Faculty of Science and Technology
Department of Computer Science

Giga-View

A distributed gigapixel image viewer controlled by mobile devices

—

Einar Kristoffersen

INF-3981 Master thesis in Computer Science, June 2015

Abstract

About 10 years ago the Tromsø Display Wall, a wall size tiled display containing multiple displays and computers, was built. During the last years, a gigapixel image viewer has been implemented and deployed on this wall. By tiling small image fragments, the viewer is capable of displaying images with the resolution of gigapixels.

Now, Tromsø Museum is deploying a new display wall and will be needing a new gigapixel image viewer. As the current viewer is hard to maintain and doesn't meet the desired usages or requirements of the museum, the need of a new design is growing.

This thesis presents Giga-View, a gigapixel image viewer controllable from mobile devices. It's design makes interaction available through a web browser and requires no software installation or high processing power from the device being used. Fulfilling the goal of high throughput and low latency, Giga-View is capable of processing an input stream with the frame rate of 60 frames per second, without any noticeable delay being built up. The amount of data being processed each second corresponds to 33600 image fragments, translated into 520 MB of data. This result was achieved by the use of various caching techniques, which combined, increase performance and make the application very efficient.

Acknowledgements

First, I would like to thank my advisor, John Markus Bjørndalen for providing great guidance, feedback and motivation throughout this project. I would also like to thank my co-advisor, Otto Anshus for giving me inspiration and useful insights.

A thank goes to the members of the HPDS group, as they have been of great help with technical issues and providing information about the instructive work environment.

I would like to thank my fellow students for being helpful in technical discussions, feedback and support through five years of study at the university. You will not be easily forgotten and I hope some day we could work as colleagues.

I would also like to thank my friends and family for their support through my whole education, reminding me of the importance of taking a break from time to time.

Finally, a special thank goes to my girlfriend, Therese Aune, who always gives me motivation, support and love. She makes me happy and keeps me going, even in hard and stressful times. For this, I am forever grateful.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
GB	gigabyte
GPU	Graphics Processing Unit
HPDS	High Performance Distributed Systems
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
KB	kilobyte
LRU	Least Recently Used
MB	megabyte
NFS	Network File System
OpenGL	Open Graphics Library
PB	petabyte
SDL	Simple DirectMedia Layer
TCP	Transmission Control Protocol
UiT	University of Tromsø
VSync	Vertical Synchronization

1 Introduction

In the later years, the HPDS group at the University of Tromsø has developed a distributed gigapixel image viewer, running on the Tromsø display wall. The image viewer is a software component distributed over the display tiles in the display wall, where each of the tiles puts together image fragments to form a larger image. Together, the fragments form an image with the resolution of gigapixels.

During 2015 a new display wall will be installed at the Tromsø Museum. The current image viewer works in the Tromsø Display Wall, but is not flexible enough for future use in the display wall at Tromsø Museum. It does not support on the fly reconfiguring or on the fly switching of images. Also, it does not work with the Display Cloud[2] infrastructure intended to be used at the museum.

As the current implementation of the image viewer is hard to maintain and does not meet the desired needs of the museum, the need for a new design is growing.

To integrate better with Display Cloud and provide a more flexible solution, such as on the fly reconfiguration and switching of images, we expect that a new implementation based on a concurrent design will work better.

As the museum will not have a functional gesture system in the beginning, it's desirable that the image viewer should be controlled by mobile devices. For certain uses of the Museum wall, it would be interesting to control applications on the wall using mobile devices instead of using a gesture system. To support this, we need to allow multiple devices and multiple types of interaction systems control the viewer.

This thesis presents Giga-View, a distributed gigapixel image viewer controlled by mobile devices. It builds upon the capstone project "Concurrency in a Gigapixel Image Viewer" conducted autumn 2014 by the same author. This technical report describes the architecture, design and implementation of Giga-View.

1.1 Problem Definition

The problem definition for this thesis is as follows:

> *Develop an architecture and protocols that let mobile devices, such as phones and tablets, control the gigapixel image viewer. The system should be flexible enough that other interaction systems could be plugged in, such as gesture interfaces, in the future. During the project, concurrent design patterns for a distributed gigapixel image viewer should be investigated and evaluated. The student shall implement a working prototype based on a chosen design pattern, which will be evaluated through experiments on the Tromsø Display Wall.*

The interpretation of the problem definition leads to the task of creating an architecture and design of an gigapixel image viewer based on the principles of concurrency. As the users of image viewer most likely will use different mobile devices, a desired need is to create a control interface, such as a web interface, that do not require any software installation on the users' devices.

A problem with the image viewer developed in the capstone project was that it created a certain delay in time from when the user interacted with the system to the image was displayed on the wall. As the users of this image viewer will be using mobile devices connected through a wireless network, the importance of achieving low latency grows in order to improve user experience. Also, the greatest bottleneck from the last prototype was the time used on fetching images. Therefore, during this project, various caching techniques will be used in order to improve the fetch time.

The experiments conducted on the image viewer should determine how well the system performs and whether or not there still are any bottlenecks remaining or caused by the new design.

1.2 Giga-View

By developing Giga-View, we aim to create a image viewer that fulfils the needs of Tromsø Museum and being more efficient than the one from the capstone project. Giga-View should be more flexible and applicable as is can be controlled by mobile devices as well as stationary computers. The web interface could in theory be accessed by any device with a web browser, but not all devices or web browsers will be supported in this first prototype.

As we want the Giga-View to be used at the Tromsø display wall as well as the one at the museum, the users of this application might have the need of changing the frame rate to a value that fits their current run. Therefore, it is desirable to develop an image viewer capable of delivering a frame rate up to 60 frames per second. A frame rate this high creates a very high throughput, which makes it even more important to keep a low latency, as a delay could be build up quickly when processing great amounts of data.

Creating a design for accessing images fast and on-demand where data is written once and read often is of high prior in this project. Therefore, various caching techniques will be applied to the system in order to achieve this goal. The caching techniques claiming our main focus will be prefetching and cache abstractions based of the idea of object storage, where raw pixel data will be stored as objects.

Giga-View will be distributed over the display tiles in the display wall it runs on. The architecture of the display wall, whether it is the Tromsø display wall or the one at the Tromsø Museum, is already build. The image viewer will be developed with regards to fault tolerance based on the underlying architecture in the display wall.

1.3 Contributions

The contributions produced in this project are:

- A concurrent design pattern for a gigapixel image viewer
- Implemented and working prototype based on the chosen design pattern
- A web interface making it possible to control the image viewer from mobile devices like phones and tablets
- Evaluation of the image viewer via documented experiments and mea-

surements

- Experiences gained from the project

1.4 Project limitations

Before starting this project we were facing a set of factors limiting our project and implementation. As the time frame for this project was limited, there was no focus on the security aspect of the system. Any potential security threats has not been treated or evaluated, nor has the implemented prototype been improved to handle security risks that might exist.

During the project, the prototype has been developed on the Tromsø Display Wall, as we did not have access to the display wall at Tromsø Museum. Therefore, the design and implementation has been limited by the architecture of this display wall.

The prototype is developed in such a way that mobile devices can control it, but so far the prototype has only been tested with phones and laptops. It does not support tablets, as such a device was not available under the project. Still, the design and implementation makes it fairly easy to add this kind of support later.

The implementation is currently not integrated with Display Cloud. Neither are on the fly reconfiguration or on the fly switching of images supported, but the design of the image viewer has been made in such a way that it is possible to integrate with these features later.

1.5 Outline

The outline for the rest of this thesis is as follows. Chapter 2 gives an insight in work related to this project. The chosen architecture and design pattern is outlined in chapter 3. Chapter 4 describes the implementation of the chosen design. Giga-View is evaluated in chapter 5 with a series of experiments and measurements, followed by a discussion in chapter 6. The conclusion, concluding remarks and future work are given in chapter 7.

2 Related Work

In this chapter we will go through some related work having great influence on this thesis. The work presented in this chapter is either related to or has some similarities to this project.

2.1 The Tromsø Display Wall

In 2004-2005, a group of people at University of Tromsø (UiT) started working on the Tromsø Display Wall, a wall-sized tiled display consisting of multiple smaller displays and computers[1].

The display wall is running on a cluster of 28 computers using NFS, where each computer is used to drive its own projector. The projectors are organized in a 7x4 grid forming a 22-megapixel display.

The size of this wall gives a high resolution, allowing the users to see the data in its entirety from a distance and gives the users closer to the display a more detailed view.

The wall also consists of a set of cameras making interaction possible by movements. The system determines a 3D location of an object by recognizing hands and fingers. No markers is required to do this and the system enables touch-free interactions with the wall. It also supports interaction by the use of a mouse, keyboard and devices, like tablets.

2.2 Concurrency in a Distributed Gigapixel Image Viewer

This capstone project was conducted autumn 2014 by the same author, with the goal of investigate and evaluate a concurrent design pattern for a distributed gigapixel image viewer. During the project, a working prototype was implemented based on the chosen design and it was evaluated through experiments on the Tromsø Display Wall.

The contributions from the project were a concurrent design pattern for a distributed gigapixel image viewer, a working prototype, documented experiments and measurements, experiences gained and lessons learned.

The results from this project did not fulfil the expectations gained in advance. The cache was not properly implemented and as a result, the cache hit ratio didn't reach more than 60-80% depending on the different datasets and cache sizes. The prototype was constructed to kill fetch requests using too much time and with a 15% kill ratio, not all image fragments meant to be rendered did come to the rendering process. The measurements indicated that fetching the image fragments was the main bottleneck in the whole pipeline. The fetch time was highly affected by the hit percentage in the cache and thereby, every miss were contributing to increase the kill percentage.

This project was wrapped up by mentioning a few further work suggestions, such as changing the graphical framework, change design of the cache and implement other caching techniques like prefetching. On the fly reconfiguring of the image viewer was also a suggestion of improvement, making the client able to switch between images to view without restarting the image viewer.

2.3 Gigapixel image viewers

For the moment, there exists a number of applications and frameworks for viewing and processing gigapixel images[7, 8, 9, 10, 11]. A lot of these systems are using a common strategy of image processing in order to display them, by putting together smaller image tiles to form an image of very high resolution. Panning and zooming is often based on the image viewer to exchange image tiles inside a bounding box as the image is moved or zoomed into. Interaction with the various applications differs, as their overall design and run environment is not the same, but interaction by gestures, touch surfaces, mouse and keyboard are the most usual to find.

The usage area of such systems depends on their underlying architecture. By running a gigapixel image viewer on a massively tiled display wall, the system can be used for research, e.g. microbiology, where an image from a microscope can be displayed over a great surface. This could make it easier for scientists to work together, as they all can look at the same image from a distance instead of looking down in each their microscope.

Giga-View is strongly related to these projects, as the architecture of the gigapixel images has great similarities and makes the applications to use the same tiling strategy when displaying an image. However, the use area is not the same. Some of these projects are created with the purpose of displaying gigapixel images on massively large displays, beyond the size of the Tromsø Display Wall. These viewers are used for researching huge data sets[14], whereas Giga-View is developed to be a part of the Tromsø Museum's exhibits in accordance with their desired needs.

2.4 Web map services

Google Maps is a web map service that lets you display map images from all over the world in your web browser. You can watch the earth from a distance and zoom in on desired locations. When using Google Maps in your browser, the service will give you a set of image tiles where the amount of tiles will vary as the browser window and screen size will form a bounding box. There are different kinds of tiles, like satellite, map and overlay tiles and Google Maps are made of dozens of thousands of these[3]. E.g. a map tile gives you an image of a specific location and an overlay tile (partly transparent) gives you the name of the country, city or street name. Google Maps uses three coordinates to determine a specified location. This is a x, y and z value and the Google Maps Application Programming Interface (API) performs an HTTP request using a combination of these to load images into the client's bounding box[4].

This work is very similar to the tiling of image fragments in Giga-View. The web interface also uses the device's screen size to form a bounding box for the image being viewed. Image fragments from the same, requested zoom level are tiled by Giga-View into a greater image, which can be moved. As the image is dragged around, new image fragments are tiled inside the bounding box.

There has been some other work on improving the performance and interactivity of geographic map servers by building a caching and tiling map server[5]. As the Google Map API gives the browser a bounding box for the tiles, the tile server will fill it with cached tiles on a request from the API. The cache server requests the map tiles at all zoom levels from the Map servers by the use of adapters. There are different adapters for each type of map server. This work was done with a small amount of map tiles and in order for the system to efficiently support more map coverage, the developers must investigate the performance options of the naming scheme.

This work has inspired the author with the idea of using cache servers to retrieve image fragments to the viewers.

2.5 Facebook photo caching

In 2010 Facebook's users had uploaded over 65 billion photos, making it the world's biggest photo sharing website[16]. For each uploaded photo, Facebook creates four images in different sizes, storing around 260 billion images translated to more than 20 petabyte (PB) of data. As users upload more than 1 billion photos a week, the amount of stored data grows quickly. Still, the site is capable of serving one million photos per second at peak.

Haystack is an object storage system optimized for Facebook's Photos application and is "protected" against millions of photo requests by the above cache layers consisting of a browser cache, the Edge Cache and the Origin Cache[17].

The client's browser is Facebook's first cache layer, using an in-memory hash table to test existence in the cache, storing objects on disk. It uses the Least Recently Used (LRU) eviction algorithm and serves 65.5% of all the photo requests.

The edge has an in-memory hash table holding metadata about stored photos. It also contain large amounts of flash memory to store the actual photos. If a cache request hits, the photo is retrieved from the flash and returned to the client browser. On a miss, the photo is fetched from the Origin cache and inserted into the edge cache.

Hash mapping routes requests from the edge cache to the origin cache based on the unique id of the requested photo. The origin cache also has an in-memory hash table holding metadata about photos stored and large amount of flash memory for storing the actual photos.

Facebook's backend, Haystack, store photos at store machines. A store machine represents a physical volume as a large file consisting of a superblock followed by a sequence of needles, where each needle represents a photo.

From the description of this paper, it is made clear that Facebook's photos is accessed in the following way. It is written once, read often, never modified and rarely deleted. Data is organized on the store machines and cache layers based on access type, e.g. write request goes to the store machines and read requests are handled in the upper cache layers. As read requests occurs often the data is placed close to user. Write requests must be addressed to store machines that is not full yet.

The author of this thesis was inspired by this paper to create cache levels based on different needs regarding the movement of the image.

3 Architecture and Design

The architecture and design created during this project is based on the design of the image viewer prototype developed through the related capstone project described in section 2.2. The contributions from the capstone project was used to create a design more efficient and user friendly by the development of a web interface to control the image viewer and implementation of the project's future work suggestions.

The gigapixel images compatible with this image viewer are build up by a hierarchy of directories where a root directory contains one subdirectory for each zoom level. A zoom level directory contains a set of image fragments, which put together form the whole image or the global image at this particular zoom level. When the zoom level increases by one, the global image size is doubled and the number of image fragments are increased respectively. The gigapixel image can therefore contain a number of image fragments per zoom level, from one image fragment at the first level to hundreds of thousands at the last.

Before creating the design, there were some issues we had to think about. As a gigapixel image can contain a huge amount of image fragments, each display node might not have enough memory to cache the whole image and if they did, it might probably take a while before all nodes finish loading the whole gigapixel image. Therefore, a different design pattern had to be made where each display node had to limit the number of image fragments held in its cache.

3.1 Architecture

The image viewer and its web interface is designed using the following components: A web server for hosting the interface, a coordinator, a set of viewers and a cache.

The involved components pass data to each other in a way that reminds of the the pipelined architecture used in the earlier capstone project. If we look away from the web interface, one important difference between the systems is that we have moved away from the idea of one way communication and towards the concept of two way communication between the coordinator and the viewers. The data is transferred from one component to the next starting with the client interacting with the system through the web interface and ends when the image is displayed by a display tile. We will now give a brief overview to explain each of the involved components that together makes the web interface and the improved image viewer.

First, a web server is hosting the interface accessible through a web browser on mobile devices as well as stationary computers. For the client, this interface is mainly about having the opportunity to interact with the image viewer, but for the rest of the system, it's about passing data further to the next component, the coordinator. The data is passed in the form of states, where a state is the position in pixels to the image being viewed.

The coordinator's task is to listen for states sent through the interface and then forward them to the participating viewers. It also has the responsibility for synchronizing the viewers and detecting the need of prefetching image fragments into the cache.

The viewers are computer nodes corresponding to a chosen number of display tiles in the cluster. Image fragments located on the shared disk space Network File System (NFS) are fetched by the viewers into the cache and drawn onto their display, hence the name viewer. An image fragment is located by the use of a x, y and z coordinate, which put together corresponds to the file name of the fragment. The x and y values are the image coordinate in a 2 dimensional space when the fragments are organized in a grid where coordinate (0, 0) starts in the upper left corner. The z variable corresponds to the zoom level. We will talk more about the cache, including cache layers in section 3.5 and 4.4.

The design is build upon the idea of a master-slave architecture, where the coordinator is the master and each viewer is a slave. We would like to point out that without synchronizing the viewers, the frames will be displayed at random times at each viewer. The viewers might or might not view frames at

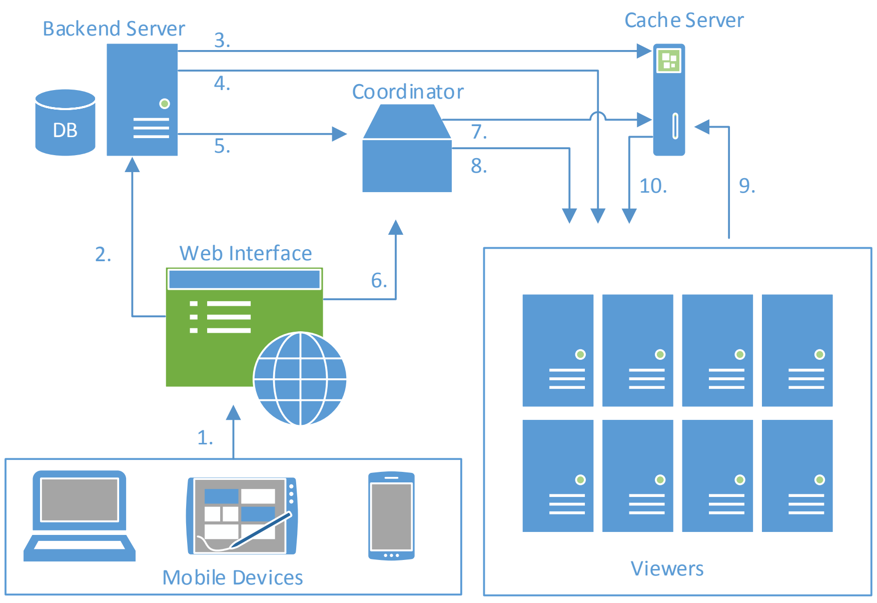

Figure 3.1: This figure shows the conceptual architecture of the system. 1. the client interacts with the system through the web interface on a mobile device. 2. the interface sends an initial request to the backend server, which starts the cache server (3), the requested viewers (4) and the coordinator (5). 6. the web interface produces a stream of image states to to coordinator. 7. The coordinator sends a prefetch request to the cache server. 8. The coordinator forwards the received states to the viewers. 9. the viewers requests image fragments from the centralized cache server and 10. the image data is sent in return.

the same time, but such an important detail should not be left up to chance. By synchronizing the viewers, the chance of displaying the frames at approximately the same time is highly increased. A master-save architecture seems appropriate to such an approach, as the coordinator is capable of synchronizing the viewers and control the data flow to them.

3.2 Web interface

The web interface makes it possible to interact with the image viewer by moving it around and zooming in and out on it. The interface is accessed via the device's web browser and is hosted by the rocks cluster at the Tromsø Display Wall. It lets the client select which image to display and what tiles in the display wall to use.

As the client might want to use the same image and display tiles later or often switch images and display tiles, the chosen image and display tiles are saved in a configuration. In this way, a client can easily switch between run configurations to use different images and/or display tiles. All data about images and run configurations is stored in a database located at the web server.

The selected image is displayed at the client as well as the display wall to make navigation of the image more practical for the client. Each time the client moves the image or zooms in or out on it, the image position containing the x, y and z coordinates is updated. This position is periodically packed into a state and sent to the coordinator.

3.3 The Coordinator

The coordinator is designed to do three main tasks:

1. Forward states to the participating viewers.
2. Send prefetch requests to the cache.
3. Synchronize the viewers in order to keep them from displaying parts of the same frame at different times.

The coordinator is constantly listening for states sent from the client and when it receives any, it will forward them to all participating viewers by the use of a broadcast. The states are received and broadcast concurrently to speed up the process. Somewhere between receiving a state and broadcasting it, the coordinator will determine the need for prefetching the images requested by this state into the cache. If it is necessary to prefetch the any image fragments caused by the change of state, a prefetch request is sent to the cache where it will be fetched from disk.

The coordinator is also responsible for synchronizing the viewers. This have to be done because the viewers will not necessarily use the same amount of time to execute each request. Synchronizing in this context means to let the viewers finishing their work early wait for the others before continuing. The coordinator can control this by sending out synchronize requests to all participating viewers and wait until all of them has responded before sending the next state. Each viewer will not respond before it has finished the work it has already started. It is wise to synchronize the viewers as the consequence by not doing it is that the viewers will show fragments from the same frame at different times. Something as simple as the image moving smoothly across the screen would not be possible as some parts of the image would move faster than others and if the image actually did do such a thing without synchronizing, it would be a pure coincidence.

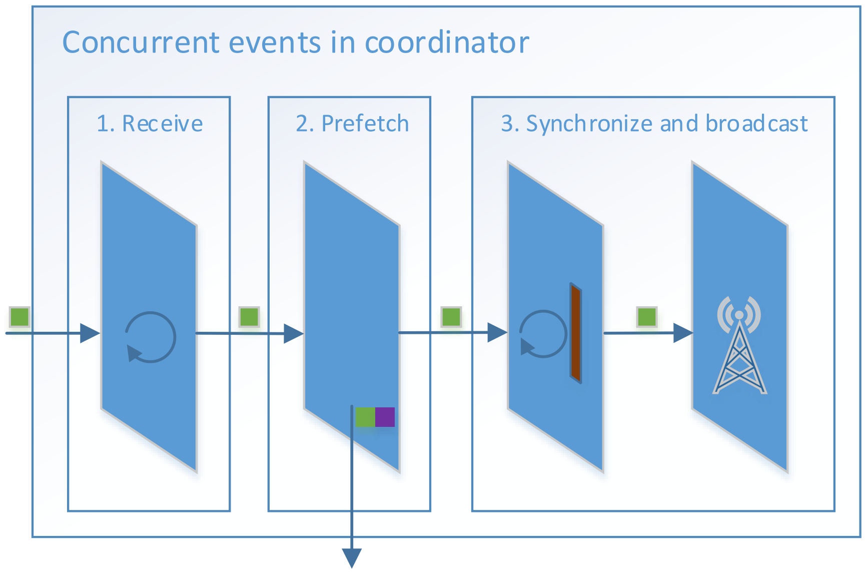
Concurrent events in coordinator
1. Receive
2. Prefetch
3. Synchronize and broadcast

Figure 3.2: This figure shows the coordinator's design and concurrent events. Each box illustrates a concurrent event. 1. the coordinator receives a state from the web interface. 2. the coordinator sends a prefetch request to the cache server with additional fetch data. 3. the coordinator synchronizes the viewers before the state is broadcast to them.

3.4 The Viewer

The viewer is designed to do three main tasks:

1. Use received states to determine which image fragments to use.
2. Fetch these images from the cache.
3. Draw them onto the screen on their calculated positions.

When the coordinator broadcasts a state, it doesn't know which of the viewers that actually needs it, as all viewers might not view a part of the global image from this particular state. Each of the viewers are left with the task of determining whether or not to display any image fragments at all by using the state provided by the coordinator. As mentioned earlier, a state is the position of the global image in x, y and z coordinates and by adding the image size for the current zoom level, we now know the area spanned by the image. By comparing this area with the area spanned by the display tile in its position in the tile grid, we can determine if any image fragments are intersecting. Intersecting image fragments will be fetched and drawn, which brings us to the next step.

When we have determined whether or not to fetch images, the selected images are fetched from the cache. We will go further into details about the cache in the next section, but for now we will concentrate on the viewer.

In the process of determining which images to draw we did already calculate the image fragments' file name on disk and it's position to be drawn. This makes it fairly easy to request these images from the cache, as the key to each cache entry is the file name to the corresponding image data. A huge difference from the design in the capstone project is that we don't request an image from the cache before we know we will use it. In this way, the cache will only contain valid images, in contrast with the old design where the cache could be filled with invalid file names containing nothing.

After the viewer has retrieved the image fragments from the cache, they are all drawn onto the screen using their calculated position. One of the lessons learned from the capstone project was that using the CPU to do graphics rendering is inefficient and a poor use of available computing resources. Therefore, the design was changed to enable the Graphics Processing Unit (GPU) to do the graphic rendering and releasing the Central Processing Unit (CPU) form this task. We will go further into how this is done in the implementation chapter.

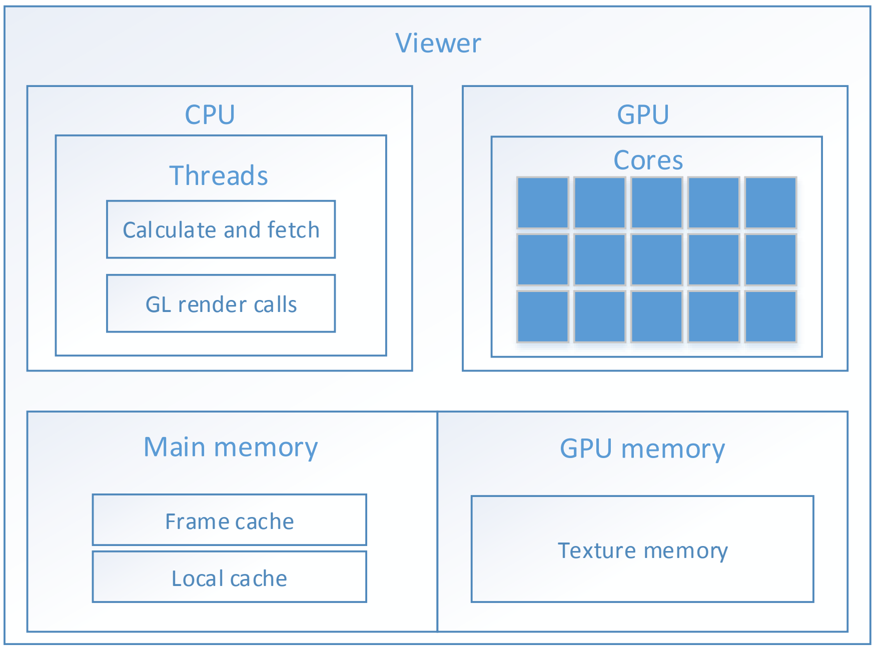

Figure 3.3: This figure illustrates some of the components in the viewer. The two main threads is one responsible for calculating the image positions and fetching them. The second is responsible for making all calls to Open Graphics Library (OpenGL). In main memory we find the frame cache and the local cache. The frame cache holds references to textures and the textures holds image data located in the GPU's texture memory.

3.5 Cache

Until now we have been talking about the cache as one single component. This is not exactly true. The cache consists of three cache levels:

1. Frame cache
2. Local cache
3. Centralized cache

The two first cache levels are found once at each viewer and the third is a centralized cache server, a single node reachable for all the viewers. The cache levels have different sizes and usages.

3.5.1 The cache levels

The frame cache only contains references to image fragments located in GPU memory from the current frame being displayed. If the image is not moving too fast, a lot of these image fragments should be reused when receiving the next state. We will took into the results of this in chapter 5, after running the experiments. The frame cache is designed to cover most of the requests and still be very small.

The local cache is capable of storing several frames and stores only the image fragments just displayed, creating a map over the viewer's recent whereabouts during the last moments. This cache level is great to have when the global image is moving a lot over the same area. It can be seen as a bit redundant to have both a frame cache containing only one frame and a local cache containing only a few more, but even if the same fragment is stored in both cache levels, the data is not exactly the same. We will give more details about the data contained in the cache levels in the implementation chapter.

The centralized cache is capable of holding a number of image fragments equivalent to 2-3 times the combined viewport of the participating viewers. The big advantage of this cache level is that once it has fetched an image fragment requested by a viewer, all other requests to it by other viewers will result in a cache hit, as long as it is not replaced.

The frame cache is the one closest to the viewer and therefore the one receiving the request directly from it. On a miss, the request will be forwarded to the local cache and if it is missing there too, the request is sent out to the cache server.

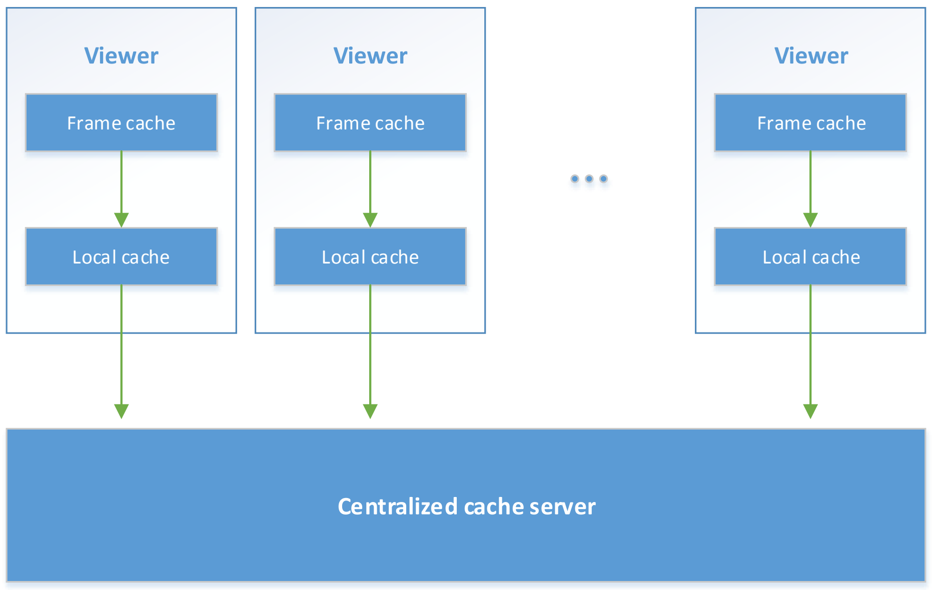

Figure 3.4: This figure illustrates the different cache layers and the relation between them. There are one frame and local cache in each viewer process and one centralized cache server common for all viewers.

3.5.2 Replacement algorithms

As the cache levels are meant to be used in different ways, they are not using the same caching algorithms.

The frame cache is using a very simple algorithm to replace data. As it only holds one frame at a time, it will compare the new frame to the one it holds and discard all image fragments not intersecting to make room for the new frame. Because of the small size of the cache, this method for replacing frames goes fast. It might not give the same performance if we increase the size of the cache, though, as it has to go through all cached fragments in order to delete the old ones. This happens once for each frame and would always give a best case scenario equal the worst case, O(n). For large cache sizes this algorithm will might be too time consuming, affecting the performance of the image viewer.

The content of the local cache forms what we call the “tile-tail”, meaning the image fragments already displayed by the viewer it belongs to. As the cache fills up, the tail grows longer and to deal with the issue of replacing the content, a LRU algorithm fits good with the need of this cache level. This cache level is designed to have cache hits if the image’s move direction changes to an area where it has already been just moments ago.

The centralized cache uses a totally different replacement algorithm. This cache level is the only one using prefetching to load images before they are requested and therefore we must be careful not to replace these images in the cache. One of the better ways to avoid this is to adopt a replacement policy that only removes image fragments from the cache that is outside the viewport and in the opposite of the current move direction. In this way, we will neither replace any of the images in the viewport, currently used by the viewers, or any of the prefetched image fragment soon to be used.

3.5.3 Prefetching algorithm

Two components are involved when prefetching states. That is the coordinator and the cache server. The method for prefetching image fragments is therefore divided into two parts. First, determine whether or not to prefetch images based on the current state and second, fetch the new image fragments into the centralized cache.

As mentioned earlier, the coordinator is the one detecting when to prefetch. This is done by keeping track of the first image fragment in the upper left corner and the last one in the button right of the viewport. By doing this it can detect if these fragments changes from state to state. When they do, depending on the move speed, the next fragments in these positions might not be the neighbours of the previous ones. Therefore, we have to find the number of fragments between the new and old image fragment and send it, as well as the current state and move direction, along to the cache server.

When the cache server receives a prefetch request, it calculates the n*m first image fragments to fetch in the opposite of the move direction, n being the received amount and m being the number of image fragments that fits in the viewport's opposite 2D axis of the move direction. E.g. If the global image moves to the right making 2 new images to appear in the x plane and the number of image fragments in the viewport height is 4, then the cache server should prefetch the 8 image fragments most left in the new viewport caused by the given state.

3.6 Communication

Considering the given architecture, there are a number of ways to let the components communicate between each other. From the capstone project we learned that one of the better approaches was to let the coordinator broadcast the states to the viewers. By using this method of communication, the viewers becomes loosely coupled, which is a benefit in a design like this where many units of the same component are involved.

This approach is based on the principles of a linear broadcast, where the viewers has to react to a scheduled program, i. e., the stream of states provided by the coordinator during the usage of the image viewer. From here we can go several ways to achieve the broadcast.

One way is to transfer the states by using a hierarchy of nodes, where the coordinator is the root node and the viewers form the rest. The benefit of this approach is that the coordinator only have to forward the states to a small number of viewers, saving time and CPU expenses at the coordinator. Thereby, the viewers can forward the states to the ones further down in the hierarchy concurrently. Using a huge number of viewers, this approach might be time efficient, but it comes with a cost. The viewers are no longer loosely coupled and if one node goes down, some of the others might not get the states being forwarded.

Another way is to let the coordinator broadcast the states to all viewers. This method was used in the capstone project and in theory, it might use more time on broadcasting each state to all viewers. As the number of participating viewers was a maximum of 28 at the Tromsø Display Wall and this method of communication seemed to work fine, we kept it in the new design.

By using a linear broadcast, it would seem like the requests are received more or less at the same time at all viewers. Still, the viewers might use a different amount of time to receive and process a request. The greatest reason for this, due to the approach of communication, could be caused by the network traffic and routing to each viewer.

4 Implementation

During this project, a prototype was implemented using the design pattern described in chapter four. Keeping the principles of concurrency from the last prototype has been in focus throughout the implementation of the image viewer. That involves the coordinator, the viewer and the cache server.

The go programming language[18, 19, 20] was chosen to implement the image viewer. This programming language fits good as it has built-in features for achieving concurrency relatively easy. This was used in a combination with OpenGL[21, 26, 27] bindings as graphical framework to render image fragments.

The web server hosting the interface was implemented using flask[23] and hosts the web interface build by a combination of html, javascript, jquery[24] and ajax. The interface gives the client the ability to set up the run configuration to be used by the image viewer. The configuration is written to several json files and opened by the image viewer components at startup.

All communication between the components, except between the client and the web server is carried out using web sockets over Transmission Control Protocol (TCP). The client-web server communication is based on Hypertext Transfer Protocol (HTTP) requests, both synchronous and asynchronous.

4.1 Web interface

The web interface is divided into two parts, the client side and the server side of the application.

The client side of the application sends requests to the server side to provide the necessary data to initialize the image viewer. The image viewer is started by the server on the client's command using python's subprocesses to create separate os processes for the coordinator, the cache server and each of the chosen viewers. After the image viewer is started, the client communicates directly with the coordinator without using the flask server as a middleman.

To make sure that the coordinator and the viewers are up and running before the client starts interacting with them, a poker script is deployed. This script pokes each of the components by sending them poke requests until it get responses from all of them. The web server waits until the poker script has finished poking all the components before giving the client the ability to interact with the image viewer.

Local storage[25] is used to store data at the client, such as the image position, size of the selected display area, the selected input flags and run configuration. This, to remember the client's settings when navigating between the web pages at the site.

The interface uses some default settings making it easier for the client to use the system without having to provide much data. For the more experienced users, a drop down dashboard is available to change the default settings. The settings available is a set of input flags for selecting which cache levels to use, whether or not to use prefetching, synchronizing and logging. By default, the image viewer runs at 60 frames per second, but this value can also be changed.

The request flow from the client to the coordinator is limited by the same rate value as the fps at the viewers. This means that by using the default setting of 60 frames per second, the image position will be packed into a state and sent to the coordinator periodically 60 times a second.

4.2 Coordinator

The coordinator has gone through some huge changes since the capstone project. In the capstone project, the coordinator was located at the client's computer using an Simple DirectMedia Layer (SDL) window to gain input and interact with the viewers. This has all been changed in the current implementation. The coordinator is now found at the display wall cluster and is started as a separate os process by the flask server at the frontend. It now works as a server, receiving states directly from the client. Depending on the settings chosen by the client, the coordinator will prefetch and synchronize the viewers in addition to broadcast the state to them.

If prefetching is enabled, the coordinator will use each state to determine whether or not new image fragments will appear in the image viewer's viewport during the next frame. This is done by keeping track of the first image fragment in the upper left corner and the last one in the bottom right. If one or both of these images changes, the coordinator will send a prefetch request to the cache server as well as the current move direction and the amount of images between the last image fragment and the one replacing it. This number tells the cache server how many images to prefetch in the the move direction axis.

After the coordinator has determined whether or not to prefetch image fragments at this particular frame, it will broadcast the state to all participating viewers by concurrently looping through the established websocket connections to each of them, forwarding the state. After the state has been broadcast to all viewers, the coordinator will send a synchronize request to them if synchronization is enabled.

The above statement of synchronizing is not completely accurate on how often synchronize requests are sent to the viewers. The client has the opportunity to set the number of states sent between each synchronize request, making the coordinator wait n states between each time it synchronizes the viewers.

4.3 Viewer

As the coordinator, the viewer has also gone through some changes from the implementation in the capstone project. There are two changes making a graphical difference from the client's perspective. That is the change of graphics library to render image fragments and the applied caching techniques to fetch images. Both changes affect the behaviour of the viewer and the rest of its implementation. We will now talk a bit more about the graphics library and come back to the cache in section 4.4.

4.3.1 Graphical framework

In the last prototype, image fragments were blit onto a surface by using SDL. This gave a bad performance and few opportunities when it came to graphical rendering.

Now, the SDL library is used only to load an image fragment from disk into the cache and OpenGL is used as the graphics library doing the rendering. This change gave a better performance, but also some unseen consequences.

As it turned out, OpenGL was not thread safe and made the viewer crash at random times. As mentioned, the viewer was implemented by the use of concurrency and in order for it to be able to use OpenGL to do rendering, a separate thread was created for making GL calls only.

The OpenGL context can only be bound to one thread at a time, making this is one of few proper ways to do graphics rendering when operating with multiple threads or in our case, goroutines[28, 29, 30]. Using go's channels, a draw request can be sent from other goroutines to this single graphics routine and make the GL calls from here.

In this context, using OpenGL instead of the SDL library, means to use the GPU to do graphics rendering instead of the CPU. Not only did this increase the number of frames rendered per second, but it also released the CPU from doing all the rendering and lets the GPU do what it was meant to.

By using this graphics library, we gain a visual benefit when zooming. Instead of jumping directly to the next zoom level, OpenGL gives us the ability to stretch the image fragments enough to make a stable resizing effect of the global image from one zoom level to another. As the image fragments is already loaded into textures, they will be reused and the only changes have to be made are the size of each fragment and its position in the viewer's viewport. All fragments at a given zoomlevel are stretched until the size of the global image equals the one at the next zoom level.

4.3.2 Data flow

Each viewer has one goroutine constantly listening for incoming states from a websocket. When a state is received, it is sent to another goroutine over a channel where it is used to determine whether or not to render image fragments at all. Sometimes a viewer is not supposed to render any image fragments in a frame because the global image has moved outside this particular viewer's viewport. If that is the case, the state is dropped and the viewer is waiting for another state to receive.

If the global image is inside the viewer's viewport, then the viewer will use the state and its position in the tile grid to determine what images to render and where to render them in the viewport. When this is done, it's time to fetch the image fragments from the cache. We will go further into details on how the cache works in the next section.

When all image fragments has returned from the cache, they are sent over a channel to the GL rendering goroutine. Here they are put into a texture and rendered at their belonging positions.

A consequence of using OpenGL is that we cannot use concurrency to render images as before, but this might not be a loss, as we expect rendering image fragments sequentially with OpenGL to go faster than blitting them concurrently with SDL.

4.4 Cache

All three cache levels are implemented using maps with the file name of an image fragment as a key to store the image data behind. The file name of an image fragment is a string corresponding to it's position in the global image when the image fragments are all tiled in a 2D plane using an x, y and z coordinate in a particular zoom level z. We will now give a more detailed description of each cache level's implementation, starting with the one furthest from the viewer.

4.4.1 The centralized cache

The centralized cache is implemented as a cache server mainly handling 2 types of requests. That is prefetch requests from the coordinator and fetch requests from the above cache level at the viewers. Of course, the cache server will only receive prefetch requests if prefetching is enabled by the client. If it isn't, the cache server will load image fragments from disk when receiving fetch requests from the the viewers.

Prefetch requests

A prefetch request contains the request type, the state of the global image in which to prefetch, the current move direction and the number of image fragments to fetch in the opposite of the move direction.

```
var prefetchRequest string = "requestType/state/direction/amount"
var prefetchRequest string = "prefetch/0|0|10/left/1"
```

When such a request is received, the cache server will determine the exact image fragments to load into its map. By using the provided move direction, the cache determines what fragments to fetch in opposite direction, where the new fragments appear. The given amount, n, tells how many fragments to load in this direction and the number of image fragment in the opposite xy-axis, m, gives a total of n*m image fragments to load on a single prefetch request. The boundary for the image fragments contained in the cache is an extension of the viewport by one image fragment in each direction. If a fragment is positioned outside this bounds, it will not be loaded.

On the first prefetch request the cache server has to load all image fragments inside the bounds into the cache. The fragments are loaded concurrently, meaning that the number of fragments to load should not have too much affect on the total time used on loading them.

Fetch requests

A fetch request consists of two parts, a string telling what kind of request this is and the generated file name of the image fragment being requested. The file name does not contain the file extension, but it will be added later when loading the image fragment.

When receiving a fetch request, the cache server simply looks up the requested fragment in it's map by using the provided file name as a key. On a hit, the data behind the key, as well as additional data about the fetch operation, is used to create a cache data object. This data object is marshalled by a json codec from the websocket package before sent back to the viewer requesting it. On a miss, the image fragment is loaded from disk and put into the cache in addition to the procedure just mentioned. The two last arguments in the marshalled data object is added in order to count cache hits and misses as well as measuring fetch- and load time at the viewer.

```
var fetchRequest string = "fetch/0-0-0"
var response  = CacheData{
    srfc surface.Mini,
    "0-0-0" string ,
    hit bool,
    loadtime time.Duration
}
websocket.JSON.Send(response)
```

The cache server is using the SDL library's built-in load function to load the missing image fragments. This gives us a SDL-surface containing a lot of data, some that we won't even use. As we are going to send this back to the viewer and the number of cache requests can can be really large at times, the surface is trimmed to only contain the data absolute necessary for it to be rendered. Everything except the image width, height and the raw pixel data is removed, leaving us with much less data to pass to the viewers. For the ease of understanding, we will refer to this data as a minisurface.

4.4.2 The local cache

The local cache only receives cache requests if the required data is not available in the frame cache. Each cache entry in this cache consists of a minisurface and a timestamp, which is accessed by the same file name key as in the centralized cache. By storing a timestamp with each minisurface, we can easily find the least recently used image fragment by using golang's time package.

On a hit, the minisurface is simply returned to the frame cache and on a miss, the viewer will either send a fetch request to the centralized cache server or load the image fragment itself, depending on the cache server is enabled by the client or not. If the viewer has to send a fetch request to the cache server, it will get a json string in return, which will be unmarshalled into a minisurface and stored in the local cache. If the viewer has to load the image fragment itself, it will be loaded into the local cache in the same way as the cache server does, by stripping the surface into a minisurface. In this way, we don't have to store data in the cache that is not going to be used and the cache will have more room for more important data.

4.4.3 The frame cache

The reason for using this cache level came with the idea of using OpenGL. The main difference between this cache level and the other two is that we go from storing a minisurface to loading it into a texture and then store the texture. The texture's image is contained in the GPU memory and when the frame cache has to replace its content, it simply reuses textures from the previous frame. This is why the frame cache is so important, because a hit saves a lot of time loading the minisurface into a texture. The only change on a hit is the position to render the texture and some times the texture size in pixels. It also saves time on reusing textures instead of creating new ones for each minisurface.

The number of image fragments contained in a frame displayed by a single viewer can vary from each frame. To separate textures ready to be used and the ones that is going to be reused, we always select free textures from a texture pool. At startup, the texture pool is filled with a number of textures according to the maximum number of image fragments that fits into a viewer's viewport at once. Every time we load a minisurface into a texture, the texture is fetched from this pool and before we start loading the minisurfaces belonging to the next frame, all textures from the previous frame that is not going to be used in the next, are removed from the cache and put back into the texture pool, free to be used again.

5 Evaluation

The gigapixel image viewer will be evaluated through several iterations of experiments and measurements. For each iteration, the results will be the foundation for the next one. In advance, we have planned experiments for a few iterations, but based on the results we might have to add more or change the ones we have planned. As an experiment can result in a dead end, there might be other measurements than planned that is more useful to research. Also, we might get results indicating that it's more interesting to measure something in the complete other direction than planned. This is the why we have chosen to do experiments in several iterations.

5.1 Methodology

Before we start looking into how the experiments were executed in each iteration, we will give a brief technical description of how the experiment environment was set up.

5.1.1 Tromsø Display Wall

All experiments and measurements were done using the Tromsø Display Wall at the University of Tromsø. The display nodes, as well as the frontend, run Linux with the Ubuntu 14.04 distro (64 bit). The display tiles run x11 for graphical output with go version 1.3.1 and python 2.7.6 installed on them.

The hardware of a display node consists of a Intel Xeon W3550 processor, a GeForce GTX 560 Ti graphics card and 12 gigabyte (GB) of system memory. All the display tiles are connected to the internet by Gigabit Ethernet through the same switch. Each of the projectors in this wall displays an image with resolution of 1024x768 pixels with refresh rate of 60 Hz.

We are aware of the fact that others might use the cluster when we do our experiments, which can affect our measurements. There is little we can do about this, as we don't have the opportunity to claim the whole cluster to ourselves.

5.1.2 Data set

All experiments in this project use the same gigapixel image consisting of 10 zoom levels where each of the image fragments has the resolution of 256x256 pixels. This image has the approximate size of 5 GB, all zoom levels combined. Through the experiments, a stream of image states will be generated to simulate the client interacting with the viewer through the interface. Two kinds of streams will be generated with different image movements, one used when experimenting with the cache levels separately and another for when they are combined. The input stream created when experimenting with the cache layers separately makes the image move in a rectangle before zooming in and out on it. When multiple cache layers are enabled, we will use an input stream more suited regarding the usage of the enabled cache layers. We will come back to this in section 5.4.1.

Mainly, the image viewer will run fullscreen, with all 28 display tiles enabled as this would create a greater dataset. The size of the display area in combination with the input stream and the move speed will determine the amount of image fragments that will be used from the selected gigapixel image for each run. If any results indicates that we should experiment with a smaller number of display tiles, we will do so in accordance with the needs as we see fit.

5.1.3 The image viewer

The experiments done to the image viewer will be executed with all 28 available viewers enabled at 60 frames per second. When running, each viewer will write data regarding measurements taken into its own log file. This happens once after every frame has been viewed, creating one log file per viewer with one in each log file for each frame displayed. Each viewer will display the same amount of frames and after all viewers has finished their input stream, all log files will be parsed in order to collect the log data and compare the data belonging to the same frame. In this way, when finding the average of some measurement, all measurements of the same property in the same frame will be added and divided by the number of its appearances. Each viewer will first do this once for each image fragment appearing in the frame and then it will be done once for the number of viewers participating. All cumulative measurements will be the average value of a property measured once per frame, e.g. the cache hit, during the same frame. It will be collected over the number of frames displayed, increasing and decreasing respect to the average of the value in each frame.

Synchronization will be disabled during the experiments, mainly because it makes the system use a bit more time when running. Later when measuring the frame rate, it will be enabled again.

The web interface was not used to control the image viewer under the experiments. It was developed by using Google Chrome, version 41.0.2272.118 and have been used to control the image viewer when using the prototype outside experiments. Evaluation of the web interface has only been done manually and not by the use of benchmarks.

5.2 Iteration zero - baseline experiments without caching

In order for the results of our experiments to have any meaning, we will first create a baseline with no caching techniques enabled. Whether or not the system will gain any performance from applying caching techniques will be determined by comparing the baseline to the results of our coming experiments and measurements. This will be discussed further in chapter 6.

5.2.1 Baseline

The measurements we will take when building the baseline are the frame rate and the average fetch time without any cache layers enabled. These factors indicate how well the system performs. Synchronization should not be enabled when building the baseline. As no cache layers are enabled, the total fetch time is the time it takes loading the image fragments from the NFS, i.e. the load time in addition to the time it takes to load the image fragment into a texture. We will use these results later when analysing the cache layers.

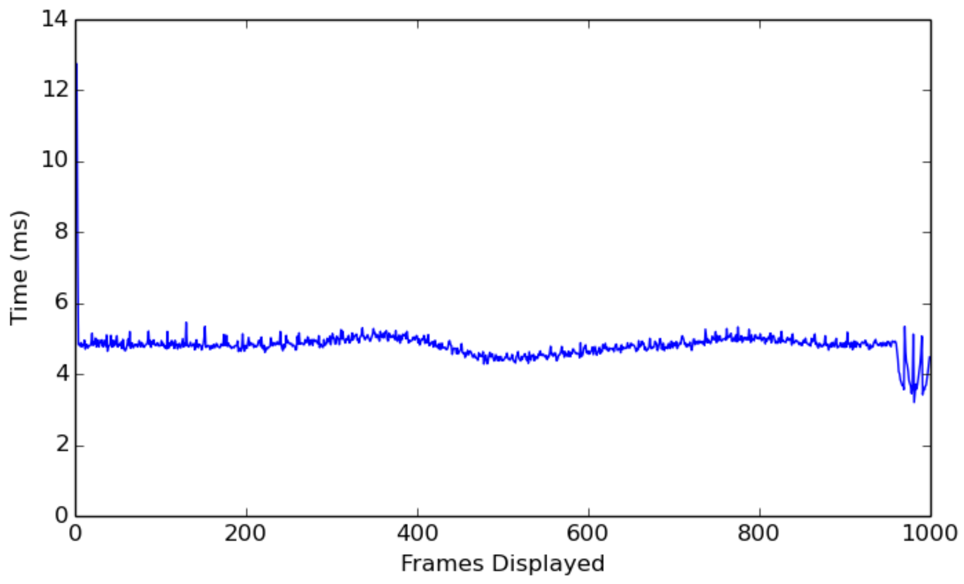

Figure 5.1: This graph shows the average fetch time without any caching layers enabled and with all 28 viewers enabled. Each sample in the graph is the average fetch time of all viewers during the same frame. Each viewer creates an average fetch time from all image fragments in a frame and then an average is created out of all viewers' average fetch time from the same frame.

Figure 5.1 shows that it takes about 5 milliseconds in average to load an image fragment from the NFS and into a texture. Depending on the image position, a frame is capable of containing upto 20 image fragments at once inside a single viewer's viewport.

5.3 Iteration one - fetch time and cache hit

After creating a baseline, we can now start measuring the hit ratio and average time being used on fetching image fragments from the separate cache layers. The total fetch time is measured from the moment the viewer finds out that it needs an image fragment until the fragment is loaded into a texture. In order to isolate the time being used on communication between the viewers and the cache server, the remote fetch time is only measured when the cache server is enabled. It starts when a request is written into a websocket and ends when the same viewer receives a response from the same connection. The time it takes to load an image fragment from the NFS and into the centralized cache is called the load time and this time duration is subtracted from the remote fetch time when writing to the log file. The psudocode below illustrates how total fetch time and remote fetch time are measured when sending fetch requests sequentially to the cache server.

```
func loadImageFragmentsIntoTextures(fnames []string){
    fragmentChan := make(chan *imageFragment)
    startFetchTime := time.Now()

    for fname := range fnames{
        go func(fname){
            fragmentChan <- sendFetchRequestToCacheServer(fname)
        }
    }

    for range len(fnames){
        fragment := <-fragmentChan
        bindToFreeTexture(fragment)
        log.WriteToFile(time.Since(starFetchTime))
    }
}

func sendFetchRequestToCacheServer(fname string) *imageFragment{
    var fragment imageFragment
    ws := createWebsocketConnection()
    message := MarshalFetchMessage(fname)

    startRemoteFetchTime := time.Now()
    websocket.SendMessage(ws, message)
    websocket.Receive(ws, &fragment)
    remoteFetchTime := time.Since(startRemoteFetchTime)
    log.WriteToFile(remoteFetchTime-fragment.LoadTime)

    return &fragment
}
```

We will run each experiment with one alternating factor to begin with, the move speed of the image. Three different move speeds will be used, creating three different datasets as the dataset grows with the move speed.

5.3.1 Fetch time

From the capstone project we learned that the fetch time was the main bottleneck in the system and therefore we have focused on improving the fetch time during this project. Through this section of experiments we will measure the fetch time separately on each of the implemented cache layers and compare them up against each other to find out how much time we can expect them to use individually. We expect the fetch time to increase for each cache level further down in the system, as the path for finding the image fragment grows longer for each miss.

To find out how much time is used on a single fetch request to the different cache layers, we will conduct the experiment on the image viewer by running it several times with each of the cache layers enabled separately. We will measure the fetch time at the cache server with and without prefetching to see if the fetch time changes. As the cache server is located on a different machine than the viewers, we will also measure the time used on sending the data to and from the server, i.e. the remote fetch time. This experiment gives an indication of which time units to use when comparing the results and what combinations of the cache layers to use in order to get the best result.

5.3.2 Cache hit

The main reason for measuring the cache hit at this point is to see if the fetch time from the other experiments are related to the cache hit ratio. A cache hit should result in low fetch time in contrast with a cache miss, that in this case, should load the image from the NFS and use a lot more time. The cache hit ratio will be measured in the same experiment as the fetch time, by measuring cache hit and miss in each of the cache layers separately. The centralized cache exists in one process located at the frontend, unlike the frame and local cache that relies in the same process on the same display tile. For each request sent to the cache server, the image data is returned with a value telling whether or not the request resulted in a hit or miss. Each viewer will log the hit ratio from the cache server based on the responses from its own requests. We will look at the cache hit ratio at the cache server with and without prefetching to see if we gain a greater cache hit ratio by doing so.

5.3.3 Results

The results gained from the first iteration will be presented in three sections, one for each cache layer. As mentioned, the experiments was ran by using three different move speeds for the image. The move speed is measured as the number of pixels moved between each frame, and will be denoted as “p/f” (pixels per frame).

Frame cache

By collecting the results from the experiments run by enabling only the frame cache, we were able to create graphs illustrating the fetch time and the cache hit ratio. Figure 5.2 shows the fetch time represented in three graphs, using three different move speeds when moving the image across the screen. We first tried creating graphs showing the fetch time for both cache hits and misses, but as a cache miss results in loading the image fragment from the NFS, the fetch time could take up to several milliseconds. The graph shows the average fetch time for all image fragments in a frame from all participating viewers. A single miss occurs often when using many viewers and makes the fetch time increase from nanoseconds to milliseconds. The result was an unreadable graph and because of this, the fetch time represented in the graphs using cache hits only.

As we compare the graphs in Figure 5.2, we see that the fetch time in general varies from 150 to 300 nanoseconds. We explain this difference as the display tiles using different amount of time when fetching the image fragments from the frame cache. Some might be faster than others, making some tiles to generate a higher average fetch time during the same frame. As the move speed increases, we see that the fetch time occurs more often with a high value around 250-300 nanoseconds. What actually happens is that when the data set is increased by the move speed, we get a greater variation of requested image fragments and the duration of an image fragment's life time inside the viewer's viewport decreases. The amount of different versus the amount of similar image fragments are requested often when using high versus low move speed.

The 40 last frames in these three graphs are the zoom movement, first zoom in 20 frames and then zoom out 20 frames. We expect the increased fetch time at the end to be the change of image size, as image fragments are stretched when zooming.

During the experiments, the cache hit ratio was measured along with the fetch time. The results from the cache hit measurements was collected into three graphs, but as they were nearly identical, we will only show one of them.

From Figure 5.3 we see the cache hit increasing fast and reaches quickly 96%. It stays there until the end, where it decreases a little when zooming out. This is expected, as zooming out means that all image fragments in the frame must be replaced, versus zoom in, where we scale up the current images until we reach a new zoom level.

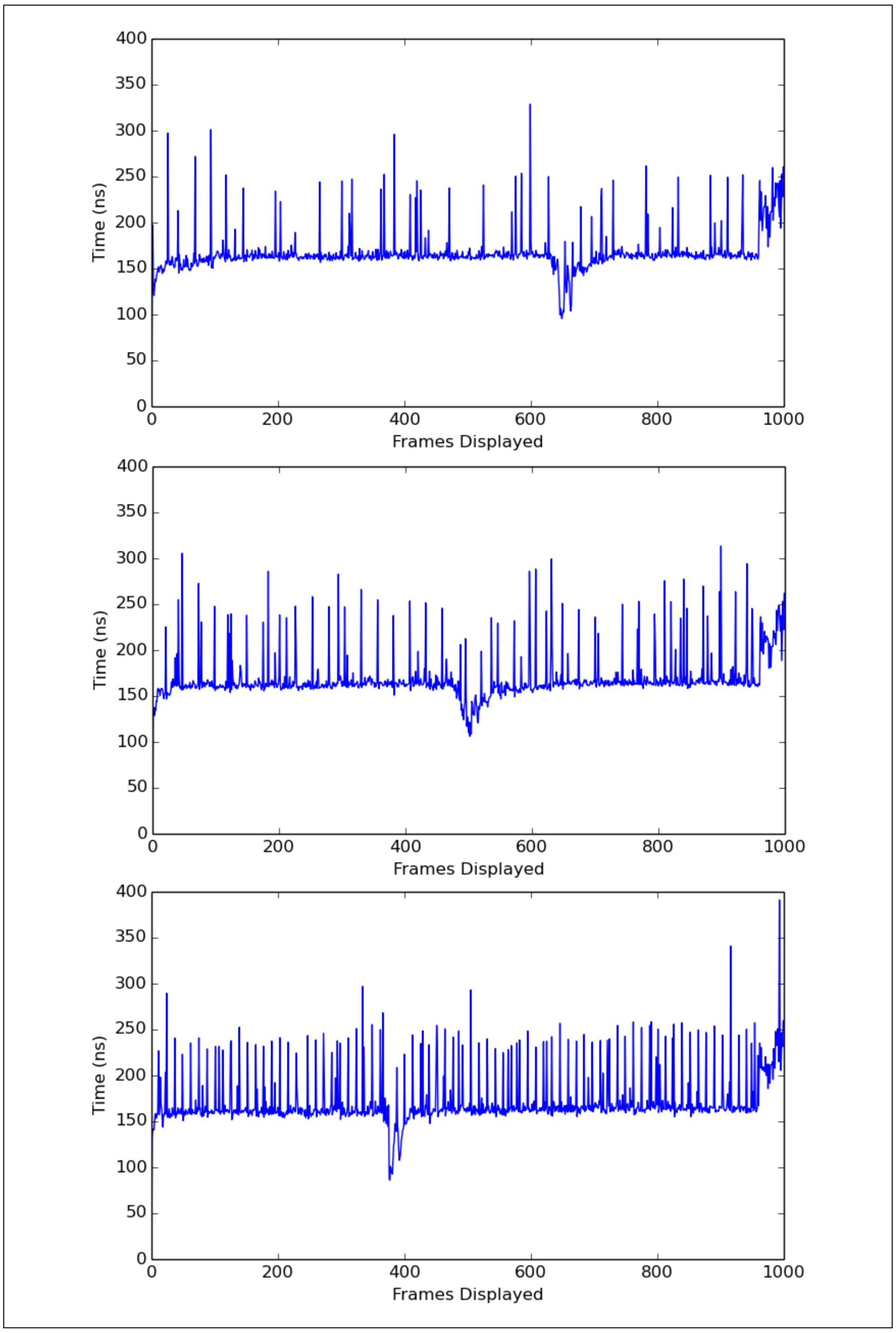

Figure 5.2: Average fetch time using the frame cache only. The first graph represents the average fetch time when the image is moving with the lowest speed of 5 p/f, the second with 10 p/f and the third with 20 p/f. Only cache hits are shown in these graphs and 28 viewers were used when generating them.

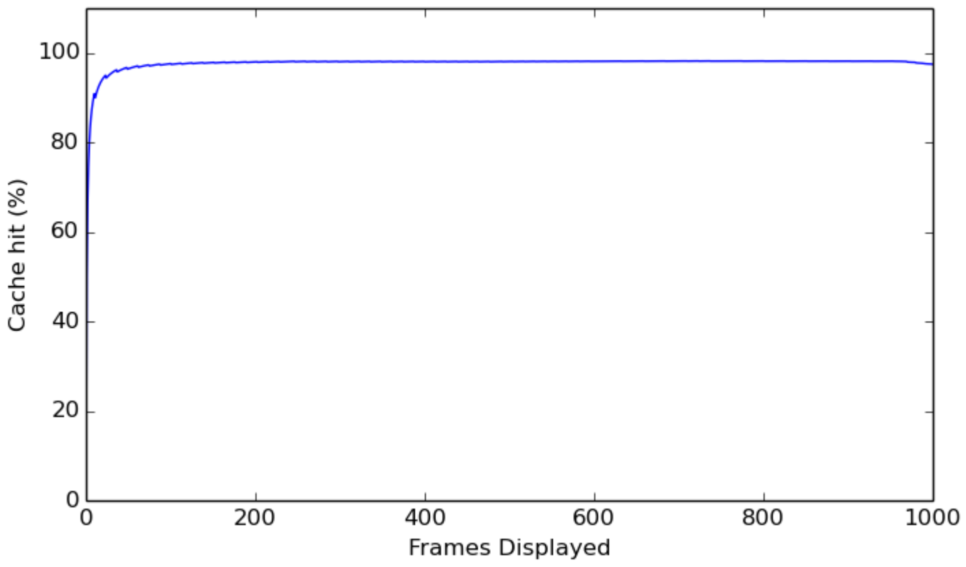

Figure 5.3: Cumulative cache hit in percentage with only the frame cache enabled.

Local cache

After gathering the results from the experiments with the frame cache, new experiments was conducted with the local cache. As we faced some problems representing both cache hits and misses in the same graph regarding the frame cache, the experiments with the local cache also resulted in graphs showing the fetch time for cache hits only.

As we can see from the graphs in Figure 5.4, a fetch request from the local cache takes microseconds compared to the graphs in Figure 5.2, the frame cache, using nanoseconds. As both cache levels are located in the same process on the same physical machine and they both use a map containing the image data, a lookup in the map should not differ in time, but as the data from the local cache has to be bound to a texture, the fetch time automatically increases when using the cache levels further down in the system. If we compare the graphs from the local cache experiments, we see that the fetch time is almost the same in all three graphs, between 200 and 250 microseconds. Even if this is a thousand times more than the frame cache, it would still be possible for the image viewer to run with a high frame rate, like 60 frames per second.

In each of the graphs there are a few spikes, indicating that the fetch time was 4-5 times greater than the overall average. We are not completely sure what causes them, but for the moment we assume they are noise from other applications running at the cluster during the experiments.

The cache hit ratio was measured during the experiments, resulting in three graphs and just like the frame cache, these graphs was nearly identical. One of the graphs is shown in below.

From Figure 5.5 we can see that the hit ratio increases rapidly at the beginning, reaching 97%, where it stays until we start zooming out at the end.

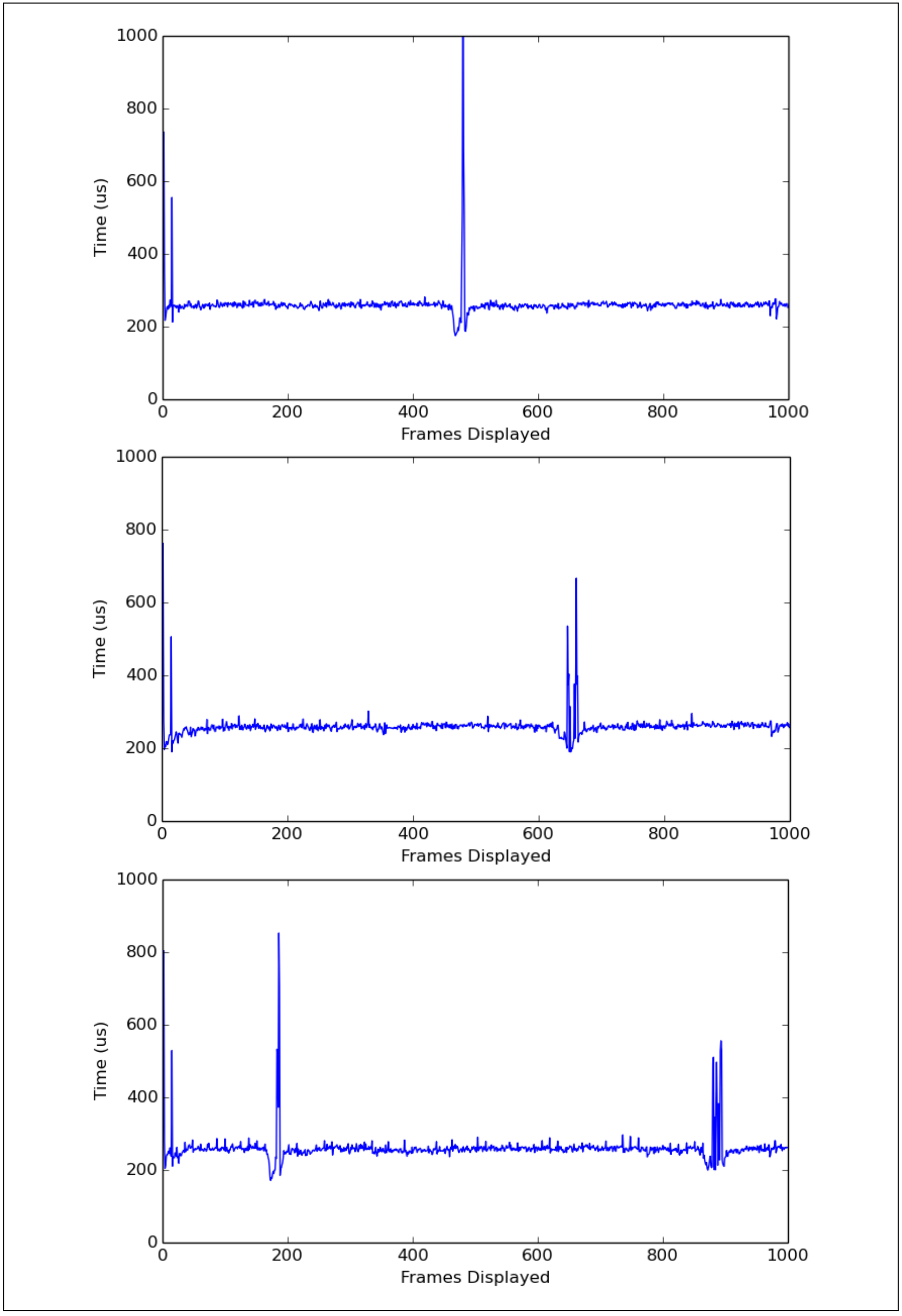

Figure 5.4: Average fetch time using only the local cache. The first graph shows the average fetch time when the image is moving with the lowest speed of 5 p/f, the second with 10 p/f and the third with 20 p/f. Only cache hits are shown in these graphs and 28 viewers were used when generating them.

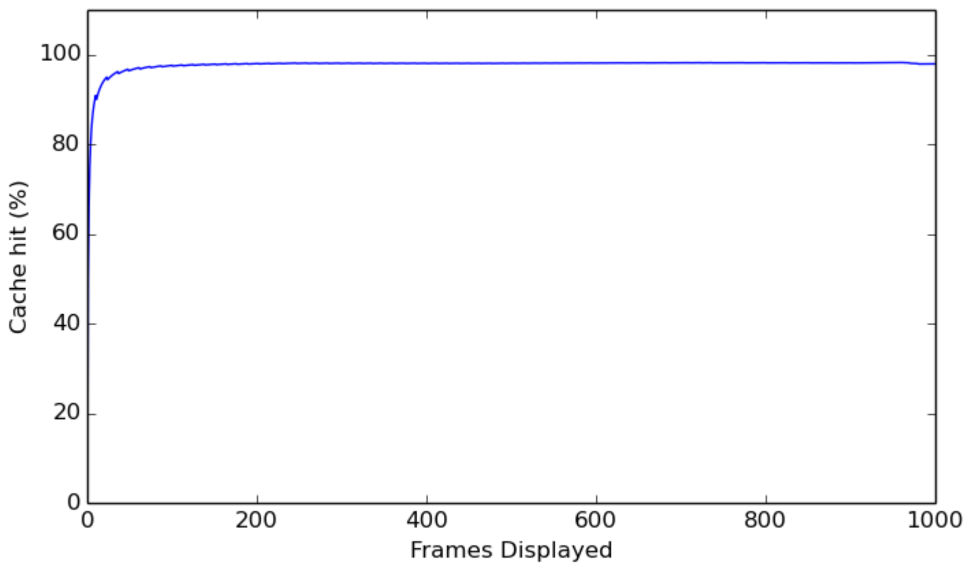

Figure 5.5: Cumulative cache hit in percentage with only the local cache enabled.

Centralized cache

The last experiments conducted in this iteration were based on finding both the total and the remote fetch time using the centralized cache. The cache hit ratio was also measured during these experiments. The experiments were conducted three times with the same parameters as with the other two cache layers. The results were collected into three graphs showing the total and remote fetch time for the different move speed parameters.

As we can see from Figure 5.6, the average total fetch time generally lies between 60 and 65 milliseconds. Also, most of this time goes into remote fetching image fragments from the cache server. At the end of each of the graphs, the remote fetch time decreases and as a direct consequence, the total fetch time does too. We suspect the decreasing remote fetch time at the end is caused by fewer fetch requests being sent to the cache server as some viewers completes the input stream before others. This would not happen if the viewers were being synchronized, as the coordinator would force them to wait for each other.

From Figure 5.7, we see that without enabling prefetching, the hit ratio in the centralized cache grows rapidly in the beginning. In contrast with the other cache layers, here it reaches approximately 100% cache hits. The high hit percentage in this layer is because the same image fragment is fetched once and used multiple times by several viewers. Once fetched into the cache, a request for it will result in a cache hit as long as it is inside the combined viewport of all participating viewers.

When prefetching is enabled, the cache hit is exactly 100%, as no image fragments are missing when requesting the first frame. Comparing the two graphs in Figure 5.7, we can see that they are almost the same. The main difference is that enabling prefetching makes the image fragments current in the centralized cache already before the first fetch request arrives. Even if the prefetching is enabled, we will not gain any visual benefits, as the average fetch time still will be the same after from the second frame and out.

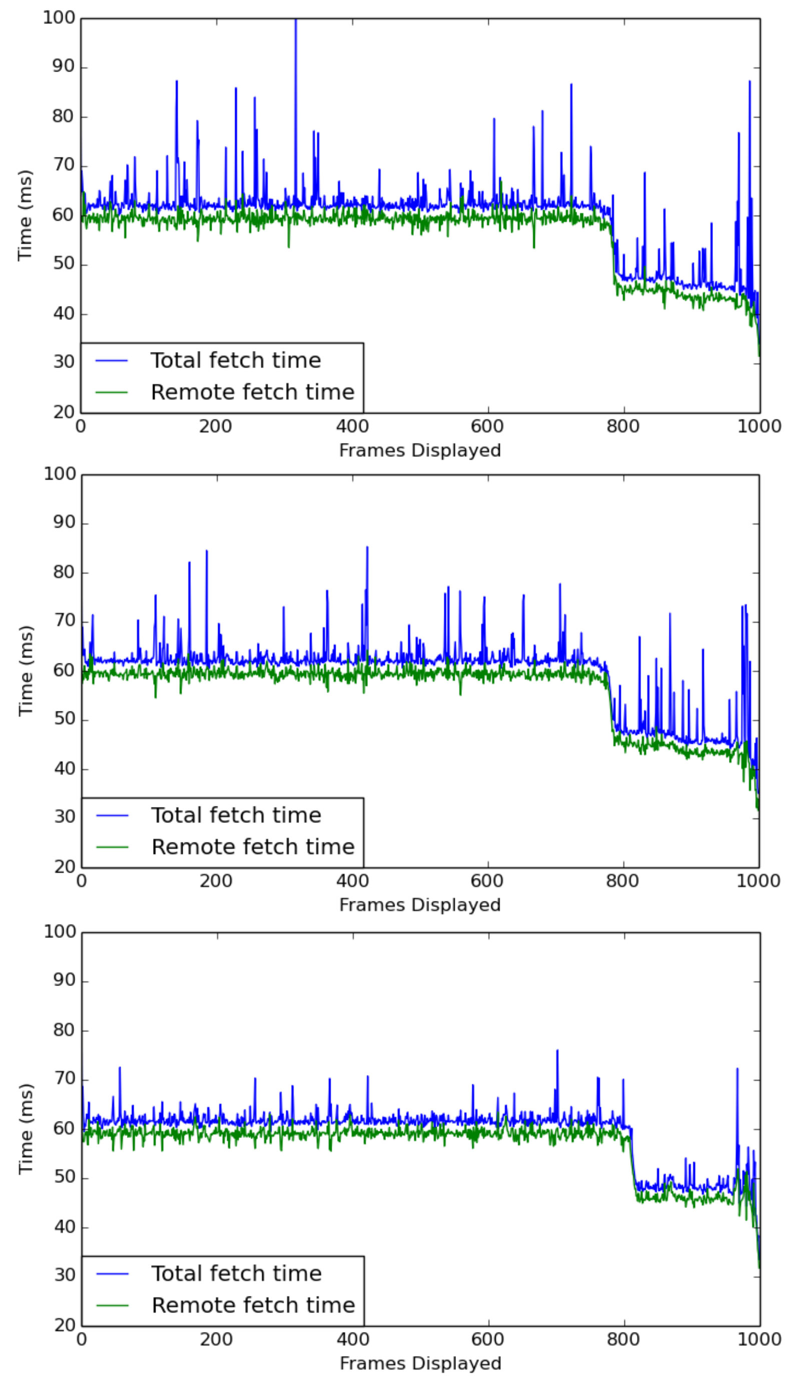

Figure 5.6: These three graphs show the total and remote fetch time with only the centralized cache server enabled using all 28 viewers. The first graph represents the image moving with the lowest speed of 5 p/f, the second with 10 p/f and the third with 20 p/f.

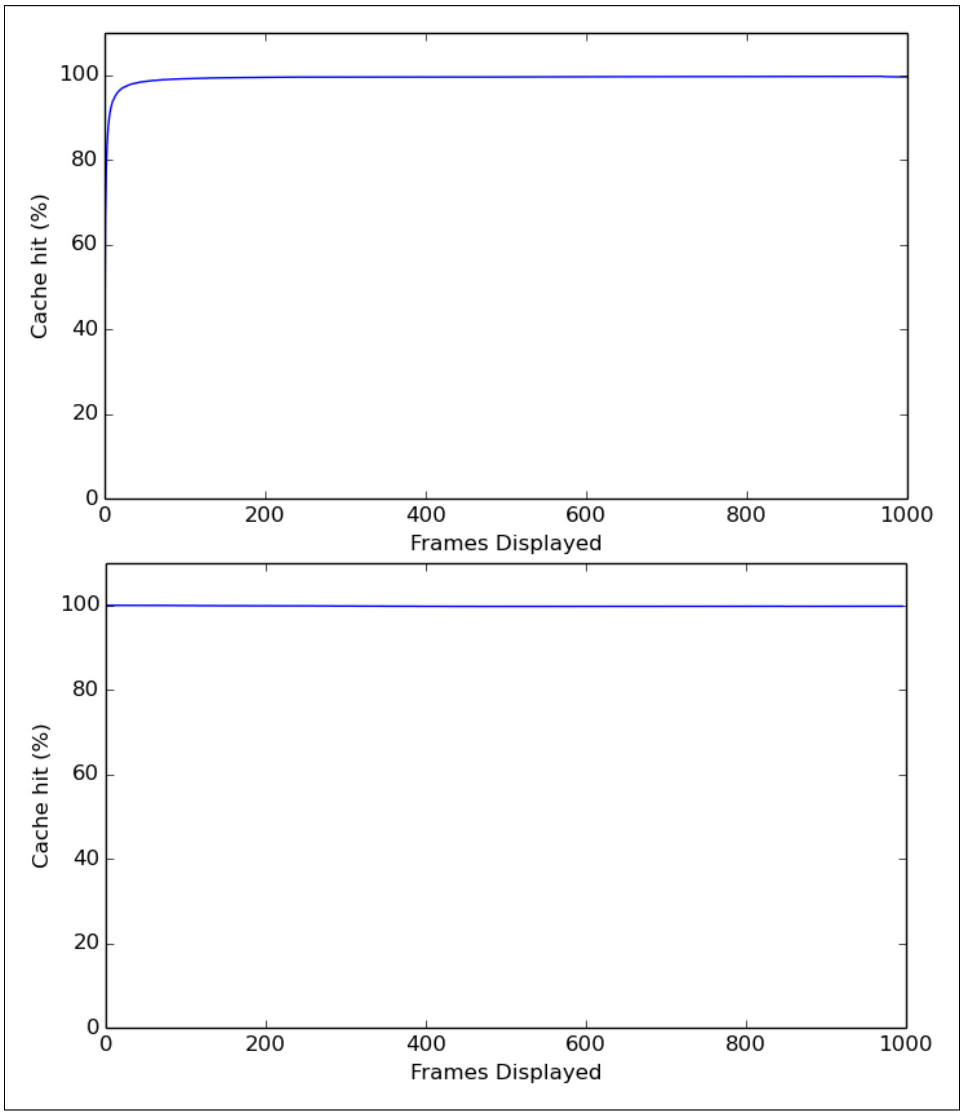

Figure 5.7: These graphs show the cumulative cache hit in percentage with only the centralized cache server enabled. The first one, without prefetching and the second with prefetching enabled.

5.4 Iteration two - multiple cache layers and cache server network flow

From the results of the first iteration, we can see that the fetch time grows rapidly from each cache layer. In advance, we expected the results from the fetch time measurement to give the best results when all cache layers are combined, but after a quick peak at the graphs, we see that the cache server has a fetch time way to high to give a good performance at all. As we already know how much time we can expect each cache level to use on cache hits, it would be interesting to find out how many requests goes to each of the cache layers.

As it turns out, the cache server uses a lot more time than expected when delivering images to the viewers. A closer look at the graphs reveals that most of the time goes into communication between the viewers and the cache server. Rather than combining this cache layer with the others, we will conduct other experiments on it in order to determine what makes this layer so inefficient.

5.4.1 Combining cache levels

In iteration one, we conducted experiments on each of the cache layers separately and because of this, the experiments resulted in 100% of the cache requests being received in each of the cache layers. When the cache layers are combined, we expect the frame cache to protect the local cache so that very few requests reaches the local cache. If the centralized cache was enabled too, it would be protected by both the frame cache and the local cache, resulting in even less requests being received. The ideal request flow would be the first cache level, or the one being most efficient with respect to the time used, delivering most of the requests. We will conduct experiments measuring the number of requests reaching each cache layer in order to determine how well the frame cache protects the local cache.

In these experiments we will use another input stream more suited for the usage of the local cache. The movement will still be in a rectangular shape, but we will use a smaller area to simulate many short movements over the same area.

As the frame cache will protect the local cache from a lot of the cache requests, the only requests received in the local cache will be the ones missing in the frame cache.

In the beginning, the local cache will contain little or no data and the first cache requests after the first image load will be registered as a hit in the frame cache and not in the local cache. Therefore, we expect the the local cache hit to decrease during this experiment compared to the cache hit measured from this cache layer separately.

By running the viewer three times with the frame and local cache enabled, increasing the move speed for each time, we collected the data into three graphs as shown below.

Greater move speed increases the data set, which affects both the cache layers. Requests sent to the frame cache will more frequently result in a cache miss, causing more requests to end up in the local cache. We can see this in Figure 5.8. If we compare the three graphs, we can see the hit ratio in the frame cache decreasing when the move speed is increased. The figure also shows the local cache's hit ratio being changed more often, as a consequence of the cache miss in the frame cache.

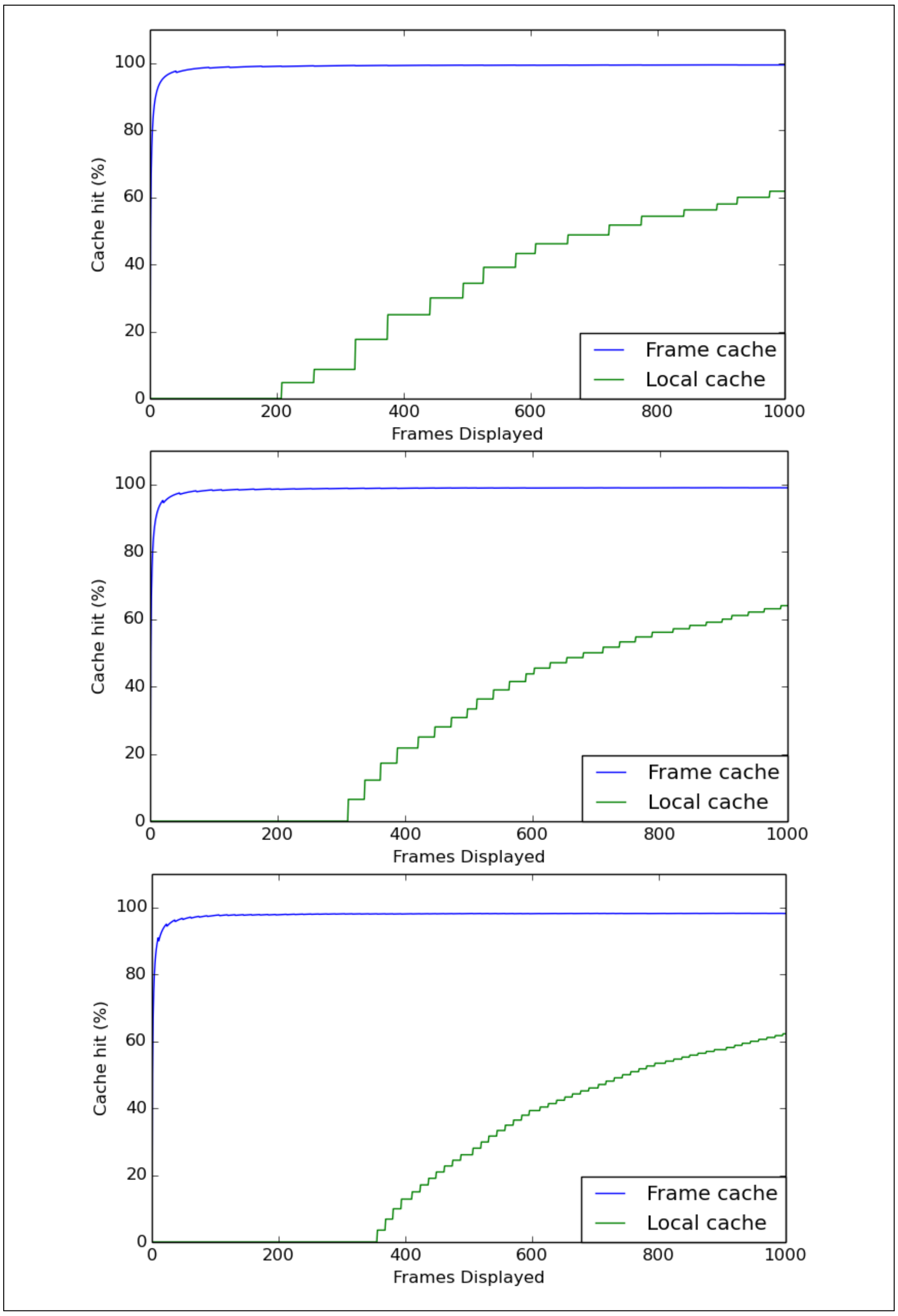

Figure 5.8: Cumulative cache hit in percentage with both the frame cache and the local cache enabled. The first graph shows the hit ratio when the image is moving with the lowest speed of 5 p/f, the second with 10 p/f and the third with 20 p/f.

5.4.2 Cache server network traffic

The cache server is now ruled out when combining cache layers for an optimal fetch time, but it would be interesting to see what makes the remote fetch time to increase when using this cache layer. We will conduct an experiment using the cache server only, where we measure the remote fetch time as we increase the number of participating viewers and requests sent. The number of open connections at once and concurrent request handling might affect the time used on sending the image from this particular cache level.

As the viewer and cache server run on different physical machines with unsynchronized clocks, without proper clock synchronization we cannot determine how much time is being used each way. Still, we assume most of the time being used on sending data to and from the cache server goes to transferring image data in the response, as the response has a much greater size than the request.

During this experiment, the remote fetch time was measured by using three different send techniques. One, where the request for each image fragment in the frames were sent sequentially, another where they were sent concurrently and a third where all image fragments in the frame was bunched into one request. The graph below shows how using these send techniques affects the remote fetch time when increasing the number of participating viewers.

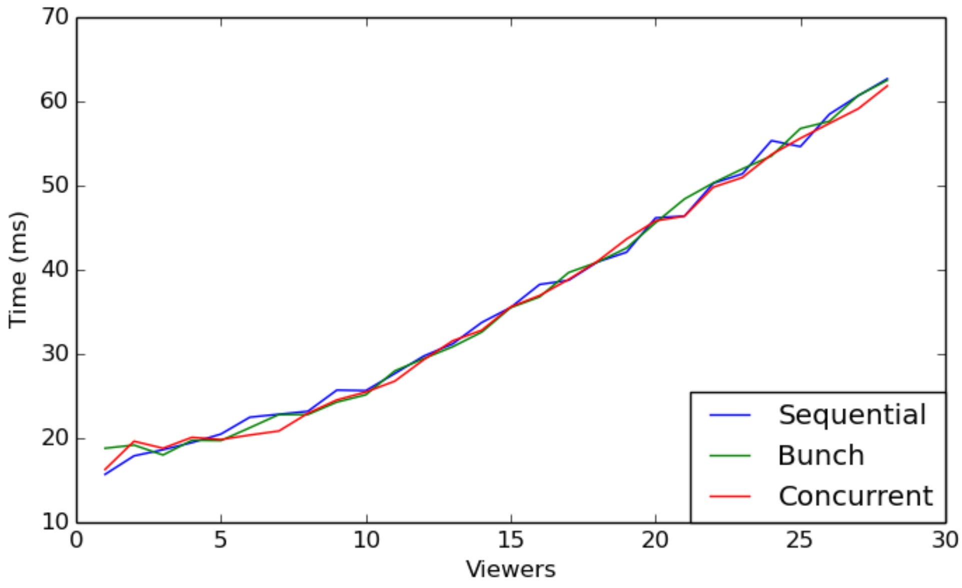

Figure 5.9: Average remote fetch time when sending requests sequentially, concurrent and bunching them while increasing the number of participating viewers.

As we can see from Figure 5.9, sending the image fragments one by one sequentially, concurrently or the whole frame at a time doesn't seem to differ in average fetch time used. What we do see is that using more viewers with more requests does in fact increase the remote fetch time. It increases to around 60 milliseconds when using all 28 viewers in the cluster. This is the same value as the time duration measured earlier using the cache server.

5.5 Iteration three - frame rate

The fetch time is a direct influence on the frame rate, as long fetch time results in few frames being viewed. In this final iteration, we will measure the frame rate in order to find out if the image viewer lives up to our expectations and desired need.

5.5.1 Frame rate

Until now, we have conducted experiment where the image viewer was run without synchronizing the viewers. In this experiment we will measure the frame rate with and without synchronizing the viewers and see if the performance changes. We will also compare the results to the frame rate measured from the baseline.

Figure 5.10 shows that using both the frame cache and the local cache gives a frame rate around 60 frames per second. Synchronizing the viewers seems to straighten out the graph and we see that using synchronization don't decrease the frame rate. The baseline gives a frame rate between 25 and 30 frames per second, which is barely half of the frames processed when applying the cache layers mentioned.

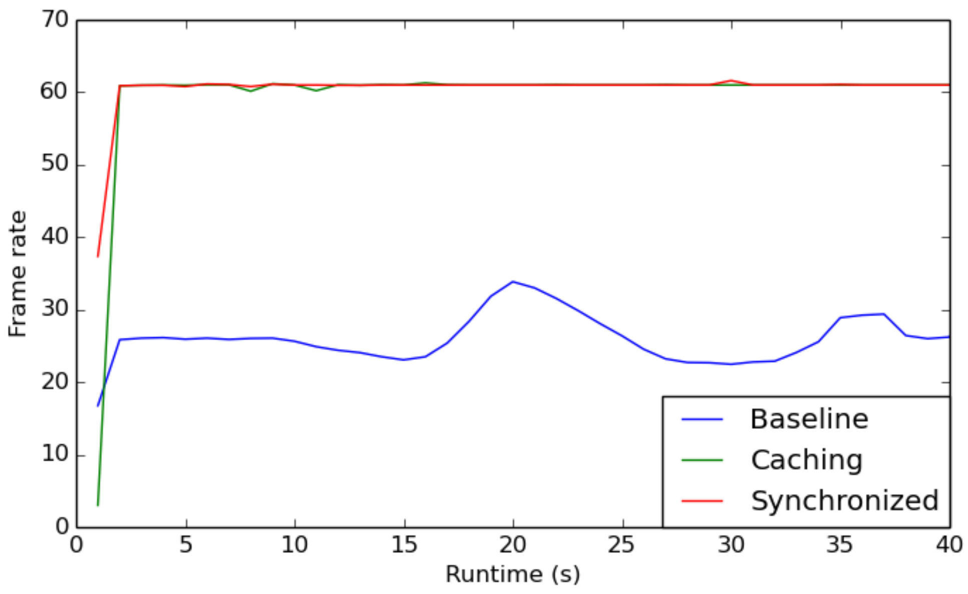

Figure 5.10: This graph shows the average frame rate during the runtime of the viewer. The image viewer was run three times. First, with the baseline enabled, second, with the frame and local cache enabled and third, by synchronizing the viewers with the frame and local cache enabled.

6 Discussion

Through the experiments and measurements explained in the last chapter we gained a lots of results, both expected and unexpected. During this chapter, we will take a closer look at the results and answer questions regarding why we get the results that we do, what they mean and whether they look promising or not. We will also discuss whether or not the results lived up to our expectations and if we got any surprises.

We will start by discussing the scenario giving the best results during the experiments, using both the frame and local cache. This section will contain some of the principles behind distributed systems and how the image viewer responds to them. Performance in the sense of throughput and latency, and fault tolerance are subjects we have chosen to focus on.

6.1 Optimal cache layer combination

In chapter 5, we determined that enabling both the frame cache and the local cache gives the best results regarding fetch time, which directly affects the frame rate. When it comes to the cache hit ratio, the cache server had the advantage of prefetching and was thereby the cache layer giving most cache hits. Still, 100% cache hit doesn't help if we are not able to retrieve the cache data fast enough. Both the frame cache and the local cache also have a very high hit ratio with 96% and 97%. The most critical point seems to be at the initial frame where both cache layers are empty. This is expected as prefetching is not implemented for any of these cache layers.

6.1.1 Performance

In this project, the performance is measured by the frame rate the image viewer is capable of deliver. As the client initializes the image viewer with the desired frame rate, the experiments done to the image viewer were all executed with the highest frame rate at 60 frames per second. As we saw from the measurements, the image viewer is capable of holding the given frame rate both with and without synchronizing the viewers.

By looking at the results from the baseline-, fetch time- and frame rate experiments, we can see that applying these cache layers to the application increases the performance, making the image viewer twice as efficient.

Throughput

In this application, the throughput is determined by the frame rate, the frame size and the size of the image fragments. The image viewer processes about 33600 image fragments each second. This happens when the image viewer has 28 participating viewers with a frame rate of 60 frames per second, where a viewer's frame contains an average of 20 image fragment. This translates to approximately 520 MB of image data per second, as the images have an average size of 15,5 kilobyte (KB).

When comparing the baseline up against the image viewer, with the frame and local cache enabled, the frame rate is doubled. As a consequence, the throughput is being doubled too.

By taking a closer look into the measurements regarding the fetch time, it uses an average of 150 nanoseconds in the frame cache and 250 microseconds in the local cache. Assuming all cache requests results in a cache hit, the frame rate should become much greater than 60 frames per second. After some research and a talk with some of the members of the HPDS group, it turns out that the display wall uses Vertical Synchronization (VSync). This is a software component limiting the graphics card so it doesn't increase the frame rate more than the refresh rate of the projectors[31][32]. If the frame rate grows larger than the refresh rate, the risk of screen tearing increases, which makes VSync a good idea as the projectors' refresh rate is around 60 Hz. As VSync limits the application's frame rate to 60 frames per second, we would have to turn off VSync in order to benchmark the image viewer beyond this limit. We have not focused on doing so as there are no need for the image viewer to run with a frame rate greater than 60. However, it would have been interesting to measure the application's limit regarding the throughput.

Latency

In our system, latency occurs from the moment the client interacts with the web interface, to the corresponding frame is displayed on the display wall. During this project we have not measured the latency, partly because we have not observed a noticeable latency being built up when running the viewer and partly because we wasn't sure how to measure it. The latency could be measured by making the viewers send a "done" message to the web interface after drawing a frame and let the web interface measure the time used from a state is sent until a response is received. The danger of measuring the latency in this way is that the time duration might be incorrect as it takes some time to transfer all done messages from the viewers to the web interface. Also, as we have experienced with the centralized cache server, 28 viewers spamming a centralized unit with requests might result in a huge latency itself. Even if the data size is only a fraction of the size of a fetch response, receiving 28 responses 60 times a second makes 1680 responses per second, which might create an incorrect latency measurement.

In the capstone project, the frame rate was limited to 30 frames per seconds in the experiments. As the image viewer couldn't always deliver a greater frame rate, in order to prevent a huge latency, the viewer was implemented using buffered channels for communication between the goroutines receiving states and fetching image fragments. In this way it was able to drop states if the buffer was getting full.

The image viewer developed in this project is at the moment implemented using unbuffered channels, not dropping states. This, to prevent screen tearing between the viewers by the state dropping. In advance, we were hoping for the viewer to be able to deliver a high throughput by applying the designed cache layers and measure via the experiments whether or not this was the case. If it should turn out from the results that the system was not able to deliver the desired frame rate, we would have to change the implementation and maybe the design. As we have seen from the measurements, the image viewer is able to deliver the desired frame rate, processing all incoming states.

6.1.2 Fault tolerance

Some times when the image viewer was used on all display tiles in the cluster, some of the viewers' viewports were black under the whole run. It was determined that this problem was caused by faulty projectors and not the compute nodes, meaning the image viewer did work as it should, except some of the display tiles could not display their image. This is a perfect example of how vulnerable a distributed system can be, as it just as well could have been some of the display tile machines crashing, e.g. because of overheating. As this problem relies in the architecture of the display wall and not in the image viewer, it's little we can do about it.

Often when a node goes down in a distributed system, to deal with the problem, another node will take its place. In this architecture that is not possible, as a projector is connected to one computer node only. As there are no other machines physically connected to the projector, if a computer node goes down, no other will take its place. Considering this architecture, we created a design in which the viewers running at the computer nodes are loosely coupled. If a viewer goes down because of a faulty computer node, the other viewers will still be running and not even notice that the viewer disappeared.

We can, however, improve our application with respect to fault tolerance by eliminating single point of failures. As the cache server is out of the picture, we are left with two single point of failures in the image viewer's design pattern. That is the backend server hosting the web interface and the coordinator, forwarding the states to the viewers.

Both the backend server and the coordinator run on the frontend node in the rocks cluster and if this computer node goes down, none of them will be available. To access the compute nodes, we must go through the frontend node. As the backend server must be kept on the frontend node in order to be able to provide the web interface, we cannot prevent this single point of failure. The coordinator, however, do not have to run at the frontend node. It could be placed at one of the 28 compute nodes, but as the coordinator causes both network traffic and some CPU processing, it might slow down the viewer running at the same compute node and thereby the whole image viewer. In order for the coordinator to not affect the viewers by running on the same compute node as any of them, we were forced to keep it on the frontend node.

Still, an improvement could be to let the backend server detect whether or not the coordinator is alive and if it's not, start a new coordinator process. This could work in a case where the coordinator experiences an error, making only the coordinator process to stop. If the coordinator makes the frontend node to crash, this idea would not work.

6.1.3 Scalability

By manually using the application and through automated benchmarks, we see that it scales from using one display tile to the maximum limit at the cluster, 28. We were not able to increase the amount of active viewers any further, but we do expect the coordinator to become a bottleneck when scaling the system to include a large number of viewers. We believe that the broadcast implementation is not efficient enough to be used by a high number of participating viewers and that it will slow down the application. In order to deal with this potential problem, we could deploy another broadcast design and implementation. Letting the coordinator broadcast to a set of viewers, which again broadcast the states to the rest could be an approach to consider. The downside of using this approach is that the viewers are no longer loosely coupled. If a viewer receives a state and crashes before it reaches to broadcast it, the other viewers waiting for this particular state will never get it.

6.2 Centralized cache server

From the results of the fetch time experiments, we noticed that the cache server's fetch time decreases when zooming out, compared to the other two cache layers where it increases. This is caused by two factors. One, as we pointed out earlier, some viewers completes the input stream before others, making some viewers to send fetch requests when there are less load and less requests at the cache server. Two, as the zoom movement changes the size of the image fragments, fewer and fewer fragments are needed in each frame, resulting in even less fetch requests sent to the cache. The time used on changing the size of the textures goes unnoticed in this cache layer as it is very small compared to the fetch time, which brings up the question of how we thought this would be efficient in the first place.

The main idea with the centralized cache server was that each image fragment loaded upon a request from one viewer would be available in the cache for another. As the image is moving across the screen, it is likely that more than one viewer would have the need of displaying the same image fragment during the movement. In advance we thought a centralized cache server could work, as the hit ratio would be very high and all machines are connected through Gigabit Ethernet. It turns out that the high hit ratio doesn't weigh up for the number of open connections frequently transmitting great amounts of data. As we compare the remote fetch time to the fetch time achieved in the baseline experiment, it would be more effective to load the images directly from the NFS.

6.2.1 The problem

From the results of the experiments regarding remote fetch time, we see that using a single centralized cache server might not be the way to go. Too much time goes into sending the image fragments from the cache server to the viewers. After looking at Figure 5.9, we see that the remote fetch time increases with the number of participating viewers, but even if we are using one single viewer, it still takes between 15 and 20 milliseconds before the viewer receives one image fragment. By enabling prefetching, the cache server achieved 100% cache hit during the whole lifetime of the experiment, but as the problem does not lie in whether or not the request is a hit or miss, it doesn't make a difference what the hit ratio is.

To be exact, the problem is that it takes too much time sending image data over the network with a centralized architecture where the cache server is using concurrent request handling. As we have already tried bunching image fragments, sending them sequentially and concurrently without any difference in time, we will have to consider other design patterns and implementations for achieving an efficient level three cache. There are several alternatives to solve this problem and we will now discuss some of them.

6.2.2 Alternatives

As the image viewer performs well without the cache server, we might not need a third cache layer. Still, a miss in both the frame cache and the local cache will result in loading the image fragment from the NFS and as we see from the baseline, this gives an average fetch time of 5 milliseconds. If we want to improve the fetch time in such a case, we could consider some of the following approaches.

Distributed cache layer

One approach to the problem is the idea of a distributed level three cache. As each of the image viewers cannot store the entire gigapixel image, an alternative could be to implement a distributed cache layer, where each of the compute nodes store a small part of the gigapixel image.

In order to retrieve images from another compute node, communication like remote procedure calls could be used. For this approach to be a better alternative than the centralized cache server, we will need a distribution algorithm capable of distributing the image fragments to the various compute nodes. If the image fragments are not properly distributed, one compute node might receive the cache requests from all the others and we are basically back to a centralized cache server with an even smaller cache size.

Also, the gigapixel images compatible with the image viewer can be of a much greater size and contain a lot more zoom levels than the image used in our experiments. Therefore, we cannot let the distributed cache layer store all image fragments from the gigapixel image, as some gigapixel images might be of a greater size than the memory limitations on the cluster. This requires a solid replacement policy and prefetching might be a great idea in order to improve the cache hit ratio and fetch time.

Coordinator as prefetch detector

In advance, the programmer was thinking of an approach where the viewers were prefetching image fragments into their local cache layer based on a prefetch command from the coordinator. In this way, there would be no need for a third cache level, as the local cache would always have a cache hit of 100%. The problem with using the coordinator for determining the need of prefetching in this approach is that the prefetch request might not be received by the viewers early enough, risking the coming cache request to result in a miss anyway. In order to use the coordinator as a prefetch detector, we might have to use another prefetch algorithm based on guessing which direction will be the next. If we assume the current move direction will remain over multiple frames, we will probably have guessed right a lot of times, but in the cases when the direction changes, this method of prefetching will fail.

Viewer based prefetch signalling

Another approach could be to let each viewer signal its neighbours to prefetch a set of image fragments. E.g. if the image moves to the right, all viewers could send a prefetch signal to the viewer on its right side. In this simple example, each viewer will have to send one prefetch signal only. Using this principle as a foundation, we could add support for all six directions over three axes.

Unfortunately, this approach also have some problems that prevents us from achieving 100% hit rate. If we look a the example where the image moves to the right, we see that the viewers most to the left are not receiving any prefetch signals. Whatever direction the image moves to, some viewers will not get the prefetch signal. Running fullscreen at the Tromsø Display Wall, a move direction along the x-axis will result in 1/7 viewers to not receive the signal. Moving in a direction along the y-axis 1/4 will miss the signal and in the z-axis all viewers will miss it.

Also, as we pointed out in the previous section, the viewers will not be able to send a prefetch signal based on the current incoming state early enough. Therefore, we will have to use another prefetch algorithm. Even if this approach cannot achieve 100% cache hit in all cases, it still would be better to deploy it to our current implementation, as it often would give us cache hits where we otherwise would have load the image fragment from the NFS.

6.3 Problems, bugs and errors

During this project we faced some problems, which not all have been solved yet. Some of them have been small and of minor importance for the performance, and others have made the image viewer a danger for the display wall cluster. We will go through some of them and give a brief description of the problem and how we solved it.

6.3.1 Memory leak

During the experiments when using only the cache server without prefetching and any other cache layers enabled, there was discovered a major memory leak. Even if the cache server was limited to only contain a maximum number of 2000 minisurfaces at this point, it used more memory than it was allowed and during a few minutes it had used 100% of the cluster memory.

At first we thought the cause was due to the fact that a single line of code was commented out, as a result from an earlier debugging session still remaining in the source code. It would have made sense, as this particular line of code was responsible for inserting the loaded minisurfaces into the cache map. As only the cache server was enabled with no protection against cache requests from the other two cache layers, it would receive all cache requests from all 28 participating viewers.

Like we have discussed earlier i this chapter, the image viewer has a high throughput with around 520 MB of data each second. Go's garbage collector runs periodically each other minute[22]. During these two minutes, the cache server would have tried to load about 62,4 GB of data into the memory and we was sure the missing line of code made the loaded image fragments stay in memory until the garbage collector freed them.

Unfortunately, after inserting this line of code again, the cache server did not change its behaviour. By using the built-in profiling tool in the go programming language, it turned out that the memory leak was caused by not properly closing the websocket connections at the server side. As the cache server manages connections and requests concurrently, by not closing the connections between each frame, the amount of open websocket connections grows quickly and over time, it would would require more memory than available. After closing the connections properly and ending the goroutines responsible for this, the memory usage did not exceed the given memory limit.

6.3.2 To fast prefetch rate

After the experiments were done, a bit later we were using the image viewer manually over a long period of time and by surprise, we discovered that the centralized cache did not have a cache hit of 100%. The hit ratio decreased over a longer time period because the cache server was prefetching image fragments too fast compared to how often the viewers requested them. A while after this cache was filled, the cache server started to replace image fragments that was still in use by the viewers, believing their location was outside the current viewport. When the viewers then requested these image fragments, the result was always a miss. This was not discovered in the experiments, because they did not last long enough in time.

To deal with this problem, we could have adapted the prefetch rate at the cache server in order to wait for all viewers to finish viewing the frame before prefetching a new one, but as the experiments gave the results they did regarding the cache server, we did not spend time trying to correct this bug.

6.3.3 Shaking zoom

There appears to be a bug in the implementation of the image viewer's zoom movement. As we zoom in and out, the image is moving back and forth just a bit, creating a shaking effect. As the screen of the mobile device, either phone, tablet or laptop, are smaller than the display wall, the image's position has to be scaled up in order to match the same position at the display wall.

We expect there to be a bug in the translation of the image position as a "misplacement" of just a few pixels on the mobile device would be of a greater magnitude when scaled. Such a miscalculation would be capable of creating this kind of effect.

This issue has not been solved yet, as we have chosen to keep our focus on the image viewer itself, it's design, implementation, experiments and results.

6.3.4 Synchronize rate

As we have described earlier, the viewers are synchronized by the coordinator once after each frame. We can, however, change the synchronize rate by choosing how many frames to forward to the viewers before synchronizing them. For a while, we did try to increase the amount of states being forwarded before synchronizing, but in some cases when moving the image across the screen, we ended up with the image disappearing and coming back, creating a blinking effect.

Synchronizing the viewers once per frame might be a bit more than needed, but because of the good results we got from the experiments, we kept this synchronization rate to ensure that screen tearing does not happen.

7 Conclusion

Through this project, we have researched design patterns for a gigapixel image viewer controlled by mobile devices via a web interface. A prototype was implemented from a chosen design pattern, which have been evaluated through experiments and measurements.

The image viewer prototype is capable of delivering a frame rate of 60 frames per second while running fullscreen with 28 viewers, where the viewers are synchronized. The results from the experiments shows that using two cache layers, the frame and local cache, gives the best performance with regards to the fetch time and as a direct "consequence", the frame rate.

The latency was not measured, but by manually using the application, we don't notice a delay or a latency being built up over time. With this, and through the experiment results, we make the conclusion of our application to have high throughput an low latency.

As the image viewer receives a input stream consisting of the current state of the image position, other interaction systems, such as gesture systems, are supported as they can periodically send the image position to the gigapixel image viewer.

We are happy to say that the end result solves the problem definition and that we are satisfied with the results given by the evaluation of the prototype during the limited time frame of this thesis.

7.1 Concluding Remarks

The development of this application was time consuming as the application was developed without a proper Integrated Development Environment (IDE), nor the opportunity to build the application on the laptop used under the implementation.

There exists a plug-in for the go programming language in the Eclipse IDE, but this was discovered a bit too late in order to have any use for it during the development of the prototype.

Without the opportunity to build the source code on the same machine as it was written, the developer pushed the code into a git repository from his laptop and pull it to the display wall cluster in order to compile it. The compute nodes at the cluster could be wiped out at any time without notice and therefore the cluster could not be used as a storage alone. This was inconvenient as the code had to be compiled quite often and thereby, one of the causes to why the implementation took so long.

Building go applications can be done using a single command. As our application contained multiple components, supposed to run on different machines, building the whole application requires more than one build. Therefore, python was used to create a make script building the different components.

7.2 Future Work

Although we are proud of the result gained and that Giga-View solves the given problem definition, we still have some improvements that would make the application better. Some of them would improve user experience and others would improve the performance and cache hit ratio.

7.2.1 Prefetching

In section 6.2.2, we mentioned some alternatives to a centralized cache server. We believe some of these suggestions would improve the performance by increasing the hit ratio and thereby improve the average fetch time even more. As we ended up with an application not using prefetching, which conflicts with the ideas we had before the project, we would like to see this concept being used in the future.

7.2.2 Multiple users

In order to further develop this application, multiple users should be supported. At the moment, the image viewer only supports one user at a time, viewing only one image. As a user may not run the image viewer on all display tiles, some would be free to use by others and this suggestion should therefore be considered under further development.

7.2.3 Extend flexibility

The implemented prototype is currently not integrated with Display Cloud. Neither are on the fly reconfiguration or on the fly switching of images supported, but the design of the image viewer has been made in such a way that it is possible to integrate with these features later. By implementing these features, the application will be far more flexible for use at the Tromsø Museum and the UiT.

7.2.4 Web interface - image tiling

For the moment, the image displayed in the web interface at the client is not exchanged when reaching new zoom levels on zoom interaction. The result is a image with small resolution being stretched into a size 2^n times greater than the original at the n-th zoom level, making the whole image so blurry that you cannot see what's in it. During this project, we have not focused on improving the user experience by tiling images in the web interface. This is a task we leave as future work as the time frame of this thesis was limited.

Bibliography

[1] Anshus, O., et al. "NineYears of the Tromsø Display Wall." Proceedings of Powerwall, SIGCHI Workshop. 2013.

[2] Tiede, Lars, John Markus Bjørndalen, and Otto J. Anshus. "Cloud displays for mobile users in a display cloud." Proceedings of the 14th Workshop on Mobile Computing Systems and Applications. ACM, 2013.

[3] Aydin, Galip, et al. "Building and applying geographical information system Grids." Concurrency and Computation: Practice and Experience 20.14 (2008): 1653-1695.

[4] Google Maps explained "Traffic Analysis on Google Maps with GMaps-Trafficker", [Online]. Available: http://www.ioactive.com/pdfs/SSLTrafficAnalysisOnGoogleMaps.pdf. [Accessed May 25th 2015].

[5] Liu, Zao, et al. "Implementing a caching and tiling map server: a web 2.0 case study." Collaborative Technologies and Systems, 2007. CTS 2007. International Symposium on. IEEE, 2007.

[6] Haklay, Mordechai, and Patrick Weber. "Openstreetmap: User-generated street maps." Pervasive Computing, IEEE 7.4 (2008): 12-18.

[7] Kopf, Johannes, et al. "Capturing and viewing gigapixel images." ACM Transactions on Graphics (TOG) 26.3 (2007): 93.

[8] Ponto, Kevin, Kai Doerr, and Falko Kuester. "Giga-stack: A method for visualizing giga-pixel layered imagery on massively tiled displays." Future Generation Computer Systems 26.5 (2010): 693-700.

[9] Yamaoka, So, Kai-Uwe Doerr, and Falko Kuester. "Visualization of high-resolution image collections on large tiled display walls." Future Generation Computer Systems 27.5 (2011): 498-505.

[10] Powell, Mark W., Ryan A. Rossi, and Khawaja Shams. "A Scalable Image Processing Framework for gigapixel Mars and other celestial body images." Aerospace Conference, 2010 IEEE. IEEE, 2010.

[11] GIGAmacro "Exploring Small Things in a Big Way", [Online]. Available: http://gigamacro.com/. [Accessed May 28th 2015]

[12] Wallace, Grant, et al. "Tools and applications for large-scale display walls." Computer Graphics and Applications, IEEE 25.4 (2005): 24-33.

[13] Stone, Maureen C. "Color and brightness appearance issues in tiled displays." Computer Graphics and Applications, IEEE 21.5 (2001): 58-66.

[14] Stöckli, Reto, et al. "The Blue Marble Next Generation-A true color earth dataset including seasonal dynamics from MODIS." Published by the NASA Earth Observatory (2005).

[15] Grammeltvedt, Åsmund. "Frag: A distributed approach to display wall gaming." (2006).

[16] Beaver, Doug, et al. "Finding a Needle in Haystack: Facebook's Photo Storage." OSDI. Vol. 10. 2010.

[17] Huang, Qi, et al. "An analysis of Facebook photo caching." Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013.

[18] Go, "The Go Programming Language", [Online]. Available: https://golang.org/. [Accessed May 6th 2015].

[19] The Go Programming manguage "The Go Project", [Online]. Available: https://golang.org/project/. [Accessed May 25th 2015]

[20] Go documentation "GoDoc", [Online]. Available: https://godoc.org/. [Accessed May 25th 2015].

[21] Go-gl, "Go bindings for OpenGL", [Online]. Available: https://github.com/banthar/gl. [Accessed May 6th 2015].

[22] Dave Cheney "Visualising the Go garbage collector", [Online]. Available: http://dave.cheney.net/2014/07/11/visualising-the-go-garbage-collector. [Accessed May 26th 2015]

[23] Flask, "Flask - web development, one drop at a time", [Online]. Available:

http://flask.pocoo.org/.[Accessed May 6th 2015].

[24] jQuery, “jQuery API”, [Online]. Available: http://api.jquery.com/. [Accessed May 6th 2015].

[25] LocalStorage, “Window.localStorage”, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. [Accessed May 6th 2015].

[26] OpenGL, “OpenGL tutorials”, [Online]. Available: http://www.opengl-tutorial.org/. [Accessed May 18th 2015].

[27] OpenGL, “Introduction”, [Online]. Available: https://open.gl/. [Accessed May 18th 2015].

[28] “Parallel OpenGL FAQ”, [Online]. Available: http://www.equalizergraphics.com/documentation/parallelOpenGLFAQ.html [Accessed May 26th 2015]

[29] OpenGL, “OpenGL and multithreading”, [Online]. Available: https://www.opengl.org/wiki/OpenGL_and_multithreading. [Accessed May 26th 2015]

[30] Apple developer, “OpenGL and Concurrency”, [Online]. Available: https://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_threading/opengl_threading.html. [Accessed May 26th 2015]

[31] Geforce, “Adaptive VSync”, [Online]. Available: http://www.geforce.com/hardware/technology/adaptive-vsync/technology. [Accessed May 26th 2015]

[32] Tweakersguide, “Graphics Settings - Frames Per Second”, [Online]. Available: http://www.tweakguides.com/Graphics_5.html. [Accessed May 26th 2015]