Remote rendering of physic simulations and scalability aspects in web applications

Lars Lindberg

June 9, 2014
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Jan Erik Moström
Examiner: Fredrik Georgsson

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

This thesis, assigned by Algoryx Simulation AB, explores the concept of implementing a web application for managing Algoryx based physics simulations. The application enables users to get access to the Algoryx simulation software by providing a user interface for clients to submit scenes files to be simulated, and to view a 3D visualization of the finished simulations rendered directly in the web browser. This allows clients to do away with the work of performing the compute intensive physic simulations locally and instead hand over that responsibility to the web service, while the client only needs to handle the actual rendering. Applications made available through a web browser allows users to get easy access to applications that does not require any installation procedures. For that reason the clients do not need any simulation software or any plugin installed to access the service. This makes it easy to share results of simulations to customers by just giving out a link that can be accessed through a browser.

This paper also includes a theoretical study on scalability in web applications. The theory explains different ways of scaling, and common techniques and methods used to help achieving scalability that can be useful when designing and building scalable web system.
## Contents

1 Introduction .................................................. 1  
   1.1 Algoryx .................................................. 1  
   1.2 Report outline ......................................... 2  

2 Problem Description ........................................... 3  
   2.1 Problem Statement ...................................... 3  
   2.2 Goals .................................................... 3  
   2.3 Purposes ................................................ 4  
   2.4 Related Work ............................................ 4  

3 Scalability in web applications ................................ 5  
   3.1 Scaling Vertically ....................................... 5  
   3.2 Scaling Horizontally .................................... 6  
   3.3 Load balancing ........................................... 6  
      3.3.1 Strategies .......................................... 7  
      3.3.2 DNS load balancing ................................. 8  
      3.3.3 Dedicated load balancers ......................... 11  
   3.4 Caching .................................................. 13  
      3.4.1 Static Content .................................... 13  
      3.4.2 Dynamic Content .................................. 13  
      3.4.3 Different cache architectures .................... 14  
      3.4.4 Cache deployments ................................. 15  
   3.5 Reverse proxy ........................................... 17  
   3.6 Message/Task queues .................................... 18  
   3.7 Scaling databases ...................................... 19  
      3.7.1 Master-Slave ....................................... 20  
      3.7.2 Master-Master ..................................... 21  
      3.7.3 Sharding ........................................... 22  
      3.7.4 NoSQL ............................................... 23  
   3.8 Cloud computing ........................................ 23  
      3.8.1 Auto scaling ....................................... 24
4 Implementation
  4.1 Development tools ............................................. 29
  4.2 Introduction .................................................... 29
  4.3 Web framework structure ....................................... 30
    4.3.1 URL Dispatcher ........................................... 30
    4.3.2 View ...................................................... 31
    4.3.3 Template ................................................ 31
    4.3.4 Model ..................................................... 32
  4.4 User management ............................................... 32
  4.5 Simulation management ....................................... 32
    4.5.1 Submitting simulations .................................. 33
    4.5.2 Viewing simulations ...................................... 33
  4.6 Database ....................................................... 34
  4.7 Ease of access .................................................. 35
  4.8 User interface ................................................ 36
    4.8.1 User authentication ...................................... 36
    4.8.2 Submitting simulation .................................... 37
    4.8.3 View simulation status .................................. 37
    4.8.4 View processed simulation ............................... 38

5 Discussion ....................................................... 43
  5.1 Scalability options ............................................ 43
  5.2 Conclusions ................................................... 44
  5.3 Limitations ..................................................... 45
  5.4 Future work .................................................... 46

6 Acknowledgements ................................................. 47

References .......................................................... 49
List of Figures

3.1 A simple load balancing example. A load balancer is distributing client requests to a set of servers. ................................. 7
3.2 DNS Tree. ......................................................... 9
3.3 DNS Round Robin. ............................................. 11
3.4 Example of different caches and their locations. ......................... 15
3.5 An example of how a queue might be used in a web application. ........ 19
3.6 Master-Slave database configuration. ................................ 20
3.7 Google auto scale framework example. ............................. 25
3.8 AWS batch processing architecture example. ......................... 27
4.1 Django Architecture. ............................................ 31
4.2 The register view. An new account requires a valid user name and password. 36
4.3 The log in view. A user name and password of an existing account needs to be provided to sign in .......................... 37
4.4 View for submitting a simulation for processing. ...................... 38
4.5 The status and various options of the simulations an user has submitted can be viewed in a table. .............................. 39
4.6 A processed simulation is being viewed. ............................ 40
4.7 The playback menu. ............................................. 41
5.1 Example on a deployment of the implemented web service. ........... 45
List of Tables

4.1 Tools used for the development ........................................... 29
4.2 WebGL and WebSockets compatibility for major web browsers. ............ 36
4.3 Key bindings when viewing a simulation .................................... 40
Chapter 1

Introduction

Web applications allow the user to get easy direct access to applications, without the extra steps of conventional software with manual download/installation procedure. With new web libraries such as WebGL, graphics rendering in specific becomes accessible through major web browser vendors without the need to use non-standard 3rd party extensions/plugins.

As such the web browser can be utilized as a remote viewer. This can be suitable with physic simulations where the simulation process and visualization can be separated. The computationally intensive physics simulation can then be made to execute on a server back-end, with multiple users connecting through a web browser as thin clients to render a viewable 3D visualization of the simulated scene.

The back-end computing resources can be optimized and made scalable to perform well under high load and process many simulations simultaneously. The clients can connect and use the service on less powerful machines without needing any simulation software installed, while still having access to the physic simulation capabilities through a web interface.

This thesis will explore the concept of implementing such a web service.

1.1 Algoryx

This thesis was done in collaboration with Algoryx Simulation AB. Algoryx, a company based in Umeå that started in 2007, is a leading provider of visual and interactive physics based simulations providing products both for the professional and education market.

By creating a web service for users to easily access the Algoryx simulation software through a web browser opens up new possibilities. Users would no longer need any specific simulation software on their computer to take part of processed simulation results, making it easy to share them with customers. Instead of sending them video or picture representations they can be given a link leading to a web page that will render the results directly in the web browser. This opens up new ways of displaying simulations, both to existing customers as well as to the public, and a quick way of publishing simulations by just giving out a link.

The service would also serve as a hub for processing simulations. Users would not need a powerful computer to get their scenes simulated by the AgX physic engine. A user could submit many simulations to the service and still continue doing other work while the back-end processing part takes care of the compute intensive tasks. When the processing is done for a submitted simulation, the user would receive a link that he/she could share.
1.2 Report outline

The report is organized as the following:

– **Chapter 2**: Describes the project more in detail and presents the goals and purposes of the master thesis.

– **Chapter 3**: Will cover a theoretical study on scalability in web applications. The theory presented can be used when designing and building scalable web systems and will cover different ways of scaling and common techniques/methods used to help achieve scalability.

– **Chapter 4**: Explains how the web service is implemented and what tools were used. Furthermore, a description is given on how the application works and is used.

– **Chapter 5**: Some points are discussed and given on scalability options for the application in an actual production deployment. Conclusions are also drawn on the project and limitations on the application are presented together with possibilities for future work.

– **Chapter 6**: Acknowledgements are given to the people who contributed to this thesis project.
Chapter 2

Problem Description

This thesis will explore and evaluate the concept of implementing a web-based application for 3D visualization of physics simulations. This will include working with various web techniques to solve such a system.

2.1 Problem Statement

In this service users should be able to easily connect as thin clients through a web browser to pick from and view simulated scenes. The graphic rendering is done by the client using WebGL, while the computationally intensive work of simulating a scene is done by the server. The server and the clients will use WebSockets for communication and data transfer, and the GUI for the client will be made using HTML5.

The project will result in a proof of concept prototype that satisfies the given goals and can demonstrate the system.

The prototype implemented will mostly be suited for a development environment. This means that the prototype will not necessary be implemented to handle a large user base, ready to be deployed and used in a production environment. An in-depth theoretical study will for this reason be made on scalability aspects in web application, looking at different concepts such as horizontal scaling, load balancing and task queues.

2.2 Goals

The goal set for the prototype is that it should fulfill the following requirements.

- **Authentication, User based system** The service should have some simple functionality to support users. It should be possible for users to become authenticated in order to be able to log in to the system.

- **Submit simulations** A user should be able to submit a simulation scene to the system for processing.

- **Feedback on submitted simulations** A user should be able to see the current status of a submitted simulation (Running, pending, failed, finished, etc).

- **View simulations** A user should be able to view/evaluate a finished simulation of a submitted simulation scene.
– **Easy access** A user should only need to use a web browser to access the service, without the need of non-standard 3rd party extensions/plugins.

### 2.3 Purposes

The main purpose of this project is to examine how to make it possible for users to evaluate animated simulation data without the need to download the whole simulation first. The aim for the prototype is to inspire and possibly function as a base to build upon to create a production ready service to be used by the Algoryx user base.

One of the purposes of the service is to give users a possibility to do away with heavy processing simulation tasks, moving the work away from the client computer. Another purpose is to make it easier to share and display processed simulations to others and perhaps open a way for quicker feedback. The service removes the need to have the simulation software installed on the computer in order to be able to view simulations. The simulations are instead made available to view through the web making them more accessible. By making the service easy to access, minimizing the effort needed to use the product, it is easier to reach out to customers and make people more likely to try the product.

### 2.4 Related Work

Two web applications available today that provides related services are GrabCAD [22] and Lagoa [29].

**GrabCAD**

Offers a cloud based collaboration tool for designing and building physical products. GrabCAD provides functionality for engineering teams to manage and sharing CAD files, and also viewing CAD models through a web browser without needing any installation or plugins. GrabCAD aims to simplify for engineers to work together by being able to synchronizing local CAD files to on-line cloud projects, managing versions of files and locking files to prevent conflicts. There is support for setting up and managing projects. Models can be shared with different groups within a project such as with colleagues and partners that in turn for instance can view and inspect, comment, mark up and take measurements on the 3D models.

**Lagoa**

Lagoa is a cloud based web service that provides photoreal 3D rendering and a collaboration platform for the browser that enables an environment for designers, engineers, architects, advertisers, among others to collaborate and work together in real time to create 3D content. The resource heavy process of rendering is done remotely in the cloud removing the restrictions placed by the limitations of the local machine. The rendering is interactive and changes are seen instantly. The service is plugin free and no installation is needed to use the product other than a supported web browser.
Chapter 3

Scalability in web applications

This chapter will present theory that can be used to design and build scalable web systems, that has the ability to grow as the problem size increases, to accommodate for an increase in usage and load. The need for a web application to scale becomes apparent when the problem size changes to such a degree that the system can not perform and operate as intended. A system can perform well during a certain load but may be unable to sustain the same performance as the load increases. When the performance drops in a web application to the point where it can not serve clients without a noticeable latency users may become frustrated as web sites today are expected to be responsive without any delay. A scalable web application in contrast is able to be enlarged to fit the problem size, keeping its performance as intact as possible. A web application can consist of various components such as web servers, background processing servers, databases among others and any of these components may become the bottleneck of the system, and a subject to scale, when the load increases. Scalability in web applications is a wide subject that can include several topics. The theory here will cover different ways of scaling, some common techniques and building blocks used to assist in achieving scalability and a look at typical components used in web applications and how they can be scaled.

3.1 Scaling Vertically

Scaling vertically, also called scaling up, is one way of scaling and implies to maximize the resources on a single machine [25]. In a hardware context this means adding more power such as processing power, memory, storage, and similar. This is the opposite of scaling out explained in Section 3.2 that focuses on utilizing multiple separate machines. Scaling vertically is generally more usable for applications that are suitable or just more convenient to run on a single machine and it is an easy way to give a machine a boost in power to handle more demanding load [44]. For example some databases has traditionally had problems with running on more than one machine as things get more complicated when for example deciding how to share tables between machines in a good efficient manner. Scaling up such a database could be easier and more comfortable than scaling it out, especially if only a small boost is needed to handle a bit more load [36].

There are downsides to scaling up. One such downside is the likelihood of downtime in the system. If an important application service resides on a single machine it becomes more likely that downtime will occur in the system. For example if the machine hosting the service crashes or must be taken down intentionally to be able to upgrade it [25]. Another
downside is that it can be expensive as high-end equipment usually comes with a higher price tag. There comes a point where it can become very expensive to upgrade a machine beyond a certain level [44]. Though if only certain components need to be improved scaling up might be cheaper than scaling out. A big problem with scaling up is that there is a limit to how much a single machine can possess in processing power and memory. It becomes harder and harder to scale vertically when the equipment used already is among the best the market can offer [36].

3.2 Scaling Horizontally

Horizontal scaling, also called scaling out, usually refers to when you add more machines instead of improving the performance of one singular machine. This form of scaling can be useful in systems that can be divided into modules or layers. For example in a web architecture with different tiers such as a front end with web servers handling client requests and a back end with databases storing data. If the services residing on these different layers are designed so they can be hosted on multiple machines it is possible scale them both horizontally and each layer separately. If the web servers for instance are receiving heavy traffic it can be easy to just add another web server to add to the front end capacity and reduce the overall load on each server, or vice versa for the databases. Scaling out works well with the cloud architecture where resources can be obtained and released on demand making it possible to be efficient in resource utilization. This is especially useful if the demand of an application is irregular [44]. More information on this subject is given in Section 3.8

Some other benefits to scale horizontally not already mentioned are the reduced risk of a single point of failure and the lesser need for high-end equipment. A service that can be hosted on multiple machines has the opportunity to support a redundancy in the system if designed correctly. It is for example possible to construct a system that does not cease to function properly if one or more machines were to fail as long as enough machines still are available to operate the service [25]. Low end machines, that are less expensive than more powerful alternatives, also tend to be sufficient when scaling out as they share the overall load [44]. Scaling out also generally enables more computing power than what is possible when scaling up.

The disadvantages of horizontal scaling includes the need for an increased maintenance. Systems that contains a large number of machines may need more resources devoted into maintaining itself [25]. Another problem is that it might be hard in some cases to make a group of machines work together to appear as a singular machine. Very large systems that scale out may also run into obstacles such as electricity, space and cooling issues [36].

3.3 Load balancing

When scaling horizontally a common way to spread traffic among the machines is to load balance them with the use of load balancers. The purpose of load balancing is to distribute load to a set of nodes. The nodes, or servers/machines in a web context, are responsible for some sort of service and load balancers will relay requests to these servers. A load balancer may be seen as a manager that spreads the requests among the set servers it load balances. This enables multiple servers to be able to transparently service the same function, as the load balancer makes them appear as a single virtual server. Load balancers therefore play a crucial role in a distributed architecture as they help grant a system to scale by being able to add more servers to accommodate a higher load. To distribute the load to the servers
3.3. Load balancing

The load balancers may take advantage of different algorithms to decide which server to forward a request. The algorithms can distribute the load in different ways and some may be more suitable than others depending on the situation. More information on different load balancing algorithms will be given in Section 3.3.1. In a web service load balancers are commonly found in front of the web servers handling the client requests, but they may exist in more places than that in more complex systems [34]. A simple load balancing setup can be viewed in Figure 3.1. Here the load balancer is distributing the incoming client requests to three different servers providing the same functionality. Depending on the algorithm the load balancer is using it may take different decisions as to which servers it will place the load onto. The clients are unaware of which server is actually serving their request.

Load balancers products can be implemented as software or specialized hardware devices and may have other functionality other than providing load distribution. A load balancer may for example be able to monitor the health of the servers and remove unresponsive servers from the server pool that handles requests [34].

Figure 3.1: A simple load balancing example. A load balancer is distributing client requests to a set of servers.

3.3.1 Strategies

There are a few different techniques to distribute load. The following are some of the common approaches taken by load balancing products.

**Round Robin** Incoming requests are forwarded to servers in a rotational order. When the all servers have received one request each, the cycle begins anew with the first server again. In a setup with three servers s1, s2 and s3 the sequence order in which they may receive requests could look like: s1, s2, s3, s1, s2, s3, s1, ...

Round robin works best in a configuration where all the servers have similar capabilities and handles similar load, as the scheme itself does not for example take into account the current load or response time of the servers. If used in a configuration where the servers have uneven capabilities there may be scenarios where the round robin scheme forwards requests to weak servers close to being overloaded even though there are more powerful servers available that are also less loaded [40]. It is therefore also preferable if the requests have similar complexity as that could also factor in when some servers are put under a larger load compared to others [36].
Fastest Response Time  Directs requests to the server that responds the fastest of all the load balanced servers.

Least connections  New connections are passed to the server that maintains the least number of active connections. Machines that process connections fast will also be receiving the most connections over time and vice versa for slow machines, but this does not necessarily mean that resources are used optimally [42]. There is no consideration done for the actual processing capability of the servers, so machines that potentially could carry more load are not prioritized if other servers maintains less connections. This could be prevented to a certain degree by placing weights on servers, so that those with a higher degree of processing power are used more efficiently. The scheme adds the necessity for the load balancer to also in some way keep track and count the number of connections which adds to its complexity/overhead. Using this scheme helps with the potential problem of one server maintaining a number of longer active connections as other servers having less connections would service new connections first [36].

Predictive  This scheme normally combines a load balancing strategy with some other logic and aims to help prevent the issue of information staleness [42]. Staleness [16] may occur in a system when many transactions happens rapidly and decisions are made on "old" information. In such an environment servers that seem to be underutilized may in a very short time receive a large amount of requests and quickly become overloaded, because all work are directed to these servers until new information of the workload in the system is obtained. One example of a predictive method is the one used in BIG-IP system [18]. Servers are ranked according to the number of current connections and the rankings are also analysed over time to see trends in the performance, whether its improving or declining. Servers that possess an improvement in performance will receive more connections than ones that are declining.

Weighted  This technique can be applied to existing algorithms, such as round robin, to help with scenarios where there are servers in a system that have different capacity and can handle a different amount of traffic. By placing weights on more powerful servers will make them receive a higher dose of the load [25].

3.3.2  DNS load balancing

Load balancing using DNS is one of the easiest way to achieve load balancing [25] and is also an older type of load balancing technique developed before any dedicated load balancing devices were used [26]. DNS servers utilizes the round robin algorithm explained in Section 3.3.1 to distribute traffic. To better realize how this works and why it has drawbacks a brief understanding of the DNS system has to be established first.

DNS

Just as humans can be identified with various identifiers such as a personal name or a Social Security Number, a device connected to the internet can for example be identified by its hostname or its IP address. A hostname is for a human more readable and easier to remember than an numerical IP address which a machine can better understand. One of the main tasks for the Domain Name System (DNS) is to provide a mapping between hostnames and IP addresses, for example:
3.3. Load balancing

(hostname) www.example.com -> 203.136.53.205 (IP)

DNS is a hierarchical distributed system with a tree-like structure [28] as seen in Figure 3.2.

![Figure 3.2: DNS Tree.](image)

Each level in the tree contains a number of domains. At the very top of the DNS hierarchy are the root servers, and below lies the top-level domain servers (TLD) such as com, net, fr, etc. Another class of DNS servers are the authoritative servers. An authoritative server is responsible for a portion of a domain called a zone. Each zone has at least one server that is authoritative, meaning that it contains information about that specific zone and/or pointers to other authoritative servers of any sub domains [17]. Typically Internet service providers (ISPs) also provides a local DNS server. They are normally placed close to the client and acts as a proxy when forwarding queries [28]. In Figure 3.2 we can see that example.com is a sub domain to the top-level domain .com. Inside the example.com domain we have the two zones example.com and abc.example.com.

The DNS tree is used to make lookups on hostnames to get the corresponding IP address. Imagine the case where a client makes a request to www.example.com

1. The client sends the request to its local DNS server to translate www.example.com to an IP address.
2. The query is then forwarded by the local DNS server to the root.
3. The root finds the suffix .com and answers with the IP addresses for the top level domain server responsible for .com
4. The local DNS server asks the .com top level domain server for the IP address of www.example.com
5. The TLD server does not know the answer to the question but it knows of the authoritative server for example.com

6. The local DNS server receives the IP address of the authoritative server for example.com and resends it the query.

7. The authoritative DNS server for example.com knows the IP address of www.example.com and responds with the answer.

8. The local DNS server receives the answer and can now give the host the IP address of www.example.com

If every lookup of an IP address had to go through the root servers, they would be put under a heavy load. To prevent this some DNS servers can cache query results to its own local memory. A DNS server that has cached a mapping of a hostname to an IP address can reply that information when asked, even though that DNS server is not the authoritative server for the hostname. Hosts and mapping between hosts and IP addresses can change and are not permanent. Therefore DNS servers only keep the cached results for a set time before they are discarded.

DNS Round Robin

The DNS system enables the ability to perform load balancing by providing support for associating multiple IP addresses to a hostname instead of a one-to-one mapping previously explained e.g.

(hostname) www.example.com -> 203.136.53.205 (IP)
(hostname) www.example.com -> 203.136.53.206 (IP)
(hostname) www.example.com -> 203.136.53.207 (IP)

When a request to resolve a hostname arrives at the DNS server it will respond with the list of IP addresses associated with the hostname. The list is normally reordered in a round robin fashion for each request. When the client receives the list of IP addresses it usually picks the first IP address from the list. When a second client receives a response from the DNS server it will get the same IP addresses but ordered differently [27]. An example of this can be seen in Figure 3.3.

In the scenario where no caching is utilized the following happens. When client 1 asks for the IP of www.example.com the request goes through the DNS network until it arrives at the authoritative DNS server for www.example.com which in return responds with a list of IP addresses where the IP 203.136.53.205 is the first one. When client 2 asks for the IP address of www.example.com it will receive a list of IP addresses where the IP 203.136.53.206 is the first one. Thus the two clients has been directed to two different web servers hosting the same website www.example.com.

One of the issues with DNS round robin is caused by the DNS caching mechanism. Assume that the DNS servers starts utilizing caching in the example above. Client 1 asks for the IP of www.example.com and when the local DNS server receives the reply it caches the result www.example.com -> 203.136.53.205. When client 2 makes the same request as client 1 asking for the IP address of www.example.com the local DNS server already knows the mapping and can answers directly with the IP 203.136.53.205. Client 2 is now directed to the same server as client 1. This results in poor load balancing [7]. Another drawback with
the cache mechanism is that updating records may take a long time making removing and adding machines troublesome as cached entries may still lead to a machine that in reality is removed [25]. A DNS server also has no way of telling whether a server has crashed and has become unavailable, without using any extra external software, and will continue to distribute traffic to those servers. Finally the DNS load balancing can not determine the load of servers to make any intelligent decisions to balance the load. These issues makes DNS load balancing not optimal for load balancing servers, but it provides an easy and cheap way of distributing load. In some instances, for large sites, DNS load balancing can also be used as a way to distribute traffic on to multiple redundant dedicated hardware/software load balancers [39]. The limitations with DNS load balancing are addressed in dedicated hardware and software load balancers.

### 3.3.3 Dedicated load balancers

Load balancers can operate on different levels in the OSI layer and are typically divided into two separate groups depending on what level they operate on, namely layer 4 or layer 7 load balancers. A load balancer product may however include both layer 4 and 7 capabilities.

Layer 4 load balancers distributes requests to a group of servers based on the information found at the network and transport layer, such as IP, TCP, UDP protocols [19]. These load balancers commonly acts as a virtual server and distributes the incoming traffic to a pool of servers that provides the same service and performs the actual work. Layer 7 load balancers can perform application level load balancing by gathering information found in the application layer. This makes them able to distribute requests based on the content inside the request itself. Devices that directs requests to nodes based on layer 7 information can be said to perform layer 7 switching also called request switching, application switching, and content based routing. By performing layer 7 switching the load balancer can route
requests to specific servers based on the application data and policies. This enables the system to use servers that are specifically optimized to serve a certain type of content or functionality. Based on the request the layer 7 load balancer can forward the request to a server specifically optimised to handle it. Some content may have different requirements in resources needed, for example in processing power or I/O throughput. By grouping servers together that manage content they are optimised for improves the efficiency. One server group can for instance be tuned to serve images and another to execute a server side language such as PHP. The separation also allows for scaling each server group individually for better efficiency. In contrast, the layer 4 load balancer generally wants the cluster it load balances to serve the same content [33]. The data parsed and inspected at the application layer may for example be the URL, HTTP headers, cookies or data found in the application message itself. By operating on the application level allows for more features. Requests could for instance be directed to servers specifically implemented to handle premium users, to separate them from the general users [31]. Two of the layer 7 load balancing methods are URL switching and Cookie switching [20].

**URL switching** By looking at the URL the load balancer can take decisions on to what server group to direct the requests based on user defined configurations. It might look at certain strings in the address or at extensions such as .png to make the decision. URLs containing the string "/start" could for example be sent to one server group while URLs ending with ".png" could be directed to another server group.

**Cookie switching** Refers to the ability to look at cookie information to direct requests to a server group or a specific server. When a client makes its first request, the load balancer can return a cookie with embedded information about what server or server group should handle further requests. The client then stores the cookie locally and when making further requests the cookie information is being sent within that request. The load balancer can then make decisions based on the information it retrieves from the cookie on where to send the request.

With the additional features comes some drawbacks. Typically layer 7 load balancing requires more complex operations that can be more resource taxing on the load balancer and the overhead can also be larger compared to layer 4 load balancing [30] such as the additional parsing of data. Performing layer 7 load balancing generally requires the load balancer to maintain two TCP connections during a exchange between a client and a real server. First the client TCP connection needs to be terminated at the load balancer. As the data in a request will not be sent until after the TCP handshake has been performed, the load balancer can not read the needed information such as the HTTP header to make a load balancing decision until after the connection has been terminated. Secondly when the load balancer has made a load balancing decision a new connection has then to be established between the load balancer and the real server it has chosen. As layer 4 load balancing does not need the application layer data and already has the information needed to make a load balancing decision in the TCP handshake messages, it can forward these TCP handshake messages directly to the real server [32].

**Persistence**

The issue with a client being load balanced away to a different server during a session can complicate things, commonly when the session state is stored locally and not in a shared space like a database [24]. Storing the session locally on a server may work when there is
3.4 Caching

Caches are a fundamental engineering tool widely used in many places in the computing environment such as in hardware, operating systems and in various types of applications such as web browsers. A cache can be seen as a fast short term memory with a limited amount of space that utilizes the principle of locality making it contain data that is accessed frequently. In a web context caches can be used in different places, such as very close to the client or in front of web servers, to help improve the performance [34].

Before looking more into different types of caches, two types of content that are possible subjects for caching and are typically present in web applications will be explained.

3.4.1 Static Content

Static content is data that does not change and generally does not need any special treatment before being sent to a requesting client. A static file can for example be a .png file containing a site logo or a CSS file. As static files does not need to be processed before handed over to a client, there are some ways to improve the performance when they are being dealt with. There are for example servers that are optimized to handle static files and content delivery networks, see Section 3.4.4, can be used to place these files closer to the user base. Static files that needs to be processed before used by a client are harder to optimize. Files that needs to be restricted to certain users for example needs to be checked by the application before sending them out to clients [36].

3.4.2 Dynamic Content

Dynamic content are files that are being generated dynamically and are created in some way before sent to a requesting client. A web page file that needs information from a
database before being sent as a response to a request is an example of a dynamic file. Some dynamic files are easier to handle than others. Dynamic files for requests that are generic can for example be cached, but files that needs user specific content or real time information are more difficult to cache. Those files are less generic because they are specific to each individual or time sensitive. A simple approach to handle dynamic files can for example be to cache recently requested dynamic content by writing them to disk as static files. Before generating new dynamic files, the web server checks these files and uses them if possible. The issue with this approach is that the content being displayed may not always be the most updated content that currently exists [36].

3.4.3 Different cache architectures
Caches can be structured differently to fit a certain systems, depending on where and how the cache is supposed to be used.

Local cache
A cache can be placed locally on a server. When a requests is made to the server it can check its cache for the requested data and respond directly if the data is present. If the data does not exist in the cache the server will have to fetch it from the persistent storage where it is located. This method has problems when servers are scaled out. Each server can still use its own cache, but has no way of utilizing the caches placed on the other servers. Imagine three servers with local caches where server number three has data A in its cache. Two requests for data A arrives at server number one and two. As the servers can not fetch the data from each other both server one and two needs to fetch the data A from the persistent storage. This can be seen as if only a fraction of the actual cache capacity of all the servers is actual used. The data A will now also be on all three server caches resulting in redundancy as they store the same data [34].

Global cache
A global cache is a cache that is accessible for all the servers. Instead of checking a local cache all the servers are using the global cache. One problem with this architecture is that the global cache can become overloaded when many servers are utilizing it. On the other hand, because it is a single cache there are opportunities to make it fast by using specialized hardware which may make it fit in certain architectures [34].

Distributed cache
In a distributed cache architecture each server has a chunk of the whole data and together they logically form a combined cache. In contrast to the local approach, where each server has its own local cache, this can be made to scale well when more servers are added as all the servers are utilizing the same virtual pool of memory. For example, five servers with a local cache size of 100 MB each would make the total cache capacity 500 MB large [1]. To distribute the data to the caches a consistent hashing algorithm is normally used. That way a decision can quickly be taken when it needs to be made clear where to look for a specific data item in the distributed cache.

One of the hurdles a distributed caches has to deal with is how to handle if servers are disconnected from the others by crashing for instance. To not bother with the chunk of the cache that is lost at all is one way as the data is still obtainable from the original persistent
Another way is to store multiple copies of the data on several servers. This may minimize the data loss from the cache when a server is disconnected, as the data that was lost may still be present in another server that still is functioning, but adds more complexity [34].

### 3.4.4 Cache deployments

As explained in Section 3.4 caches can be used in many different places and may be used to fulfill different purposes. This is true also within a web environment. Caches are utilized in various locations ranging from being located close to the client machine to being located close to the back end service. An illustration of the different caches described in this section can be seen in Figure 3.4.

![Figure 3.4: Example of different caches and their locations.](image)

#### Browser caching

A browser cache can remember previously seen files, such as pictures on a site, by storing them on the client machine itself so they can be reused when repeated requests are made. Early on when internet connections was not as fast as they are today, this made a huge difference in performance. By having the files on the client machine the network connection was bypassed [36].

#### Web proxy caching

Instead of having the browser accessing the internet directly, requests are relayed through a proxy which is placed between the web browsers and the Internet. The proxy would therefore be the one to make connections over the internet instead of the clients themselves. The proxy can also cache content much in the same way as the browser. On a network
with many users a proxy helps decrease the bandwidth being used and overall increase the performance [36].

**Content Delivery Network**

A Content Delivery Network (CDN) is a network of servers, hosted by a third party, spread out in different key locations around the world serving cached content, from back end servers, to users. The idea is that users will be able to request data from a location that is closest to them reducing the time it takes to fetch files and therefore improving loading times in contrast to retrieving data from some server located far away. Other benefits are that the availability of content is increased as the CDN replicates pieces of data from the original web application making it hosted on multiple servers. The bandwidth is also decreased as the CDN provider serves some of the data, taking a part of the load away from the actual web servers hosting the web application. Typically static content such as Javascript files and images are hosted on content delivery networks as that type of content is easy to cache [25].

**Reverse proxy caching**

This type of proxy operates in between the clients and the front end web servers, and handles requests coming in from users. Reverse proxies can help offload the applications servers, letting them focus on their important tasks with running the actual web application. Reverse proxies will be examined more thoroughly in Section 3.5, while this section will focus on their cache abilities. One use of a reverse proxy is to let it cache content from the applications servers. The purpose of this is to make it able to handle requests from users without involving the actual application servers. The proxy acts on behalf of the web server by either returning directly if data is present in its cache or, if not, first fetch the data from the web server and then return a response as well as cache the data. The home page of a company web site for example may have a heavy usage of pictures and other types of multimedia files, that would be work intensive for the application servers to handle. By caching this data on the proxy it can serve the static content the majority of the time making the application servers handle less requests and thus having less work to deal with.

Reverse proxy servers that serves static content are often optimized to have better performance in serving the data very fast to the user. One problem might be to get the cache to posses the correct data. The application servers may not be able to tell if something went wrong on the proxy, like if incorrect, outdated or sensitive data was served from the cache. Because of this it is a good idea to only cache data that is assured to be safe and won’t produce a faulty site [36].

**Application Caching**

This type of caching can be said to operate between the front end web servers and the back end. Often the behaviour of this type of caching is implemented by the developers to fit the specific application [36]. Application caches are often used to store frequently accessed data from the database to minimize the database access or to store the results of expensive computations. A common cache system used for application caching is Memcached [1], which is of a distributed cache type explained in Section 3.4.3. A typical usage of application caching utilizing Memcached can be demonstrated in the following example.

Imagine a site that handles and displays text comments by users, where the comments are stored in a database. To dynamically create a completely new page as the content
changes could be a lot of work with querying the database to fetch the comments. Instead individual comments could be cached by Memcached. When a new comment is posted, the page displaying all the comments would have to be generated anew to include the new comment. As the previous comments already exists in the cache, only the new comment would need to be handled by the database. The new page is generated by fetching all comments from Memcached instead of going to the database. Imagine that only a few users are allowed to post new comments and that comments are added more infrequently, say once a day. An option could now be to cache the whole page to avoid both querying the database and to re-render the page for every request. In the first example where all users could make comments and comments are added more frequently it would be less effective to cache the whole page. The page would be invalidated quite often as it changes and would need to be re-created. Different cache behaviour is more or less appropriate for different types of web applications. With the right knowledge the performance can be improved significantly by tailoring the cache to fit the web application in question [36]. It could for example be a better idea to cache the rendered page in a reverse proxy to avoid requests to the web application.

3.5 Reverse proxy

As explained in Section 3.4.4 a reverse proxy can help optimizing the performance by acting as a cache in front of the web server. In addition to that a reverse proxy is typically able to perform other methods for improving the overall performance such as acting as a load balancer and performing various optimization techniques.

Other than providing a cache to offloading the number of requests to the application server the reverse proxy can also act as a web accelerator. A web application typically generates dynamic content and serves the content to requesting clients. Serving clients takes up resources on the server until the response has been completed. This makes it more costly to serve slow clients as they hold resources for a longer time than clients with fast connections. This is referred to as "spoon feeding". As application servers needs to generate dynamic content they generally need more CPU and RAM than static content servers that may simply interpret a request and handing back the requested content to the client. This may possibly make the application servers more expensive than their static server counterpart. It is therefore preferable if the application servers can focus on generating content and deliver the content fast, than spending time spoon feeding clients. With a reverse proxy in place the connection between the proxy and the application servers becomes local enabling better throughput and latency, making the application servers being able to focus more on generating content as the content can be delivered and stored more rapidly to the proxy cache. Instead the proxy cache becomes responsible for taking care of the task of delivering content to the clients acting as a buffer preventing slow clients from affecting the application server performance [42].

A similar technique to optimize the web application server is called collapsed forwarding. The idea is to collapse requests that are similar to each other together into one singular request and return the response to all the clients whose requests were collapsed. This means that if there is a high traffic to the web server and many clients are requesting the same data at the same time and that said data is not in cache, those requests can be merged together making it possible to only have to read the requested data from disk once [34]. The implementation of this could be done by only allowing one request to go through the proxy at a time. The rest of the requests will be waiting at the proxy, preventing them from overloading the system, until the response is arriving from the web server and they all will
be answered [37]. This type of situation can for example happen when a popular cached object expires and needs to be renewed, giving many close to simultaneous requests at once for that data [5].

3.6 Message/Task queues

A queue pattern can make the system operate more asynchronous and can be used to help improve the performance and availability of a system. Another property of the queue based pattern is that it provides the possibility for a separation between front-end and back-end services by making them more loosely coupled. Some use cases for a web application where a queue might be useful includes [46]:

- When tasks need to be performed that takes a longer time to complete.
- When tasks require a heavy usage of resources, i.e CPU intensive tasks.
- When tasks need access to an external service that may be unavailable.
- When a lot of tasks might be requested at once.

In a synchronous system where a client makes a requests and waits for the server to respond, long running tasks might quickly become a problem. When the server can handle the requests relatively fast in such a way that it does not receive more requests than it can service, the synchronous approach works good. If the server cannot do this, clients have to wait for other requests to finish before being serviced themselves. Increasing the work capacity by scaling out the servers and load balancing them might not be enough in some cases. Tasks with certain properties makes it complicated to distribute the load in a fair way.

A typical usage of the queue pattern is quite simple. When a client makes a requests to the server to perform some sort of task, the task is placed into a queue. The server can then return a response quickly to the client removing the need for the client to wait for the work to be done. Instead the server can let back-end workers fetch and process work from the queue and only let the client know that the requests has been received. To know when the task has finished the client might have some way of requesting the status of the tasks it has requested. An example of how the queue can be used is displayed in Figure 3.5

By adding a queue to the system a separation between the client request and the response can be built making the client able to operate asynchronously. Imagine for example a system where clients make requests for long running tasks to be performed. In an asynchronous system with a queue, the server can respond to the client immediately with an acknowledge after it has put the task into the queue. While the task is being processed the client can still utilize the web application. In a synchronous system, the same client would be blocked and forced to wait for the task to complete before receiving a response [34]. Compared with doing a single short task synchronously the total time for completing a task might be longer when using a queue as some overhead is added to the request-response cycle. The queue pattern have other advantages that might still make it the preferable option.

The queue can act as a backlog in the system. If some back-end service is not functioning, instead of returning an error to the client saying that the task could not be completed, the task is added to the queue and the client gets a message that the task is being worked on. The user might still be able to use the site and submit new tasks. When the back-end service is fully functional again, it can retry the tasks it might have failed on and deal with the tasks still stored in the queue. The system becomes more reliable.
Another benefit of the queue is its ability to level out the load on the system. When spikes happen in the traffic the system may become overburdened. One solution to handle spikes is to use a cloud platform, see Section 3.8, that provides auto scaling for adding more capacity to handle the increased load. But even when auto scaling the system, sudden bursts in the traffic might happen to fast for the scaling to react in time, or there might be a set limit on how far the scaling should go. By adding tasks into a queue, the front end web servers can be offloaded making them able to handle more requests, by letting back-end servers handle the heavier work. The back-end service can process the tasks from the queue in a tempo they can handle without being overloaded, while the queue is storing the incoming task requests. As the traffic load varies the depth of the queue may shrink or grow. The queue may in this case be seen as a buffer between front end web servers and back end servers.

The queue makes the front and back-end more loosely coupled, making them easier to scale independently. The front end can be scaled out if the system needs to handle more requests and the back-end can be scaled out if the queue for instance is growing too large [46]. More detailed information on different metrics that can be used when auto scaling the back-end in a cloud is given in Section 3.8

3.7 Scaling databases

The database is one of the harder components to scale horizontally in a web architecture, especially the traditional relational databases. Databases that aims to guarantee the ACID [2] properties are hard to scale on a large scale. Ensuring ACID on a single machine is a different tasks then enforcing it on distributed machines. To uphold ACID over multiple machines brings overhead costs and latency problems because of the protocols needed to be performed over the network, which leads to performance issues [42]. Another factor
that comes into play when working with distributed systems are the implications of the CAP theorem [3]. The CAP theorem states that Consistency, Availability and Partition-Tolerance cannot be achieved all at the same time in a distributed system. This means that only two of the three properties can be guaranteed which needs to be considered when designing the system. In many cases it is a matter of choosing the right tool for the job and often there is a trade-off that has to be made. For instance, a web based picture application may be fine with loosening up some of the ACID constraints to support eventual consistency and gain in performance, while a banking site needs the ACID properties to function properly but will have more overhead. The remainder of this section will cover some of the common approaches taken when scaling out a database.

### 3.7.1 Master-Slave

In a Master-Slave set up, see Figure 3.6, there is one master node with multiple slave nodes that replicates the master. The master handles all the writes from the application servers while the slaves are used for reads. Some form of load balancing or distribution mechanism is used to push the reads on to the slaves.

![Figure 3.6: Master-Slave database configuration.](image)

The slaves scale horizontally making the reads of the application scaled. The presence of multiple slaves can also increase the reliability and availability of the database. The master database is a single point of failure in this setup. If the master dies it might be possible to assign a new master from one of the slaves to continue operating the system, or just allowing reads to be performed while the master recovers.

One of the problems with this set up is the replication lag that may occur. Replicating data from the master to the slaves takes time and the delay may result in application servers reading outdated data from the slaves. Slaves may for instance have a different view of the data than what is actually on the master when a read is done quickly after a write has been done. Reading outdated data may or may not be tolerable depending on what the requirements are for the application. The Master-Slave set up is mostly suited for read intensive applications, as the writes do not scale as well as the reads that can be distributed across the slaves [8].
There exists two typical schemes to perform the replication between the master and the slaves. The first one works on the concept to replicate and execute the actual operation, that changed data on the master, on the slaves. The second approach is to log the changes performed on the master and transfer the changed data to the slaves. Depending on the operations performed the two approaches has both benefits and downsides. Operations that are computational expensive and may need to process a large amount of data, but still only result in a small data change like a single row, are more suited for the second approach. Sending the small amount of data that was changed on the master is preferable than to performing the actual compute expensive and time consuming operation on all the slaves once again. In contrast, if the operation is cheap but affects a huge amount of data it is preferable to distribute and perform the operation instead of transmitting a large amount of data to all the slaves. Other factors that may change which approach fits best are for example the network bandwidth and the capacity on the slave machines. The data transfer approach might for instance be more suited in a local area network than used for wide area replication. The duplicating operation approach instead might work better on a wide area network but operate worse on less powerful slave machines. Depending on the type of work that is typically done, what performance can be achieved, the costs of CPU, memory and network bandwidth one of the approaches might be more or less appropriate than the other.

### 3.7.2 Master-Master

In a Master-Master set up there are multiple master databases. Each master has the ability to write changes to the objects within the database, and each master is also responsible for propagating the changes to the other masters to ensure consistency. This differs from the Master-Slave set up that for instance does not need to put any effort in coordinating atomic transactions. This database configuration is generally not the recommended one for common use cases but may still be viable for some systems. It has the potential to provide better availability and fault tolerance with failover and failback scenarios, but trades in performance and does not scale in a true horizontal scaling fashion. Every master still needs to perform every write and as more masters are added, with replication and possibly a latency between them, it can quickly become complex and difficult to guarantee consistency between all the masters and the integrity of the data.

To coordinate transactions in a synchronous fashion between the master databases, the two-phase commit (2PC) protocol is commonly used to help ensure the ACID properties with data consistency. The 2PC protocol consists of two phases. In a simplified description of the 2PC protocol, the first phase also called the voting phase, consists of the master attempting a transaction by notifying all participating masters that it is about to commit. The nodes participating will then perform the necessary work to prepare the transaction and notify back whether they can commit or if they have to abort. In the second phase, the commit phase, the originating master decides to commit or abort based on the answers from the participating nodes. After the decision is made it will communicate the choice out to the other master nodes that will follow through with the needed actions to perform the commit or abort. The 2PC protocol is not fail proof to all failures and might even need to be halted if one node fails which requires resynchronization before continuing again. Also because of its overhead costs it is not very suited for wide area networks and does not scale well with multiple masters. It does however provide sufficient data consistency to be used for instance in banking and stock applications.

As with the Master-Slave set up, there are ways to loosen up the ACID properties to gain
in performance. One such way is to apply asynchronous Master-Master replication. When running asynchronously each master operates independently and performs local changes. The changes are stored, for instance in a queue, and later propagated out to the other database masters according to a specified schedule. This means that there is a period of time where the data can be different at the masters, before the changes have been propagated and all masters have attained data convergence. This also means that conflicts can occur. To resolve certain conflicts, resolution rules that defines how the conflicts should be solved can be used.

For example, in a scenario with two databases that both has updated the same row to different values will get a conflict when the changes are propagated. A simple resolution rule to solve this conflict may for instance use a priority method. By setting priorities to the databases such that a high priority takes precedence over a low priority makes it possible to prevent the conflict. The value chosen to be written will in this case be from the database with the higher priority of the two databases. Another way could be to use the latest time stamp, making the latest update be written.

The performance gained partly lies in the fact that the method uses less network bandwidth. It is more efficient to propagate multiple changes as a batch then to individually propagate each and every change separately as less overhead is used with fewer established connections needed. The bandwidth gain is especially important when performing on a wide area network, with masters separated far apart geographically [38].

### 3.7.3 Sharding

The sharding technique builds on a "Shared-Nothing" approach. The concept is to break a large database into smaller database pieces, called shards, and place the partitions on independent servers, each with their own CPU, disk, memory, etc. As more servers are added the system scales further as the workload can be distributed across the database servers reducing the load on any single server. Both reads and writes can be scaled in a sharded database system as each database is responsible for their shard and does not share it with another shard database. Each application server are preferably aware of the sharded databases system making them able to redirect their queries to the database that holds the data it wants to access [8].

Aside from improving the scalability having smaller databases also brings other advantages. In large databases that requires regular backups, database optimizations and similar routines, these types of tasks can be time consuming and difficult to accomplish. Sharding the database into smaller pieces enables each independent database to be managed separately, making each one of them easier to maintain than a single large database. The drawback of this is that there instead are more databases to maintain. Another benefit with having a small database is that they perform faster, for instance in search speed. By splitting up a large database a better memory to disk data ratio is also more easy to accomplish, making it possible to store a bigger portion of the data in memory or cache, reducing the disk I/O needs. Each database shard is therefore also more likely to be able to perform on less powerful/expensive server hardware as they do not have the same needs, as a single large database has, for a high end machine [14].

The difficulties with sharding includes how to partition the database such that the work on each shard becomes even. If the partition becomes uneven a database might for example receive a larger amount of queries then the rest, making it do more work, or vice versa. Another problem is that cross-table joins are hard to accomplish in a good way as these type of queries now may involve multiple database machines and therefore becomes slow to
perform [8]. Typically this would require the application part to get involved to process the data instead of the database.

3.7.4 NoSQL

NoSQL, sometimes referred to as Not Only SQL, is a database management system that takes a different approach compared to the relational one. In a relational database data is stored in relations organized in tables with columns and rows by a fixed schema that defines how the database is structured. NoSQL databases exists in different shapes that are implemented in different ways and vary in their set of features. They are likely to be less schema strict and does not fully rely upon relational associations between objects, which helps them achieve horizontal scaling more easily. Most of them use some type of key-value approach to support looking up data by a key. This may for instance take form in a simple mapping to data objects or to a document which in NoSQL terms is a more structured data object encoded by for example using JSON. A document may also provide support for queries on the structured data when retrieving it. They are often designed to scale in a horizontal fashion on commodity hardware by working in a cluster where machines can be added and removed to change the capacity. Another characteristic of NoSQL databases are that they often relaxes one or more of the ACID properties and trades off functionality to instead for instance favour better performance and flexibility [11].

3.8 Cloud computing

Cloud computing provides computer resources and services that can take form in a variety of shapes such as servers, storage space, databases and load balancing. Different cloud vendors may provide the services at different abstraction levels.

The most bare bone service model is *Infrastructure as a Service* (IaaS) which provides processing, storage and network capabilities to the consumer who is responsible for deploying software, like choosing the operating system, and installing any other application or tool that is needed. The cloud vendor takes care of the underlying infrastructure which the consumer has no control over. Another service model is *Platform as a Service* (PaaS) which is a slightly higher abstraction from IaaS. PaaS provides computing platforms which already includes an environment with for instance operating systems, programming language and different types of applications/tools ready to be used. The consumer has no control over the server, storage and network infrastructure but has control over and can decide which of the cloud provided computing platforms that should be deployed.

The pool of resources in a cloud are highly available over the network for the consumers and accessible on-demand. The resources can rapidly be elastically released or added to scale to accommodate the current demand on the system capabilities. Some cloud vendors can offer the functionality for automatic scaling to automate much of the scaling process. Often little effort is needed by the consumer who can by himself provision the resources with minimal interaction with the service provider. The pool of resources capabilities available for the consumer can commonly be seen as unlimited, ready to be used in any amount when needed. In a cloud environment the consumer most often does not have any knowledge of a precise location of the resources other then on a higher level of abstraction such as in what country, city or data-center. Usually a pay-per-use billing method is used to charge for the resources consumed by using some metering functionality [35]. Payment is often short term based i.e., pay per hour. Cloud computing can help with removing the need for an upfront commitment enabling consumers to begin small with low start up costs and grow
their system resources as needed [6]. The risk with a long term investment in purchasing physical hardware is eliminated when using the cloud to rent computer resources instead. The start up of an web service is made more convenient with the lesser need for planning on how much demand the application will receive and thus how many servers and other IT infrastructure needs to be purchased and set up to handle that demand. If the web service becomes unsuccessful there is no need handle hardware that is sitting idle and is no longer used. Compared to possessing physical hardware oneself, renting also removes the need for maintaining, configuring and operating the infrastructure, as the cloud provider is generally responsible for handling that. Renting more resources to scale up a system for instance is made fast compared to if the hardware instead would need to be bought and installed manually [45].

Cloud computing vendors can offer many of the services needed for scaling a web application such as load balancing and message queues described in Chapter 3.

3.8.1 Auto scaling

In a scalable web application new nodes are typically added when the load increases and grows too large by scaling horizontally or by making the already existing nodes more powerful by scaling vertically. When the load decreases the nodes are typically terminated in some way. In a cloud environment the process of scaling is possible to automate. New nodes are automatically added when needed to handle an increase in traffic, and removed when the traffic is decreasing to improve on the possible underutilization of servers and overall cost effectiveness of the system. Auto scaling is suitable for applications that can be scaled horizontally and has a varying usage. The work of managing the system resources also decreases as much of it becomes automated. Automation of the scaling process can be a help in situations where the load is hard to plan and predict precisely. Often user defined policies and conditions are used to determine when to scale. These conditions can be based on various metrics such as CPU or memory utilization. The automatic scaling can also be defined to follow a schedule.

Auto scale use-case

The Google Cloud Platform [21] gives an example of a flexible framework that scales up and down as the usage of an application increases or decreases. The approach exemplifies an automatic scaling of servers/virtual machine instances to serve different levels of demands in resources. It can be applied in general to applications that performs some sort of asynchronous computation such as video transcoding or similar.

Auto scaling helps with the problem of determining how much resources needs to be provided to be able to handle the user traffic on to the application. One way would be to estimate the peak demand and making sure that enough resources are allocated to handle such traffic. The problem with this is that the load on an application may often vary with time and when the traffic dies down much of the resources in form of servers/virtual machines will sit idle not serving any purpose and may still cost money to maintain. Another way would be to estimate the average demand. This would instead have the problem with not being able to handle peak or above average traffic, making the application slow or non functional for users. In both cases it is also difficult to do the actual estimation. It may for example be hard to predict the peak demand for a completely new application as there may be little history/data to make use of when doing the estimation.

With auto scaling, resources are added when the demand is increasing and removed when the demand is decreasing. When the load is high, auto scaling will help the system scale up
to be able to service the users. When the load is low, auto scaling will help the system scale down to create a better utilization of resources and thus lowering the cost of having a high percentage of non used resources. It can be hard to tune the auto scaling to scale perfectly, but even a simplistic method can help improve the system.

The framework example, seen in Figure 3.7, contains an orchestrator tool that takes care of the auto scaling. In regular intervals the orchestrator checks the status of all the back end workers that performs the computational tasks. The workers are the machine instances containing the logic to perform the application tasks and are a part of the application business that is being controlled by the orchestrator. With the help of some user specified criteria the orchestrator can scale the workers up and down depending on the load the system is experiencing. The front end is handling the client requests and puts tasks to be performed into the task queue from which the back end is fetching its work.

![Figure 3.7: Google auto scale framework example.](image)

When the orchestrator executes its periodic task of gathering data from the orchestrated application it makes a decision based upon the provided heuristics and the data collected. It can either decide to scale out, scale down or leave the system be and not scaling at all. If the orchestrator decides that the load is too high and that the system needs to scale out it may add one or more worker instances that will join the existing pool of workers. In a cloud environment these workers instances may be created from a snapshot to become replicates of each other, with the knowledge on how to join the service by themselves. The orchestrator may also decide that the load is too low and that the system needs to scale down. Instead of adding workers, one or more workers are closed and removed. As a worker may be busy with processing a task it may not be able to shut down immediately. The orchestrator may instead tell a worker to prepare to shut down which makes the worker stop fetching any new tasks from the queue. When the worker is finished processing the task it will announce a new status to tell that it is ready to shut itself down. When the orchestrator sees this it can in turn close and remove the worker instance. One problem with this is how to decide which worker to remove. The optimal worker to remove is in general the one that is ready
to be removed first as it will then be idle the least, but other factors may be for how long
the worker has been active as it may be the case that it is not worth for billing reasons to
close an instance that has only been active a short amount of time. If all the tasks takes an
equal amount of time to process, then the orchestrator may easier be able to estimate the
time of completion for each worker. In many cases tasks takes a different amount of time
and it is not always possible to make this estimation.

The criteria for when to scale up, down or not at all may vary depending on the applica-
tion. The orchestrator may for example monitor CPU load, memory usage, the current
number of tasks being processed or any other information it requires to decide whether
the application needs to be scaled. One heuristic trigger for scaling may for instance be
the maximum average memory load for all the workers and that the value cannot exceed
a specified limit before scaling up [21]. Another way may be to monitor how many items
there are in the queue, and try to scale so that there are for example no more than 10 tasks
in the queue. If the number of tasks exceeds that value more workers are added and when
the workers have caught up they are decreased in numbers [46]. What type of heuristics
are used in an application may depend on how predictable the tasks are i.e if they are of
similar complexity. The less predictable the tasks are the harder it is to orchestrate the ap-
plication. An example of an easier application to orchestrate would be where the processed
tasks takes similar time to finish and the CPU, memory and I/O usage is similar as there
are less variables that changes. In this case the resource usage is predictable which makes
it easier for the heuristic to be based on the resource bottleneck of the application. Tasks
are not always as homogeneous as described and may vary in CPU, memory and I/O usage.
They could also differ by external factors such as a user specified duration that defines
how long a task should execute. The heuristic could in this case be based on the largest
resource bottleneck that is observed on average. It can be beneficial to try to separate these
tasks into different groups of back ends, such that CPU intensive tasks would be processed
by one set and memory intensive tasks would be processed by another. To distribute these
heterogeneous tasks in this way would make make them more predictable.

A thing to consider is to what degree the user experience should be prioritized. If the
user experience is of a high priority an option could be to be more focused and generous on
scaling out than scaling down at the cost of having a more expensive system with a worse
resource utilization because of the possible over-provisioning of resources [21].

A more specific architecture, exemplifying a batch processing system, is given by Amazon
[4]. A batch processing system also processes tasks and therefore share similar traits to the
previous example. This system consist of a front end that act as a job manager that has the
purpose of accepting jobs, relaying information on the progress of the work and to provide
access to the completed result. The job manager utilizes a storage service to store any task
data that may for instance be needed for the actual processing of the task. A message queue
is used to hold tasks inserted by the job manager. A service consisting of auto scaled worker
nodes fetches tasks from the queue and processes them automatically. The workers may
store any intermediate results to the storage servers and also the completed result when the
task is done. Statistics and task progress information are stored in a database system. This
system can be viewed in Figure 3.8
Figure 3.8: AWS batch processing architecture example.
Chapter 4

Implementation

The prototype implemented in this project is a web application providing a service for managing simulations. The main functionality the web service supports can be summarized as:

- Support for handling users, with support for registration of new users and authentication for signing in and out of the service.
- Users can send simulation scenes to the web service to be simulated.
- Users can see an overview of submitted simulations. Users can view status and download data associated to a simulation.
- Users can view a 3D rendering of the simulated scene files.

4.1 Development tools

The server side is developed on Ubuntu using Python with the Django web framework, while the client side mainly is using Javascript and HTML. The main tools used in the project can be seen in Table 4.1.

<table>
<thead>
<tr>
<th>Server side web framework</th>
<th>Django (Python)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>MySQL</td>
</tr>
<tr>
<td>Task Queue</td>
<td>Celery (Python) with RabbitMQ as broker</td>
</tr>
<tr>
<td>Simulation software</td>
<td>AgX physic engine (C++)</td>
</tr>
<tr>
<td>Client rendering</td>
<td>WebGL using Three.js 3D library (Javascript)</td>
</tr>
<tr>
<td>Front-end framework</td>
<td>Twitter bootstrap</td>
</tr>
</tbody>
</table>

Table 4.1: Tools used for the development

4.2 Introduction

Based on the goals in Section 2.2 the following issues were identified for each requirement.

- **Authentication, User based system** The web service needs a way of storing users and authenticating them.
- **Submit simulations** Depending on the simulation scene to be simulated the work can vary in time (minutes, days, weeks). The web service, as a consequence, needs to be able to handle long running tasks.

- **Feedback on submitted simulations** The web service needs a way of fetching status information of simulations that has been submitted.

- **View simulations** Viewing simulations are also, similar to submitted simulations, tasks that can vary in how long time they take to complete. To view simulations that have been processed by the web service requires the client to be able visualize the simulations in the web browser. This includes communicating with a server side process, receiving transferred data it is being fed and render said data.

- **Easy access** The client side needs to use technologies that are supported by most common modern web browsers, available without any installation procedures.

In addition, the web site needs to have an user interface that can

- Present an overview of simulations that has been submitted.
- Present a view for sending simulations to the web service
- Present views for registering a user and to log in and log out a user.

The website has been built and formed by using Django. Django is a web framework for python and was, after a study of different web frameworks, mainly chosen because of its large community, maturity and wide scale usage, making it a reliable product with a lot of documentation available to quickly get started with the implementation. Using a web framework removes the need to build common functionality that is used in most web services. The Django web framework comes with a built in development web server that has been utilized, making the the project start up and configurations easier and more swift.

### 4.3 Web framework structure

Django, as most web-frameworks, follows the Model-View-Controller (MVC) design pattern but uses a slightly modified interpretation consisting of a Model-Template-View. The Controller in Django selects what view to be used for a given URL and is managed by the framework itself through its URL configuration module. The View in Django controls what information the user is presented with, and the Template describes how the user is presented with said information. The Model handles the access and communication with the underlying database. An overall architecture of Django can be seen in Figure 4.1 and the core parts will briefly be explained.

#### 4.3.1 URL Dispatcher

The URL Dispatcher redirects a request to a callable view function based on the URL it is fed. What view to call is specified in a URL configuration module called URLconf by Django. The URLconf module defines a mapping between URL:s and view functions. It has support for regular expressions to match the incoming URL:s against patterns and to capture values to pass as function arguments.
4.3.2 View

When the URL has been matched to a specific view by the URL Dispatcher, that view function is invoked. A View is a Python function that takes a request and is responsible for returning a response. It defines any business logic necessary to produce the response it wants to return. The view may communicate with the model layer to access data, call on external services or contain any other python code with arbitrary logic to create the response which can be anything from a HTML web page, an error with a 404 status code or a file. In most cases the response will first be processed by a template before returned.

4.3.3 Template

A Template takes care of presentation choices and defines how information should be displayed. In Django the document containing the information is normally a HTML file that defines how the response should be presented to the user by rendering HTML code. Django comes with a built in template language to design the information that has to be rendered. The template language contains functionality for simple logic to be utilized such as iteration and branching defined by template-tags, and place-holders to enable data to be passed on from the View to create dynamic web pages.
4.3.4 Model

The Model is a database layer for data access. Django comes with an Object-relational mapping with support for databases such as PostgreSQL, MySQL and SQLite.

4.4 User management

To solve the issue with handling users the authentication system that exists within Django has been utilized. This user system includes, but is not limited to, support for storing, authenticating and authorizing users, password management and a simplified usage of cookie-based user sessions.

To authenticate users means to verify that users are who they claim to be and is applied when a user signs in, while authorization controls what an authenticated user is allowed to do. Authenticated and anonymous users can be differentiated from each other and this is used to require anonymous users to sign in before allowed access to certain web pages. When a user signs in and logs out of the application the Django user system is also used to manage session with Django’s session framework. When a user signs in the ID of the user is saved in the session to keep the user signed in and when a user signs out the session data is instead removed.

4.5 Simulation management

The main feature of the prototype is the ability for a client to manage simulations through its web interface. The web service presents the options for a user to:

- Submit scene files that a client want to be simulated.
- View status and download data associated to a simulation.
- View a 3D rendering of the finished simulations.

The simulation of scenes and viewing of finished simulations can be categorized as long running tasks. Making the web servers, that hosts the web application and service clients, synchronously handle tasks that takes a long time to complete might not be the best solution as described in Section 3.6. Probably the most obvious one being that the client might be forced to wait for a response and feedback from the web server while the task is being processed. To address the problem with the long running tasks in this web service a task queue has been attached as an external service to the web framework. The task queue utilized in the prototype is Celery.

Celery

Celery [10] is a distributed task queue, see Section 3.6, built in python with support for being integrated with Django. In the context of this project Celery serves the purpose of

- Enabling a pattern for minimizing the request/response cycle between the client and web server. Instead of doing the task immediately on the web server and wait for it to complete, the task is instead scheduled to be done later.
- Preventing the system from being overloaded by providing functionality for limiting the number of tasks that can be executed at the same time on a single machine.
4.5. Simulation management

- Simplifying the management of tasks by providing features such as being able to terminate tasks, either waiting in the queue or currently being processed, and the ability to fetch the current status of a task.

Celery needs a broker to pass messages between the web server and the Celery workers. The broker chosen for this application is RabbitMQ [41] as it is the one recommended by Celery. RabbitMQ functions as an intermediary for messaging and gives a common platform to send and receive messages, providing the messages a safe place to live until received.

4.5.1 Submitting simulations

In this prototype, simulations are created from scene files. The scene file defines the environment the simulation takes place in, such as type and placement of objects. Events can be set to change properties in the scene at different time stamps to create a more dynamic simulation. The AgX simulation software can simulate the scene file and store the simulation data into a Journal file that can be loaded again and used to play back the simulation. The work flow from a client submitting a scene file to finished simulation can be explained as the following.

1. User selects a local scene file, specifies a duration for how long it should be simulated and submits the data to the web server.
2. The web server receives the data and writes the scene file to disk.
3. The web server enqueues a simulation task to the task queue.
4. The web server answers the client back that the submission was successfully received.
5. A worker will process the enqueued task when it is free from other work. The task tells the worker where to fetch the scene file, the duration for how long it should be simulated, to start the simulation and lastly where the results of the finished simulation should be stored.

This way the web server is mainly focusing on serving responses to clients and makes the workers asynchronously handle the long running tasks. Feedback on the simulation task submitted can be served to the client by letting the web server ask Celery for its status.

4.5.2 Viewing simulations

Scenes files that have been simulated successfully and stored to disk can be viewed by the client. The rendering is displayed inside the client web browser by using WebGL and rendering data is being transmitted to the client by the use of WebSockets.

WebGL

WebGL [23] is a graphics API developed for the web, based on OpenGL ES 2.0. It makes use of both Javascript for the API and GLSL (OpenGL Shading Language) for shader programs executed on the GPU. It is exposed through a HTML5 Canvas element making it integrated with Document Object Model (DOM) interfaces.

WebSockets

WebSocket [15] is an API and protocol providing a standard for allowing browsers to create a full-duplex, bi-directional communication channel over TCP with a server.
This makes it possible for messages to be sent by both the client and server asynchronously at any time while keeping the connection between them open. Commonly the web is based on a request response idea, where the client is mainly responsible for getting responses from the server in the form of user interaction or polling. Techniques such as long polling that fulfill the same purposes as WebSockets, in the sense of making it possible for the server to send data to the client when it think it is appropriate, have existed before WebSockets. One of the problems with these techniques though are their usage of HTTP which brings an overhead to the communication. That is one of the reasons why WebSockets are more suitable for low latency applications which is useful in this project to be able to render the simulations smoothly [47].

The AgX simulation software is not only used for simulating scene files as seen in Section 4.5.1 but also used when viewing simulations. In this situation it serves the purpose of:

- Reading the stored simulation file and its data.
- Acting as a server for issuing remote commands to control the playback, i.e, pausing and resuming a simulation.
- Acting as a web socket server to be able to open a persistent connection with a client for transmitting the simulation data.

When a client requests to view a specific simulation, the web server will enqueue a new task for this purpose. The task tells the worker, that eventually will fetch it, where to find the simulation data and to spawn an AgX simulation instance for playback of this data. As any worker may process the task and workers can possibly execute on different machines, the task also consists of writing the connection information of the AgX instance to the database. This way the client can read the address and port of the AgX instance and establish a connection.

The data being transmitted via WebSockets when viewing a simulation are mainly the transformation matrices for the objects in the scene. For each frame in the simulation the client will receive new transformation matrices that are applied to transform the objects rendered on the client. Scene information like the number of objects, the type of each object and the sizes of each object are sent once, before the simulation playback is initiated. Using WebGL the client sets up the scene by creating renderable objects from the initial information and from that point onwards only transformations are applied to the objects it has created. The prototype support the basic object shapes of boxes, spheres, cylinders and trimeshes.

Other data transmitted via the WebSocket connection is information about the simulation such as its current state (paused, running) and the current time of the simulation. To control the playback of the simulation the client sends commands to the AgX simulation instance that will react accordingly. If the command changed the simulation in some way, the new information and status of the simulation is sent to the client via the WebSocket connection and the GUI is updated to match the state of the simulation.

\section{Database}

For storing information a MySQL database has been used. Because of the Django ORM layer that lies on top of the actual database it is relatively easy to change the type of the
database to some of the other databases supported by Django. The database is utilized in
the following categories.

**Users** The Django authentication system uses the database to store user related information such as the user name, password and the date the account was created.

**Simulation and view tasks** The database is used to both keep track of the tasks created when submitting a simulation and the tasks created when viewing a simulation. The two types of tasks stores similar data but they also have some differences. The following information is stored in the database for each separate task:

- **User** A reference to the user who either submitted a simulation or requested to view a simulation.
- **Task name** A name of the task. Used in the client user interface.
- **Identifiers** Two different identifiers are kept track of. Tasks of submitted simulations are given unique identifiers that are used to create unique folder names for each such task, where related data is stored. Celery also has a unique identifier for each task used to make celery related calls in order to, for instance, fetch information about a task or send a signal to a worker that is processing a task.
- **State** The state of the task. See Section 4.8.3 for more information about the possible states a task can be in.
- **Creation date** The date the task was created. Used in the client user interface.

Data that is stored only for submitted simulations are the following:

- **Duration** For how long the scene file should be simulated.
- **Path information** Information about where to find the files related to the task such as the scene file that should be simulated.

Data that is stored only for the tasks of viewing a simulation are:

- **Connection information** The address and port to the AgX instance running to play the simulation.
- **Process ID** The PID of the AgX instance, used to close the process.

**Task queue** Celery uses the database to store results and information about the tasks. Celery supports multiple choices for a result store, but for this prototype the already in place MySQL database was chosen for convenience.

### 4.7 Ease of access

One of the requirements put on the prototype was for it to be free from any plugins and other software that requires installation procedures. This also implies that the web service prototype should be available in major web browsers to minimize the need of installing a specific web browser for using the prototype. Reaching this goal was mainly a concern for the client rendering of simulations where WebGL and WebSockets were used.

WebGL and WebSocket are both plugin-free technologies running nativity in the browser, supported in the majority of the prominent web browsers available. Because the technologies are relatively new they are for the most part only functioning in modern browsers and not in older versions. The compatibility [9] in some of the common browsers available today
can be seen in Table 4.2. In this table the partial support means that there still might be some problems and that all users of a specific browser version may not be able to use the technology. For WebSockets the partial support refers to the WebSockets implementation using an older version of the protocol and/or the implementation being disabled by default (due to security issues with the older protocol). The partial support for WebGL mainly refers to the level of degree of the additional requirement for users to have up to date video drivers, or to the level of success in test suites.

<table>
<thead>
<tr>
<th>Browser</th>
<th>IE</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Safari</th>
<th>Opera</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>9.0</td>
<td>10.0</td>
<td>11.0</td>
<td>26.0</td>
<td>27.0</td>
</tr>
<tr>
<td>WebSockets</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WebGL</td>
<td>No</td>
<td>No</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
</tr>
</tbody>
</table>

Table 4.2: WebGL and WebSockets compatibility for major web browsers.

### 4.8 User interface

This section will describe the different parts in the user interface and how the web application is used.

#### 4.8.1 User authentication

The web application requires a user to register an account and sign in with that account to be able to fully use all the services.

**Registration** To be able to use the core features of the web application, a user has to register an account. The account requires a valid user name and password. See Figure 4.2.

Figure 4.2: The register view. An new account requires a valid user name and password
Log in  After an account has been made the user must sign in to access the main services. A user name and password of an existing account needs to be entered to log in. See Figure 4.3.

### 4.8.2 Submitting simulation

When submitting a simulation for processing a .agx scene file has to be submitted together with a duration. See Figure 4.4.

- **.agx** The scene file that will be read and simulated by the web service.
- **Duration** Specifies the simulation time in seconds. The system will step forward until the specified duration time has been reached before stopping.

![Figure 4.3: The log in view. A user name and password of an existing account needs to be provided to sign in](image)

**4.8.3 View simulation status**

When a simulation has been submitted the current status of the simulation in question, and all previously submitted simulations, can be seen in a table together with information and various options. The table can be updated by using the **refresh** button. See Figure 4.5.

- **Id** An unique id for the simulation.
- **Submit Date** The date the simulation was submitted.
- **Name** The name of the scene file that was simulated.
- **Duration** The simulation time in seconds that was specified when the simulation was submitted.
- **Status** The current status of the simulation. The status may take one of the following states:
Figure 4.4: View for submitting a simulation for processing.

- **PENDING** Simulation is waiting for execution or unknown.
- **STARTED** Simulation has been started and is being processed.
- **SUCCESS** Simulation has been successfully executed.
- **FAILURE** Simulation execution resulted in failure.
- **RETRY** Simulation is being retried.
- **REVOKED** Simulation has been revoked.

A simulation with the status **SUCCESS** has additional options.

- **Delete** Brings up a pop-up with the option to delete the selected simulation.
- **Profile** Allows the user to download a profile covering the performance data of the selected simulation.
- **Journal** Allows the user to download the journal file containing the simulation data of the selected simulation.
- **View** Allows the user to play and render the selected simulation.

### 4.8.4 View processed simulation

A finished simulation can be viewed from its beginning to its end by rendering a 3D visualization in WebGL, See Figure 4.6. The user has the ability to control the camera and display more rendering information by using the controls described in Table 4.3.

The playback of the simulation is controlled from a menu, seen in Figure 4.7, with the following component description:

1. Move one frame backward.
2. Move one frame forward.
4.8. User interface

Figure 4.5: The status and various options of the simulations an user has submitted can be viewed in a table.

3. Start/stop the simulation playback.
4. Playback slider.
5. Enable/disable the playback to loop.
6. The playback time.
7. Enable/disable real time synchronization. When enabled the simulation stepping is synchronized to real clock time. When disabled the simulation stepping will run as fast as possible.
8. The rate at which the playback simulation data is being sent.
9. The real time ratio.
Key Description
Left mouse click Rotate camera
Right mouse click Pan camera
Middle mouse click / Scroll Zoom camera
z + {Left, Middle, Right} mouse click Rotate camera
c + {Left, Middle, Right} mouse click Pan camera
x + {Left, Middle, Right} mouse click Zoom camera
s Switch rendering information and fps counter on and off

Table 4.3: Key bindings when viewing a simulation

Figure 4.6: A processed simulation is being viewed.
Figure 4.7: The playback menu.
Chapter 5

Discussion

In this chapter some points will be discussed on scalability options for a production deployment of the web service prototype implemented, by drawing inspiration from the conducted theoretical study.

5.1 Scalability options

The prototype has been implemented to function in a development environment, running on a single machine. This section will discuss some of the options that can be considered when working on bringing the prototype to a production ready state.

The web application is currently served from the Django development server. This server is easy to get a project up and running because of its minimal configuration. The server is however not recommended to run in a production environment and is not built to handle much traffic. The Django application should instead be deployed on a server built and tested to function in production. There are several options for such a server, where Apache or Gunicorn are among the more popular choices combined with NGINX for serving the static files.

The biggest bottleneck in the system is the simulation processing together with the remote rendering. Each request to one such task requires an AgX simulation instance to execute. The process of simulating a scene is a resource heavy task. Only a few such task would be able to run simultaneously, depending on how powerful the underlying machine is, when compared to how many users the web server would be able to handle. With the task queue in place the system would not overload as the queue is buffering the simulation jobs and a predefined number of workers are executing them. The queue would however grow large if users would submit tasks in a much higher rate than workers would be able finish them. This in turn would result in a long waiting time for users before their submitted simulation tasks would be processed. Because Celery is a distributed task queue it is possible to deploy new worker instances on separate machines while still making sure that all the tasks are being managed correctly. This is useful as the number of workers can scale horizontally. Adding more machines to the system running Celery workers would increase the number of simulation tasks being executed at any given time.

Having the Django application server and the database reside on the same machine makes them compete for resources. By decoupling the different components in the system enables each of them to be scaled separately.
With the Django application and database separated, the Django application servers can be scaled horizontally if needed by deploying the same setup and code base on different machines with a load balancer in front. This would however require some design changes. On a single server setup user uploaded files are stored locally on the machine. This would become a problem with multiple Django application servers as, for instance, users can be load balanced to any such server meaning that the server might not have the file stored. A solution to this problem is to use a shared storage accessible from all the servers. Going any deeper on how a shared file storage works is however out of scope in this project. One thing to consider is that because simulation data files can be large in size it is preferable if the Celery worker instances are located close to the shared storage. The Celery workers need the simulation data files locally on their machines to perform their work and thus needs them transferred to their local disk.

Another part of the system that can be separated are the two types of tasks performed by the Celery workers. Dividing the tasks of simulating scenes and playing back simulations creates better options for customizability. More or less computing resources can be put into the different queues to scale them individually depending on their load. Taking it one step further, the tasks of simulating scenes could be divided into queues for short running jobs and long running jobs. Looking at the specified duration on how long the simulation should run would be one way of determining which queue the task should be put in.

The cloud architecture would fit well with this type of web application with the scalability options it offers, especially for the compute intensive tasks. How easy the prototype is to deploy on a cloud architecture depends on the cloud vendor. Deploying on a IaaS would meet the compatibility requirements of the prototype but also require to handle much of the system management by yourself. Deploying on a PaaS would possibly require the prototype to be changed to fit the services provided by the cloud vendor, but with the benefit of requiring less management of the underlying hardware.

An example of a possible deployment of the web application can be seen in Figure 5.1. Using Amazon Web Services [43] as an use case, the Django instances could be deployed using the Elastic Bean Stalk platform for automatic management with load balancing and scaling or using EC2 instances manually together with Elastic Load Balancing. The simulation data could be stored using S3 for a shared storage place. The Celery module could also be deployed on EC2 instances using the auto scaling functionality. Using Amazon Cloud Front the large journal files containing the simulation data could be distributed out through the Amazon CloudFront network to offload the web service and provide lower latencies and better transfer speeds to clients.

To further scale this system it would be possible to add and utilize more of the techniques discussed in Section 3, but also things not covered/out of scope of this project. A cache layer can for example be added to minimize the number of queries to the database. But it would be unnecessary to scale if there is no need for it. It is therefore important to monitor the application to be able to figure out the bottlenecks and where the application should be scaled to operate as desired.

5.2 Conclusions

All the basic goals and functionality requirements initially set for the web service prototype have been met.

- The prototype has a way to handle users and provides functionality for registering users and signing them in and out.
5.3 Limitations

Figure 5.1: Example on a deployment of the implemented web service.

- Users can submit scene files that the web service will process into finished simulations.
- Users will be presented with status information about submitted simulations in a table.
- Finished simulations can be rendered on the client machine by communicating with a back-end process that executes the AgX simulation software and sends relevant simulation data to the client.
- The service is accessible through major web browsers and is free from any browser plugins and installation procedures.

The prototype is however not fully designed and built to work in a production environment as it was more built towards being able to demonstrate a concept. Therefore when implementing the prototype to fulfil the goals, less consideration were put on performance, security and fault tolerance. Even so, for displaying a concept the prototype works well enough.

5.3 Limitations

Early on in the project there were thoughts on making clients be able to connect and view a simulation while it still is being processed, and to view plot data of finished simulations. The two ideas were ultimately not included as goals of the thesis project and only to be implemented if time existed. These functionalities has not been implemented in the prototype for this project as there was not enough time.

Currently the client can not render all the shapes of the AgX simulation software. This makes scenes that contains objects that are not supported by the web application not rendered as intended, as the client wont render shapes it does not understand. The supported shapes in the web application are boxes, spheres, cylinders and trimeshes.
5.4 Future work

Before deploying the prototype into a production environment there are many aspects that has to be looked into. One has to examine possible security holes that a mischievous client may abuse. Reinforcing the application to be more fault tolerant, making sure it does not break if any part of the system unexpectedly crashes. Some of the current implementation designs also needs to change if the application should need to be scaled onto multiple machines as discussed in Section 5.1.

The client rendering can expand its support for the number of objects it is able to render. This would make the application able to handle more scenes and attain a better compatibility with the AgX simulation software.

The application can be built to let users connect to a running simulation that is being processed. This would remove the need for users to have to wait for the whole simulation to be completed before being able to view it. Another interesting thing would be to make users able to create simulations on the fly by being able to interact with the scene.

Implementing a way for a user to not only see a visual representation, but also to view a plot of the data, of a simulation would be useful to be able to evaluate the simulation data in more ways.

When streaming data to the client it would be beneficial for the client to be able to buffer the streamed data. This way a slow client has a chance to load the data before playing the simulation. It also opens up a possibility for the server side to release the process streaming data to the client, when the client has received and buffered all the streamed data it needs to render the simulation.
Chapter 6

Acknowledgements

I would like to thank my supervisor at Umeå University, Jan Erik Moström, for helping me out with the report and giving me useful advices. I would also like to acknowledge Algoryx Simulations AB for offering me this thesis project, helping me get started and providing a place for me to work at their office. I would also like to thank all the people working at Algoryx for being supportive. A special thanks goes to my external supervisor Nils Hjelte for helping me out with the implementation, explaining parts of the AgX simulation software and guiding me in this project by taking time to answer my questions.
References


