix C.1, where traffic from client-to-server (in) and server-to-client (out) was measured. The table has some inconsistencies. Most notably, the count of packets for 30 clients. The simulation caused about 880 incoming packets per second, while this should be $30 \cdot 30 = 900$ incoming packets. This could be because of the earlier discussed timing issues with the simulation. If the simulation ends too early, the average incoming packets per second can be lower than expected.

So to gather any usable data, the total number of packets per second had to be calculated using the equation in section 3.1.1, and subtracting the network traffic of the idle instance in table 4.1. Doing this renders the appendix C.2 which values were used from this point onward. In the table, there is no distinction between client-to-server or server-to-client packets or network traffic, the number shown are the total packets, bytes, and network traffic.

Table 4.6: Displaying what packet sizes were compared, their respective network traffic and the found increase of network traffic per byte added to a packet size.

Packet size (byte)	Packet size (byte)	Traffic (bit/s) per Packet	Traffic (bit/s) Packet	Network traffic (bit) per byte
24	48	474,9	651,01	7,34
24	64	474,9	771,78	7,42
24	80	474,9	890,8	7,43
48	64	651,01	771,78	7,55
48	80	651,01	890,8	7,49
64	80	771,78	890,8	7,44

From appendix C.2 it could be believed that a linear relation exists between the network traffic required per packet and the packet's size, since the traffic per packet remains consistent for packet sizes. The average traffic (bits) per packet size (bytes) was calculated for each packet size with the data from appendix C.2, and compiled in table 4.6. Different packet sizes and their network traffic were compared, and the traffic per byte of a packet was calculated. The average network traffic per byte of a packet size was 7,45 bits. The lowest packet size used has been 24 bytes, and must thus be the baseline case for the network traffic sub-model with a minimum network traffic of 474,9 bits per packet.

$$(474.9 + 7.45 \cdot (x - 24)) \cdot \Big(r \cdot n^2 \cdot f\Big) + 11660 \tag{4.1}$$

$$\begin{aligned} &(474.9 + 7.45 \cdot (x - 24)) \cdot (r \cdot n \cdot f) + \\ &(474.9 + 7.45 \cdot (y - 24)) \cdot (r \cdot n \cdot (n - 1) \cdot f) + 11660 \end{aligned} \tag{4.2}$$

The final equation 4.1 for calculating the expected network traffic would be based on the previous reasoning, parameters defined in 3.1.1, and the following: Firstly, it would be reasonable to assume that the client-to-server packets are of the same size as the server-to-client packets, since the server only relays client packets. Secondly, it was necessary to remember the network traffic caused by the idle server seen in table 4.1, since it was included in the measured traffic.

With this, a function could be constructed. It simply had to calculate how many packets will be sent based on the number of connected clients, size of rooms, and expected frequency of packets. This needed to be multiplied by the packet size, determined by the linear equation described before, and then lastly the idle network traffic was added. If one would like to use different packet sizes for client-to-server and server-to-client packets, equation 4.2 could be used where x and y represent different packet sizes.

A slight error with equation 4.1 and 4.2 is that the idle traffic will not be the same without connected ssh-client. And there might be traffic caused by AWS, since they for example, gather analytical data. So for the sake of comparison the idle traffic has to be included, but will change to an unknown number once the ssh-clients disconnect.

Appendix C.3 Shows the measured traffic from appendix C.1 compared to the calculated network traffic using equation 4.1. It is clear that the error ranges between 1,5% and 0,03%, with an average error of 0,445%. With this low average error, it can be concluded that the equation 4.1 accurately depicts the data it is based on.

The equation 4.1 fits well to the data-points used to create it, but it also has to be compared to unused data-points. Two previously unused sizes of packets, 40 and 96 bytes, were tested in four cases, each with different number of clients and room sizes. The network traffic present without a running server was once again checked, and returned different values than previously measured in table 4.1, at a total average of

Table 4.7: Compare measured network traffic of previously unused data-points to the calculated network traffic.

Total Clients	Room Size	Packet Size (byte)	Measured Traffic (kbit/s)	Calculated Traffic (kbit/s)	Error (%)
15	3	40	655,46	641,636	-2,109
60	3	40	2559,17	2566,520	0,287
70	7	40	7134,39	6986,624	-2,071
140	7	40	14230	13973,240	-1,804
15	3	96	1122,04	1092,212	-2,658
60	3	96	4350	4368,824	0,433
70	7	96	12070	11892,896	-1,467
140	7	96	24180	23785,784	-1,63

8,034kbit/s. This was used in the previous established equation 4.1 instead of the old value (11660 bits) and compared in table 4.7. The results are promising with an error percentage between 0,287% and 2,658%, and an average error of 1.38%. With this it is possible to assume that the equations can be used for the scalability-cost model to predict the required network traffic for a set of clients, a room size, packet size and a frequency at which client expect to receive packets.

4.2 Memory Allocation Sub-Model

The results in this section have been found through measuring the required memory allocation for a server with different starting parameters, and later with connected clients. It also assumes that a structure for storing rooms called *rooms* is used.

Firstly, the memory required to run the server with no memory allocated for rooms was measured. An unordered map *rooms* was created without any entries, and the memory allocation of the server was measured to 6100kbit. This was done using two commands in succession: `pidof` and then `pmap`, who retrieve the PID of a command and the memory used by it respectively. One flaw is that the latter only shows memory allocated in kbit, which could lead to some rounding errors. And it was confirmed that the size of rooms did not change the initially allocated memory, since 1000 rooms were started with 1, 5 and 10 clients and every case resulted in a memory requirement of 6496kbit.

Then several cases shown in 4.8 were started, their memory measured, and the memory required per room was calculated in two ways. The last column shows that the increase in memory allocation per room converges on a value around 0,4kbit per room. The arithmetic mean of the last column is 0,423, the geometric mean is 0.421

Table 4.8: The memory required to start the relay server with different amount of rooms.

Number of rooms	Allocated Memory (kbit)	Increase (kbit)	$\frac{\text{Increase}}{500}$	Memory (kbit) per room, compared to first row
0	6100	-	0	-
500	6364	264	0,528	0,528
1000	6496	132	0,264	0,396
1500	6760	264	0,528	0,44
2000	6892	132	0,264	0,396
2500	7188	296	0,592	0,435
3000	7188	0	0	0,363
3500	7584	396	0,792	0,424
4000	7716	132	0,264	0,404
4500	7980	264	0,528	0,418
5000	8244	264	0,528	0,429

and the harmonic mean is 0,42. With this it is concluded that the variable o in the assumed equation 3.4 is equal to 0,42kbit.

Table 4.9: Show the measuring error caused by the command *pmap*, where a different number of rooms require the same amount of memory.

Number of rooms	Allocated Memory (kbit)
0	6100
125	6232
200	6232
400	6232
500	6364

There is a peculiar occurrence in table 4.8, when measuring memory allocation for 2500 and 3000 rooms they require the same amount of memory. This was assumed to be a fault of the commands used for two reasons, one is that number are rounded to the closes kbit, the other is that during testing it was difficult to achieve results giving small changes which can be seen in table 4.9: Multiple number of rooms

were started but for example 125, 200 and 400 rooms required the same amount of memory (6232kbit).

Table 4.10: Investigate increase of memory allocation requirement with increased number of connected clients, using a server with space for 500 rooms and five clients per room.

#Rooms	Total Clients	Required Memory Allocation (kbit)	Memory Without Base Case (kbit)
0	0	6364	0
10	50	6364	0
15	75	6496	132
20	100	6496	132
25	125	6496	132
30	150	6628	264
35	175	6628	264
40	200	6628	264
45	225	6760	396

Next, the memory required per client was investigated by connecting a different number of rooms (of five clients each) to a server with space for 500 rooms, with results shown in table 4.10. Here the previous problem with the `pmap` command reappears: The same memory allocation requirement was displayed for multiple inputs, with sudden gaps in values. A pattern is visible in the table: For every 75 connected clients (rows highlighted with gray), the memory allocation requirement increased by 132kbit. If only the gray rows are considered and the memory (without the base value of 6364kbit) is divided by the number of connected clients, the value 1,76kbit consistently returned as the memory required per connected client. Hence, it is theorized that the variable c in equation 3.4 is equal to 1,76kbit for a relay server.

It was necessary to be determined if the variable c was independent of the size of rooms. Therefore, the previous experiment was performed on another room size. But this test was limited by the test setup, because ideally a much larger room size would have been tested, but that would cause each client-simulator to be able to support much fewer clients, and thus many more client simulators (and AWS containers) would have been required. So the room size was set to three, a lower room size than previously tested but still a factor of 75, which would allow more clients to run per client simulator.

The test for 3 rooms, also on a server with room for 500 rooms, are displayed in table 4.11 which shows that the memory allocation requirement mostly increased at

Table 4.11: Investigate increase of memory allocation requirement with increased number of connected clients, three clients per room.

#Rooms	Total Clients	Required Memory Allocation	Memory Without Base Case
0	0	6364	0
10	30	6364	0
15	45	6364	0
20	60	6364	0
25	75	6364	0
30	90	6496	132
35	105	6496	132
40	120	6496	132
45	135	6496	132
50	150	6628	264
55	165	6628	264
60	180	6628	264
65	195	6628	264
70	210	6628	264
75	225	6760	396

the same number of clients as in the previous table 4.10. The difference between the two tables is that for 3 clients per room, the memory requirements increased from 6364kbit to 6496kbit at 90 connected clients instead of 75. But with this pattern it is possible to assume that almost any formula will have errors since the memory requirement only changes at certain values of connected clients.

$$server_{mem}(n, r) = 6100 + n_{all} \cdot 0.42 + n_{full} \cdot r \cdot 1.76 \tag{4.3}$$

With the variable values for equation 3.4 defined the equation 4.3 could be constructed, which returns the required memory in kbit, where n_{all} signifies the number of rooms available, n_{full} signifies rooms used by clients, r the size of rooms, 6100 is the memory (kbit) required for a server with no rooms, 0.42 is the memory (kbit) required per room with no clients and 1.76 the memory (kbit) required per client in a room. But this equation had to be tested to unveil its accuracy.

Firstly, equation 4.3 was tested on randomly chosen cases to see if the equation could

Table 4.12: Testing the constructed memory allocation sub-model.

n_{all}	r	n_{full}	Memory (kbit, measured)	Memory (kbit, calculated)	Error (%)
4250	8	12	7948	8053,96	$1,33$
5000	2	33	8080	8316,16	$2,92$
6000	12	10	8904	8831,2	$-0,82$
7333	5	24	9136	9391,06	$2,79$
200	4	44	6496	6493,76	$-0,034$

accurately calculate the memory required for different amounts of rooms with no clients connected. The tests are shown in table 4.12, their measured and calculated memory requirements were compared and followed by the error given as a percentage. The average of the errors is 1,58% and the largest error is 2,79%. The larger error is, as discussed previously, probably derivative of how memory is not continuously allocated, instead it seems to be added in chunks which naturally causes an error with a continuous model.

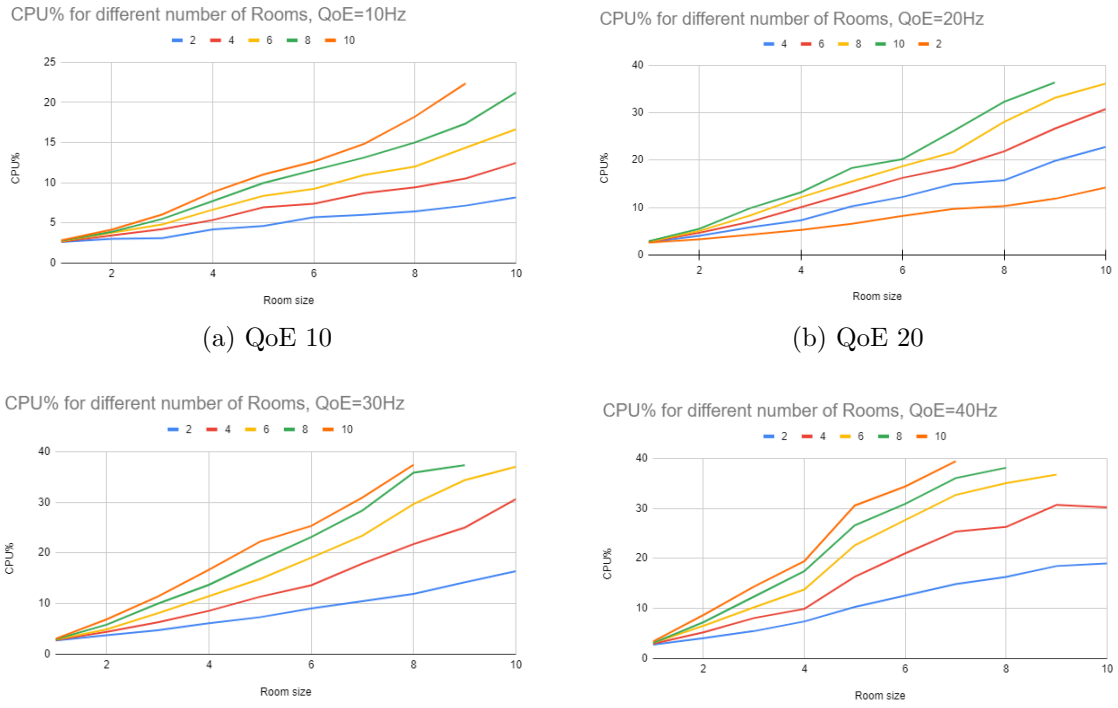
4.3 CPU Utilization Sub-Model

Firstly, the CPU% for several cases, each with different QoE, room sizes, and number of rooms, was measured as described in section 3.1.3. The resulting CPU% measurements were the average of 12 samples taken over 15 seconds. Originally, 15 samples were taken, but three were removed for different reasons: The first value was always 0%. And the following sample would often be the lowest value, thus the lowest value of the 14 remaining samples was removed. To negate that, the highest value was also removed from the pool of samples.

When observing the values in appendix D.1 there was no apparent relation in how they scaled. For each test-group, a graph was constructed using excel, seen in figure 4.1. When observing the graphs, there seemed to exist a continuous relation between CPU% and the number of rooms for different QoE test-groups.

With this assumption, several regression models were created: A linear, 2nd degree polynomial, and 3rd degree polynomial. An example of the code can be seen in appendix D.2, which was partially generated by ChatGPT. It was consulted on what to do with the data in appendix D.1 and suggested using regression models for each QoE. Then the acquired data was split into training and test sets, where for each QoE, the data for four rooms was left for testing (marked in gray in appendix D.1) and the other used to train the model (marked in white in appendix D.1).

The R-Squared value (r2) values of the regression models, indicating how close to



(a) QoE 10

(b) QoE 20

(c) QoE 30

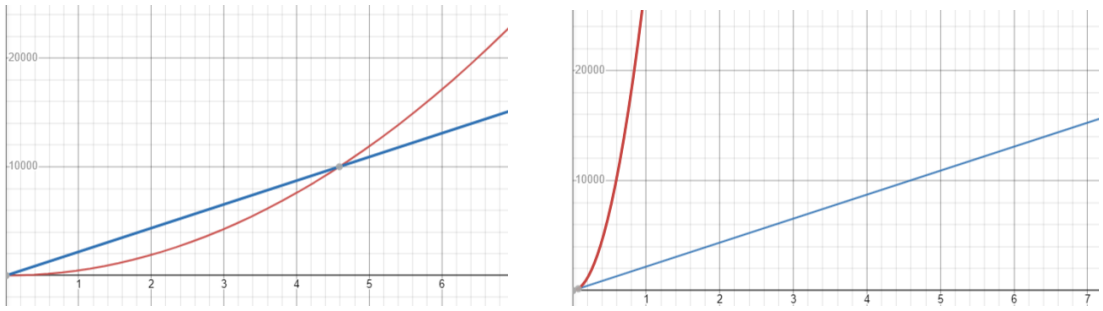
(d) QoE 40

Figure 4.1: Graphs showing how CPU% increases with the size of rooms. Each graphs shows results for different QoE, while each lineRepresent different number of rooms.

Table 4.13: Comparing the R-Squared value value of the 3rd degree polynomial model on different sets of data.

	R-Squared value	
QoE	Training Set	Test Set
10	0,995	0,992
20	0,995	0,939
30	0,996	0,990
40	0,993	0,985

the provided data the model fits, can be seen in appendix D.3. From the table, it is clear that the linear model fits the data the worst, and the third degree polynomial model fits the best with and average r2 of $0,992$. Thus, the third degree polynomial model was used to predict the CPU% of the test set. The r2 value for each QoE is displayed in table 4.13. It shows that generally the third degree polynomial model can predict the data withing the training set range.



(a) QoE 1 (b) QoE 60

Figure 4.2: Graphs displaying how memory requirements (blue line), and network traffic requirements (red line) for a packet size of 24 and different QoE, scale with changing number of rooms (while using a room size of 1).

4.4 Compare Network and Memory Sub-Models

While the CPU% sub-model does not have a formula to examine, the sub-models for network traffic and memory allocation do. In their respective section, different variables have been used, but it is possible to describe them with more similar parameters. network traffic is described as equation 4.1 while memory allocation uses equation 4.3. In the first equation, let us use a variable b for the unknown traffic not caused by the server. In the second, it is possible to assume that all rooms on the server will be full and thus use the same n for n_{all} and n_{full}. Lastly, it is important to remember that the memory allocation sub-model returns a value of kbit, while the network traffic sub-model returns a number of bits.

With the prior reasoning, we have two equations that are easier to compare. For network traffic, we get equation 4.4:

$$(474.9 + 7.45 \cdot (x - 24)) \cdot \left(r \cdot n^2 \cdot f\right) + b \tag{4.4}$$

while the equation for memory allocation changes to equation 4.5:

$$6100000 + n \cdot 420 + n \cdot r \cdot 1760 \tag{4.5}$$

We can further simplify the equation by assuming that n goes towards infinity, then the only parts of the equations that matter are those involving parameters n and r, where n determines how fast the result of the equation increases. Let the packet size be 24 for comparison's sake. This gives us equations 4.6 and 4.7:

$$474.9 \cdot \left(r \cdot n^2 \cdot f\right) \tag{4.6}$$

$$n \cdot 420 + n \cdot r \cdot 1760 \tag{4.7}$$

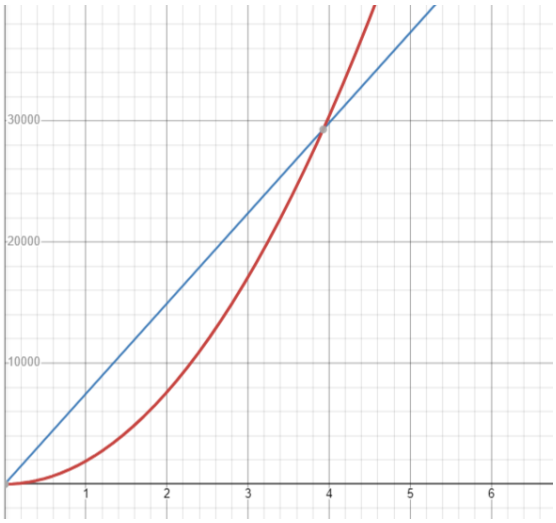
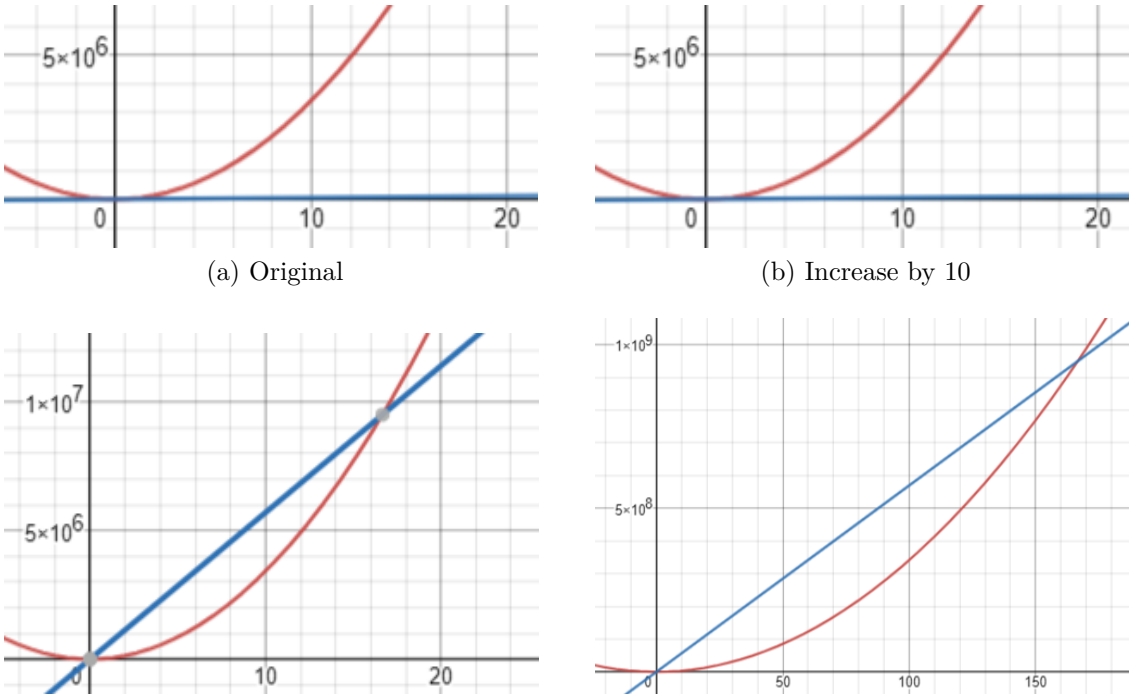


Figure 4.3: Graphs displaying how memory requirements (blue line), and network traffic requirements (red line) for a QoE of 60Hz and packet size of 24, scale with changing number of rooms (while both use the same room size of 4).

When the equations are compared in a graph in figure 4.2a, We can see that after four rooms, network traffic demands are always above memory allocation demands, even with an QoE of 1Hz. If the QoE is increased to 60Hz, network overtakes memory almost immediately, as can be seen in figure 4.2b. If the size of rooms is increased, network traffic grows even faster, as seen in 4.3. It is clear that the network traffic requirements grows quicker than the memory allocation requirements. Even more so if the packet size is increase beyond 24 bytes. What would happen if the constant multiplied with the parameters of the memory allocation requirement equation 4.7 changed? The form of the equations would remain, but each room could require more memory than used in this thesis. As seen in figure 4.2a, the network traffic requirement is not immediately higher than the memory allocation requirement. But because the growth of the network traffic requirement is quadratic, it will eventually overtake the memory allocation requirement. If the constants of the memory allocation equation grew, the point at which network traffic overtakes memory allocation will move further right in the graph.

What is required for the memory allocation requirement to grow quicker than network traffic requirement? Firstly, let us assume reasonable parameter values for the network traffic sub-model. The lowest frequency of packets acceptable for the QoE has been established to be 24Hz, and the lowest packet size is 24 bytes. Then assume that both sub-models use the same size of rooms, three. With this, the network traffic instantly overtakes the memory allocation requirement growth as seen in figure 4.4a. The effect of multiplying the constants of the memory allocation equation 4.7 by ten is not enough for a lasting impact, and seen in figure 4.4b. Once multiplied again by ten, the network traffic requirement overtakes the memory requirement at 17 rooms and forward in figure 4.4c. To once again multiply the constants by ten causes this to happen at 167 rooms in figure 4.4d. This shows that immense memory requirements per client and room are required for the memory allocation requirement to scale



(a) Original

(b) Increase by 10

(c) Increase by 100

(d) Increase by 1000

Figure 4.4: Bits of data required for network traffic (red line) and memory allocation (blue) as the constants of memory allocation are scaled by different factors.

faster than the network traffic requirement.

4.5 Comparing Cloud Providers

With completed sub-models for network traffic and memory allocation it is possible to calculate how many clients a chosen provider and container can sustain, given some parameters, if the models are changed to return the number of rooms that can be sustained given all the other parameters. The top three cloud providers (AWS, GCP and Azure) were compared to generally find what it would cost to host our relay server using them. Each provider offers hundreds of options, which can not all be tested, thus a selection has been made focusing on the E2 type (when available).

As noted before in section 2.4, different providers charge for the metrics in different ways. Let us firstly look at some containers from AWS in which CPU%, network traffic and memory allocation are grouped together for each container option. Table 4.14 shows 4 different instances provided by AWS. It confirms prior suspicions that the network bandwidth limit is what limits the number of clients a container can use. So containers that are pricier because they could technically have memory for more clients are limited by low network bandwidth limits. Thus, it becomes interesting to compare AWS pricing to that of GCP and Azure.

Table 4.15 shows different E2 containers from GCP. Containers from each type of E2 container are represented, and each type shows how many clients can be supported for

Table 4.14: The number of rooms that each metric can sustain for different AWS containers. Each uses the parameters QoE=60, r=10, and packet size of 24 bytes.

		Rooms required to run out of...		
Name	Cost ($/hour)	Memory	Bandwidth	Cost per Client per day
t3.micro	0.0104	487717,6359	1754,755387	0,00001423
t3.xlarge	0.1664	7808684,618	1754,755387	0,0002277
m7g.medium	0.0408	1951911,032	4386,888468	0,00002233
m7g.metal	2.6112	124944156,3	10528,53232	0,005953

Table 4.15: The number of rooms that each metric can sustain for different GCP containers. Each uses the parameters QoE=60, r=10, and packet size of 24 bytes.

		Rooms required to run out of...		
Name	Cost ($/hour)	Memory	Bandwidth	Cost per client per day
ec-standard-2	0,02324	10995,78863	1559,782566	0,00003575883024
ec-standard-8	0,092968	45027,54768	6239,130265	0,00003576190759
e2-highcpu-2	0,054468	26237,06576	1559,782566	0,00008380860437
e2-highcpu-8	0,217872	105988,7517	6239,130265	0,00008380860437
e2-highmem-2	0,09952	48225,34606	1559,782566	0,0001531290355
e2-highmem-8	0,39808	193941,8729	6239,130265	0,0001531290355

the option with the lowest network (outbound) bandwidth, and the first occurrence of the highest available network (outbound) bandwidth. The calculated number of clients for a certain outbound network traffic amount was calculated using the second part of equation 4.2, which considers the traffic send out from the server of a packet size y. When table 4.15 is compared to the AWS samples in table 4.14 it is evident that the AWS containers have the lowest cost per client. Thus, the best case of $0,00001423 per client per day had to be compared to what it would cost to host 1754 rooms of clients for a day using Azure.

Azure charges its users for the volume used of each metric (vCPU, memory allocation and network bandwidth). Thus, there are no pre-made containers to compare to. Instead, the cost of hosting 1754 rooms with room from 10 clients each, at QoE 60 and a packet size of 24 was calculated, to compare to the best cost so far provided by AWS. Azure, like GCP, only charges for outbound network traffic. For 1754 rooms that would be about 4,5 Gbps, or 0,56GBps. The cost per GB of network traffic using ISP is $0,08 [39]. Thus, the cost per client per day (only considering

network traffic) would be $0,046, which is already worse than every other container considered so far. The conclusion from this is that AWS seems to be able to sustain the most clients for the lowest cost.

5

Future Work

While the thesis concludes with sub-models for memory allocation and network bandwidth that can predict usages outside their test-range, much work remains to be done on the CPU% sub-model which has a limited range of predictability. Future work could easily build on the CPU% sub-model, but also improve the basis of all three sub-models. This chapter describes the future work required, while discussing why it could not be attained in this thesis, and ends with general reflections of work that can be done to improve the scalability-cost model.

5.1 Extend Test-Data

As seen in the section on results 4, the developed sub-models are often deducted from a small set of test cases, no more than ten. And this limited data was gathered closely to each other, for e.g., going from three to ten rooms, or using between five and 200 clients, while the relay server on smaller instances would be able to handle larger loads of clients.

Firstly, starting the simulations took time. Each creation of a client, by a client-simulator, took 1 second. This delay was added so that the simulator would not crash due to spending all its CPU time creating threads while ignoring received messages. Secondly, client-simulator could demand the simulator to be shutdown because they thought the QoE was missed. But due to resource sharing, clients were not always able to receive their messages in the QoE time-frame if too many clients were spawned on the same instance. So running simulations took some time, and would fail completely if not all clients could be spawned before the simulation crashed. So for each test, it was necessary to figure out how many instances were required for the client-simulators to no crash due to resource sharing. This meant repeating experiments until the correct distribution of clients across, i.e., AWS containers, was achieved.

So one reason for the few test-cases was the complications with client-simulators. But this could have been solved by using more AWS containers. In total, 3 client-simulator instances were created. No more were created to not charge Opera unknown amounts of money for the instances, since the free period for the T3.micro instance had passed. But this only covers the case when test were done using AWS. Other struggles arose when gathering CPU% data. The test cases were performed using a Windows

computer running WSL, and a macOS laptop running the server. Then the tests were limited to the resources on the Windows machine, and could not be increased.

There was also an issue of visual clutter. Each client simulator using AWS required one terminal per client-simulator program. And when the program was updated, each container was updated individually. Both of these took time, and might have been able to be done more efficiently. To be able to easier maintain the client-simulator containers would have decreased the time spent on maintenance, that could instead have been spent on testing and analyzing.

5.2 Testing Sub-Models on different hardware

The underlying hardware of the containers provided by cloud providers is not the same for all containers. For example, different containers from AWS use different processors for the same type of instances (EC2). Their M7g instances use Graviton3 processors, while the T3 instances use 1st or 2nd generation Intel Xeon Platinum 8000 series processors [23]. This, and many other factors that differ between the instances, could affect the scaling of all sub-models developed in this thesis. In this case, this thesis could act as guidance on how to perform tests to gather data for different instances.

5.3 Improve the CPU% Sub-Model

The major problem with developing the CPU% sub-model was the environment the server ran on. At first, tests were done using AWS containers, and the virtual machine environment presumably caused issues with attaining consistent readings from test cases. For e.g., the same test case could result in either 3% or 5% (a 67% increase). Without consistency, no conclusion could be drawn from the gathered data. But it is worth noting that data gather on vCPU was done with the command `pidstat`, which only returned the average CPU% for a period of time. But the problem was theorized to stem from the virtual environment of most cloud provided containers. In these, the user does not get exclusive access to hardware. Instead, they get vCPU, where each vCPU is one thread from a real CPU. So to get exclusive access without splurging on renting an exclusive access machine, the environment of the relay-server had to be changed to the one used to attain the results in section 4.3.

The new environment was using two desktops, one ran the server and one ran clients. This made CPU% readings consistent. But the number of possible clients were limited to the resources of the desktop that spawned them, which caused the gaps present in the table in appendix D.1. There was an attempt to use AWS instances to spawn clients, but I could not figure out how to allow the instances to send UDP packets outside their security group. If this was fixed, even more clients would have been able to connect to the bare-metal machine server and thus would have been able to test a larger range of cases and their CPU%.

Readings from the new environment could also be considered inconsistent. Appendix D.3 shows this, where each reading from the command `top -l 15`, for the tests on QoE=40 and room size three, is shown. While the average of the measurements continuously increase, the measurements they are based on are fairly inconsistent. The first concern is that the first measurement was always 0%. Secondly, measurements often started off small, then fluctuated between larger values. The server should manage an equal amount of messages each second, and has been confirmed to do so when creating the network traffic sub-model, so the cause of the inconsistency is unknown. But it is worth noting that some measurements naturally are above the average CPU% and thus the chosen constellation of containers and resources would in actuality benefit from a slight overestimation of CPU resources.

Future work should consider how to improve measurements of CPU%. Looking at other papers on provisioning resources, like the one from Vlad Nae et al. [29], could be a great starting point. Although their model was made for a different client-server model, parts of Vlad Nae et al. model is similar to our relay server model. For e.g., the further divide the model by considering the processing required to receive and send messages per client. This would probably not have been possible to measure using the test-setup of this thesis, since CPU% measurements have been rough and inconsistent. So in future work, the test setup should enable testing that could give smaller and more accurate CPU% readings. Other things to consider would be to increase the measurements per test case and performing more test-cases with a larger number of clients connected to the server.

Future work could also benefit from moving away from using regression models. While effective for the range of data they are trained on, this is their limit. The models can not be used to accurately predict values outside the range of the training data. This obviously limits the CPU% sub-model. The model can also not predict values for QoE. A better model would be able to predict CPU% for all combinations of future parameters.

5.4 Other Reflections

One thing to note regarding the memory allocation sub-model is that it has an initial requirement for when no clients have connected to the server. From section 4.3, it is known that 6100kbit of memory is required. This is the only baseline-requirement of every added container. The easy way to manage this would be to consider that every possible container has 6100kbit less memory available, and exclude it from any memory calculations.

The first part of the scalability-cost model should be to find if any container exists which perfectly matches the requirements. This case is highly unlikely. But to check this, the final equations of each sub-module should be used and checked against all selectable container.

Then it is necessary to consider the part of the requirements which are strictly bound to clients and rooms:

- Network traffic should be considered scaling per room, since the traffic depends on the size of rooms, and QoE, since it increases the frequency of messages.
- Memory required per client is independent of rooms, and set at 1,72kbit per client.
- CPU% is also dependent on the size of rooms and QoE since it is correlated to the rate at which network packets are sent and received.

There exist many containers which are of the same type. Previously, the EC2 (AWS) and E2 (GCP) containers, which come in many varieties, have been discussed and will be considered the type of container the foundation of the scalability-cost model is created from. The difference between these container-types are on what metric they focus on, computational power, memory, or network bandwidth. And since the distribution of the sub-model produced requirements can not be separated, they have to be hosted at a consistent ratio (for example, if a container has double the memory it should also mean that double the amount of clients are serviced by the same container), only a single type of EC2/E2 containers should have to be considered at any given time. These assumptions make it easier to construct the scalability-cost model.

As discussed previously in section 2.4, the cost of containers typically increases at the same rate that the memory scales. It might follow the scaling of other metrics, but it always follows the memory as seen from AWS's pricing tables [24].

6

Conclusion

Three sub-models were successfully created. Two of them can predict future values: The memory allocation sub-model and the network traffic sub-model. They produce requirements that are at most 2,658% off from the reality. Meanwhile, the CPU sub-model is limited to the range it is trained on. This is an obvious setback that could be improved with future work, to predict future values. But combined, the three sub-models can be used to predict bottlenecks of using different containers within a range of tested parameters. Between the memory allocation and network traffic requirements, network traffic is generally what limits the amount of clients can be hosted on a container.

The three largest cloud providers and their container options were compared. It was found that, even though AWS will let large amounts of possible memory allocation go unused, creates the lowest cost per client for a certain set of parameters (QoE 60, packet size of 24, 10 clients per room). If not leaving nay resources unused is the priority, Azure is the provider to go for since they charge only for the exact resources you use (and do not provide pre-packaged containers like AWS and GCP). Meanwhile, GCP lands in between the other two providers in that it cost more per client than using AWS but leaves less memory unused.

Bibliography

[1] H. S. Oluwatosin, "Client-server model," *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 67–71, 2014.

[2] S. Flinck Lindström, M. Wetterberg, and N. Carlsson, "Cloud gaming: A qoe study of fast-paced single-player and multiplayer gaming," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 34–45. DOI: 10.1109/UCC48980.2020.00023.

[3] *Charts overview*. [Online]. Available: https://store.steampowered.com/charts/ (visited on 11/09/2022).

[4] "Epic games on aws." (2023), [Online]. Available: https://aws.amazon.com/solutions/case-studies/innovators/epic-games/ (visited on 04/25/2023).

[5] *Steamcharts - pubg: Battlegrounds*. [Online]. Available: https://steamcharts.com/app/578080#7d (visited on 04/17/2023).

[6] V. Kumar and P. Vidhyalakshmi, "Cloud computing for business sustainability," *Asia-Pacific Journal of Management Research and Innovation*, vol. 8, no. 4, pp. 461–474, 2012.

[7] *Survival servers*. [Online]. Available: https://www.survivalservers.com (visited on 04/18/2023).

[8] *Gameservers*. [Online]. Available: https://www.gameservers.com (visited on 04/18/2023).

[9] *Basic installation*. [Online]. Available: https://ubuntu.com/server/docs/installation (visited on 04/18/2023).

[10] *Dream it. build it. grow it.* [Online]. Available: https://www.digitalocean.com (visited on 04/18/2023).

[11] P. Srivastava and R. Khan, "A review paper on cloud computing," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 8, no. 6, pp. 17–20, 2018.

[12] R. F. Buse *et al.*, "Why use cloud computing?" *Annals of University of Craiova-Economic Sciences Series*, vol. 3, no. 39, pp. 228–231, 2011.

[13] "What is containerization?" (2023), [Online]. Available: https://aws.amazon.com/what-is/containerization/ (visited on 06/09/2023).

[14] "Use containers to build, share and run your applications." (2023), [Online]. Available: https://www.docker.com/resources/what-container/ (visited on 06/09/2023).

[15] F. Richter. "Amazon, microsoft & google dominate cloud market." (2022), [Online]. Available: https://www.statista.com/chart/18819/worldwide-

market-share-of-leading-cloud-infrastructure-service-providers/ (visited on 04/20/2023).

[16] "What is aws?" (2023), [Online]. Available: https://aws.amazon.com/what-is-aws/ (visited on 01/24/2023).

[17] "Aws for games." (2023), [Online]. Available: https://aws.amazon.com/gametech/?nc2=h_ql_sol_ind_gt (visited on 01/30/2023).

[18] "Vm instance pricing." (2023), [Online]. Available: https://cloud.google.com/compute/vm-instance-pricing (visited on 03/27/2023).

[19] "Azure container instances pricing." (2023), [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/container-instances/#pricing (visited on 04/19/2023).

[20] "What is amazon ec2?" (2023), [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html (visited on 01/30/2023).

[21] "Amazon ec2 pricing." (2023), [Online]. Available: https://aws.amazon.com/ec2/pricing/ (visited on 01/31/2023).

[22] "Amazon ec2 dedicated hosts pricing." (2023), [Online]. Available: https://aws.amazon.com/ec2/dedicated-hosts/pricing/ (visited on 01/31/2023).

[23] "Amazon ec2 instance types." (2023), [Online]. Available: https://aws.amazon.com/ec2/instance-types/ (visited on 01/30/2023).

[24] "Amazon ec2 on-demand pricing." (2023), [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/ (visited on 01/31/2023).

[25] "All networking pricing." (2023), [Online]. Available: https://cloud.google.com/vpc/network-pricing (visited on 03/27/2023).

[26] S. El Kafhali, I. El Mir, K. Salah, and M. Hanini, "Dynamic scalability model for containerized cloud services," *Arabian Journal for Science and Engineering*, vol. 45, pp. 10 693–10 708, 2020.

[27] Y. Sun, J. White, S. Eade, and D. C. Schmidt, "Roar: A qos-oriented modeling framework for automated cloud resource allocation and optimization," *Journal of Systems and Software*, vol. 116, pp. 146–161, 2016.

[28] B. Bhavani and H. Guruprasad, "Resource provisioning techniques in cloud computing environment: A survey," *International Journal of Research in Computer and Communication Technology*, vol. 3, no. 3, pp. 395–401, 2014.

[29] V. Nae, A. Iosup, and R. Prodan, "Dynamic resource provisioning in massively multiplayer online games," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 3, pp. 380–395, 2010.

[30] T. Cannon, "Fight the lag! the trick behind ggpo's low latency netcode," *Game Developer Magazine's*, pp. 7–12, Sep. 2012.

[31] J. Y. C. Chen and J. E. Thropp, "Review of low frame rate effects on human performance," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 37, no. 6, pp. 1063–1076, 2007. DOI: 10.1109/TSMCA.2007.904779.

[32] S. Zadtootaghaj, S. Schmidt, and S. Möller, "Modeling gaming qoe: Towards the impact of frame rate and bit rate on cloud gaming," in *2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX)*, 2018, pp. 1–6. DOI: 10.1109/QoMEX.2018.8463416.

[33] S. Zander and G. Armitage, "A traffic model for the xbox game halo 2," in *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, 2005, pp. 13–18.

[34] T. Lang, P. Branch, and G. Armitage, "A synthetic traffic model for quake3," in *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 2004, pp. 233–238.

[35] S. Ratti, B. Hariri, and S. Shirmohammadi, "A survey of first-person shooter gaming traffic on the internet," *IEEE Internet Computing*, vol. 14, no. 5, pp. 60–69, 2010.

[36] "Amazon cloudwatch." (2023), [Online]. Available: `https://aws.amazon.com/cloudwatch/` (visited on 05/24/2023).

[37] "Pmap(1) — linux manual page." (2023), [Online]. Available: `https://man7.org/linux/man-pages/man1/pmap.1.html` (visited on 05/03/2023).

[38] "Pidof(1) — linux manual page." (2023), [Online]. Available: `https://man7.org/linux/man-pages/man1/pidof.1.html` (visited on 05/03/2023).

[39] "Bandwidth pricing." (2023), [Online]. Available: `https://azure.microsoft.com/en-us/pricing/details/bandwidth/` (visited on 06/10/2023).

A

GCP E2 Pricing

A.1 GCP E2 Pricing

Pricing of GCP E2 machines as of March 2023 [18]. Network traffic is charged separately.

Machine Type	vCPUs	Memory (GB)	Maximum egress bandwidth (Gbps)	Price/Hour ($)
e2-standard-2	2	8	4	0,02324
e2-standard-4	4	16	8	0,046479
e2-standard-8	8	32	16	0,092958
e2-standard-16	16	64	16	0,185917
e2-standard-32	32	128	16	0,371834
e2-highcpu-2	2	2	4	0,054468
e2-highcpu-4	4	4	8	0,108936
e2-highcpu-8	8	8	16	0,217872
e2-highcpu-16	16	16	16	0,425744
e2-highcpu-32	32	32	16	0,871488
e2-highmem-2	2	16	4	0,09952
e2-highmem-4	4	32	8	0,19904
e2-highmem-8	8	64	16	0,39808
e2-highmem-16	16	128	16	0,79616

B

Flowcharts

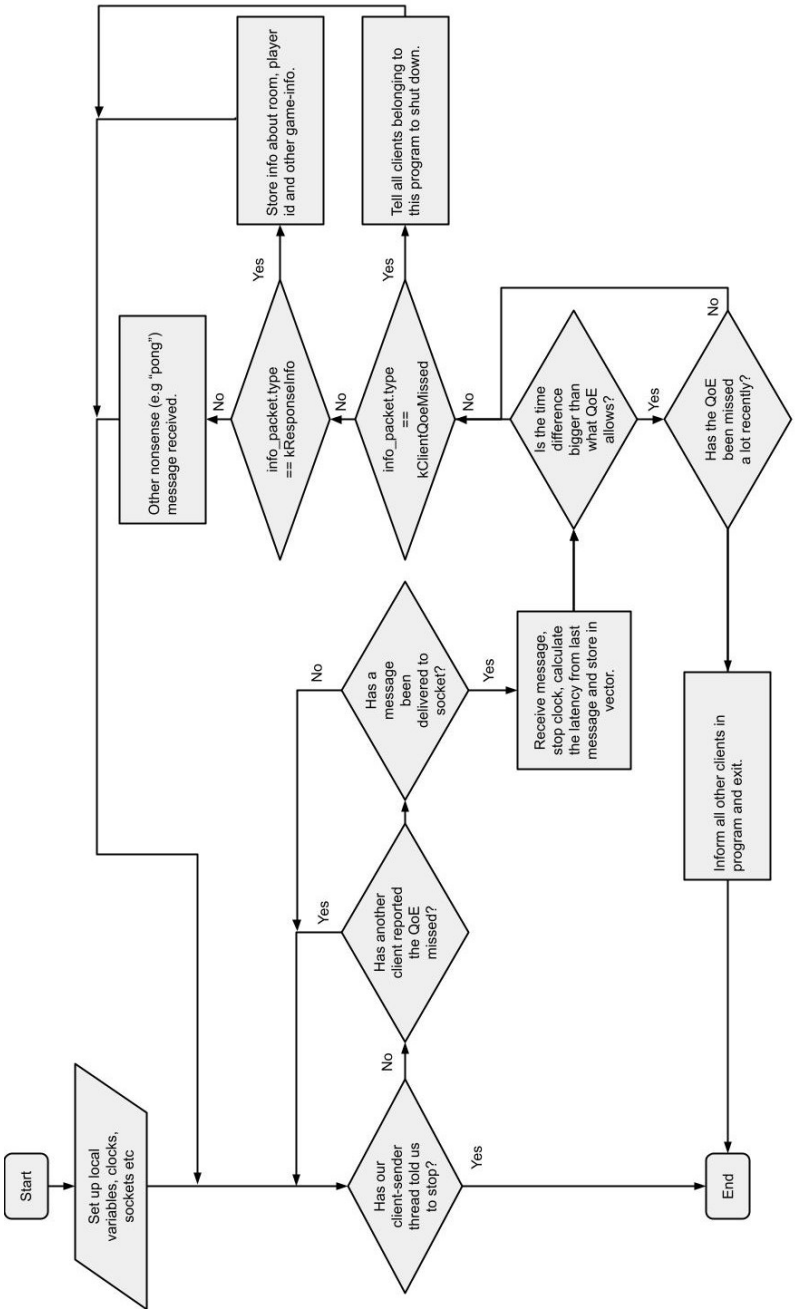


Figure B.1: Flowchart of the client-receiver thread.

C

Network Traffic

C.1 Meassure Network Traffic and Packet Sizes

Table C.1: Comparing measured network traffic (bits/s) to server-measured data (byte) sent and received by server for different room sizes. Every measurement is per second.

Total clients	Room Size	Packet size (byte)	Bytes client-to-server	Bytes server-to-client	Packets client-to-server	Packet server-to-client	Traffic (bit/s) client-to-server	Traffic (bit/s) server-to-client	Traffic per byte	Traffic per Packet
15	3	24	10515,4	21024,0	438,2	876,0	181,16 K	469,05 K	20,26	486,3
15	3	48	21925,7	42048,0	438,1	876,0	262,94 K	628,39 K	13,95	669,74
15	3	64	28003,4	56000,0	437,7	875,0	316,94 K	736,69 K	12,41	794,1
15	3	80	35043,4	70080,0	428,2	876,0	370,87 K	844,32 K	11,45	916,12
20	4	24	14021,1	42048,0	584,3	1752,0	240,34 K	921,63 K	20,5	492,55
20	4	48	28001,8	83995,2	583,5	1749,9	347,94 K	1,21 M	13,91	667,67
20	4	64	37379,2	112128,0	584,2	1752,0	429,47 K	1,43 M	12,31	787,7
20	4	80	46723,4	140160,0	584,2	1752,0	494,62 K	1,64 M	11,36	908,91
25	5	24	17521,7	70080,0	730,1	2920,0	298,56 K	1,48 M	20,17	484,19
25	5	48	35043,4	140160,0	730,2	2920,0	435,15 K	2,02 M	13,95	669,53
25	5	64	46723,4	186880,0	730,2	2920,0	527,15 K	2,38 M	14,32	793,36
25	5	80	58555,4	234208,0	732,1	2927,6	617,80 K	2,73 M	11,397	911,71
30	6	24	21054,1	105228,0	877,4	4384,5	358,92K	2,22 M	20,33	487,98
30	6	48	42171,4	210840,0	878,7	4392,5	523,14 K	3,03 M	13,999	671,94
30	6	64	56257,7	281280,0	879,1	43950	633,62 K	3,57 M	12,42	794,9
30	6	80	70321,7	351600,0	879,1	43950	744,28 K	4,11 M	11,48	918,27

C.2 Compare Network Traffic and Packet Sizes

Table C.2: Comparing measured network traffic, subtracting idle traffic, with calculated packets and bytes send and received by the server per second for different numbers of clients and room sizes at 30Hz.

Total Clients	Room Size	Packet size (bytes)	Total nmb of Packets	Total nmb of bytes	Total measured network traffic (kbit/s) excluding idle	Traffic per byte	Traffic per Packet
15	3	24	1350	32400	638,99	19,72	473.33
15	3	48	1350	64800	880,11	13,58	651,93
15	3	64	1350	86400	1042,41	12,065	772,16
15	3	80	1350	108000	1203,97	11,15	891,83
20	4	24	2400	57600	1150,75	19,98	479,48
20	4	48	2400	115200	1546,72	13,43	644,47
20	4	64	2400	153600	1848,25	12,03	766,35
20	4	80	2400	192000	2123,4	11,06	884,75
25	5	24	3750	90000	1767,34	19,64	471,29
25	5	48	3750	180000	2443,93	13,58	651,71
25	5	64	3750	240000	2895,93	12,07	772,25
25	5	80	3750	300000	3336,58	11,12	889,75
30	6	24	5400	129600	2567,7	19,81	475,5
30	6	48	5400	259200	3541,92	13,66	655,91
30	6	64	5400	345600	4192,4	12,13	776,37
30	6	80	5400	432000	4843,06	11,21	896,86

C.3 Comparing Measured and Calculated Traffic

Table C.3: Comparing measured traffic, used to construct the equation, to calculated traffic with error.

Total Clients	Room Size	Packet size (byte)	Measured Traffic (kbit/s)	Calculated Traffic (kbit/s)	Error (%)
15	3	24	638,99	641,115	0,33
15	3	48	880,11	882,495	0,27
15	3	64	1042,41	1043,415	0,1
15	3	80	1203,97	1204,335	0,03
20	4	24	1150,75	1139,760	-0,96
20	4	48	1546,72	1568,880	1,43
20	4	64	1848,25	1854,000	0,31
20	4	80	2123,4	2141,040	0,83
25	5	24	1767,34	1780,875	0,77
25	5	48	2443,93	2451,375	0,30
25	5	64	2895,93	2898,375	0.08
25	5	80	3336,58	3345,375	0.26
30	6	24	2567,7	2564,460	-0,13
30	6	48	3541,92	3529,980	-0,34
30	6	64	4192,4	4173,660	-0,45
30	6	80	4843,06	4817,340	-0,53

D

CPU Utilization

D.1 CPU Utilization Measurements

Table D.1: The average CPU%, calculated from 13 samples, for different parameters. #Rooms (r) signifies the number of rooms used for a test case. Some test-cases crashed the client-simulator before measurements could be taken (marked with -). Gray rows were used as testing set.

QoE	r	Room Size									
		1	2	3	4	5	6	7	8	9	10
10	2	2,617	3,0167	3,108	4,2	4,625	5,7	6,025	6,433	7,15	8,167
	4	2,642	3,442	4,217	5,358	6,942	7,383	8,717	9,433	10,533	12,467
	6	2,7	3,767	4,8	6,642	8,375	9,233	10,983	12,008	14,367	16,658
	8	2,767	3,883	5,5	7,717	9,967	11,575	13,142	15,0167	17,358	21,233
	10	2,817	4,167	6,042	8,833	11,025	12,633	14,85	18,225	22,35	-
20	2	2,617	3,317	4,258	5,275	6,575	8,225	9,7	10,325	11,85	26,342
	4	2,683	4,033	5,833	7,292	10,242	12,217	14,967	15,742	19,842	22,792
	6	2,758	4,7	7,0	10,075	13,175	16,267	18,467	21,83	26,65	30,758
	8	2,817	5,075	8,333	12,167	15,508	18,717	21,675	28,075	33,142	36,125
	10	2,917	5,5	9,867	13,217	18,342	20,2	26,125	32,308	36,375	-
30	2	2,725	3,742	4,733	6,133	7,317	9,05	10,483	11,9	14,192	16,408
	4	2,825	4,425	6,292	8,575	11,35	13,617	17,9	21,75	25,0	30,633
	6	2,883	4,917	8,092	11,45	14,858	19,05	23,4	29,65	34,358	37,008
	8	2,933	5,833	9,975	13,7	18,542	23,15	28,358	35,842	37,3	-
	10	3,05	6,9	11,408	16,675	22,242	25,33	30,95	37,383	-	-
40	2	2,75	4,05	5,483	7,408	10,292	12,575	14,85	16,308	18,475	18,975
	4	2,942	5,242	8,083	9,908	16,342	21,0	25,367	26,275	30,667	30,192
	6	3,108	6,525	10,2	13,792	22,617	27,658	32,683	35,025	36,725	-
	8	3,108	7,242	12,358	17,425	26,6	30,875	36,042	38,1	-	-
	10	3,383	8,683	14,3417	19,433	30,558	34,358	39,4	-	-	-

D.2 Regression Model Code for QoE 30Hz

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import numpy as np

X_q30 = np.array([
    [1, 2], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2], [7, 2],
        [8, 2], [9, 2], [10, 2],
    [1, 6], [2, 6], [3, 6], [4, 6], [5, 6], [6, 6], [7, 6],
        [8, 6], [9, 6], [10, 6],
    [1, 8], [2, 8], [3, 8], [4, 8], [5, 8], [6, 8], [7, 8],
        [8, 8], [9, 8],
    [1, 10], [2, 10], [3, 10], [4, 10], [5, 10], [6, 10],
        [7, 10], [8, 10]])

y_q30 = np.array(
    [2.725, 3.741666667, 4.733333333, 6.133333333, 7.316666667,
        9.05, 10.48333333, 11.9, 14.19166667, 16.40833333,
    2.883333333, 4.916666667, 8.091666667, 11.45, 14.85833333,
        19.05, 23.4, 29.65, 34.35833333, 37.00833333,
    2.933333333, 5.833333333, 9.975, 13.7, 18.54166667, 23.15,
        28.35833333, 35.84166667, 37.3,
    3.05, 6.9, 11.40833333, 16.675, 22.24166667, 25.333333,
        30.95, 37.38333333])

x_q30_test = np.array([
    [1, 4], [2, 4], [3, 4], [4, 4], [5, 4], [6, 4], [7, 4],
        [8, 4], [9, 4], [10, 4]])
y_q30_test = np.array(
    [2.825, 4.425, 6.291666667, 8.575, 11.35, 13.61666667,
        17.9, 21.75, 25, 30.63333333])



#########################################################
# Linear
#########################################################

# Fit a linear regression model to the q=30 data
model_q30 = LinearRegression()
model_q30.fit(X_q30, y_q30)

## r-squared value
y_pred = model_q30.predict(X_q30)
```

```
r_squared = r2_score(y_q30, y_pred)
print("R-sqaured value of linear: ", r_squared)

#####################################################
# Second degree Polynomial
#####################################################

# Step 2: Define the degree of the polynomial
degree = 2

# Step 3: Create polynomial features
poly_features = PolynomialFeatures(degree=degree)
X_q30_poly = poly_features.fit_transform(X_q30)

# Step 4: Fit the polynomial regression model
model_q30_P2 = LinearRegression()
model_q30_P2.fit(X_q30_poly, y_q30)

## r-squared value
y_pred = model_q30_P2.predict(X_q30_poly)
r_squared = r2_score(y_q30, y_pred)
print("R-sqaured value of 2nd degree polynomial: ", r_squared)

#####################################################
# Third degree Polynomial
#####################################################

# Step 2: Define the degree of the polynomial
degree = 3

# Step 3: Create polynomial features
poly_features = PolynomialFeatures(degree=degree)
X_q30_poly = poly_features.fit_transform(X_q30)

# Step 4: Fit the polynomial regression model
model_q30_P2 = LinearRegression()
model_q30_P2.fit(X_q30_poly, y_q30)

## r-squared value
y_pred = model_q30_P2.predict(X_q30_poly)
r_squared = r2_score(y_q30, y_pred)
print("R-sqaured value of 3rd degree polynomial: ", r_squared)

#####################################################
# R-squared of test set prediction
#####################################################
```

```
x_q30_poly_test = poly_features.fit_transform(x_q30_test)
prediction_test = model_q30_P2.predict(x_q30_poly_test)
r_squared = r2_score(y_q30_test, prediction_test)
print("R-sqaured value of 3rd degree polynomial,
    30Hz test-set: ", r_squared)
```

D.3 Top Measurements for QoE 40hz, Bare-Metal Machine

Table D.2: The CPU% measurements using *top -l 15* for the test-group QoE = 40hz and a room-size of 3. Highlighted in gray are measurements that do not continuously increase with the number of rooms. The average was calculated from 13 values, removing the two smallest and the largest from the set of measurements.

Measurement number	The Number of Rooms				
	2	4	6	8	10
1	0	0	0	0	0
2	4,1	6,2	5	5,9	10,7
3	5,1	9,6	7,6	8,8	15,2
4	4,6	9,4	7	8	15,3
5	5,1	9,6	6,8	15,7	17,1
6	4,2	8,4	13,4	15,7	17,3
7	4,2	8,8	13	14,3	17,6
8	6,3	6	13,1	13,6	14,1
9	6,1	6,5	11,7	15,9	8,3
10	5,6	6,5	12,7	14,9	10,2
11	6,3	6,2	11,6	15,3	9,7
12	6	8,8	11,8	14,2	13
13	6,6	9,3	11,2	10	15,4
14	6,7	9,3	8,3	9,3	17,4
15	5,7	8	7,6	8,5	16,7
Average	5,48	8,083	10,2	12,36	14,34

D.4 R-Squared Values of different Regression Models

Table D.3: The R-Squared value values of predicted CPU% values from different regression models, based on the training-data for different QoE.

Regression Model	QoE	r2	Average r2
Linear	10	0,874	0,875
	20	0,905	
	30	0,87	
	40	0,852	
Polynomial (degree 2)	10	0,993	0,982
	20	0,978	
	30	0,990	
	40	0,968	
Polynomial (degree 3)	10	0,995	0,992
	20	0,982	
	30	0,996	
	40	0,993	