Concurrency in a distributed Gigapixel Image Viewer

Einar Kristoffersen
einar.kristoffersen@uit.no
University of Tromsø
Tromsø, Norway

Contents

1 Abstract

During the last 10 years the HPDS group of the computer science department at the University of Tromsø has developed the Tromsø Display Wall - a wall size tiled display that contains several displays and computers.

The gigapixel image viewer:

Limitations: As the current implementation of the Gigapixel image viewer at the TromsøDisplaywall is hard to maintain and... the need for a new design is growing. This technical report will describe the new concurrent and flexible design of the Gigapixel image viewer.

lessons learned...

2 Acknowledgements

dedicated to.. thank advisor.. thank family and friends.. thank technical staff working at the displaywall

3 List of figures

4 List of tables

5 Introduction

5.1 Overview

About the (Tromsø) Display Wall...

What is the image viewer?

Museum...

5.2 Description of problem

What is the problem I am solving?

bottleneck, unstable?, more efficient - concurrent, maintainable, configuration, display-cloud?

5.3 Motivation

What is the motivation for this project?

Maintaining the system, more effective - concurrency, As the current implementation of the Gigapixel image viewer at the Tromsø Displaywall is hard to maintain and... Because of this, the need for a new design is growing.

5.4 Contributions

A new concurrent design of the Gigapixel Image Viewer..

5.5 Requirements and limitations

What is the requirements and limitations of the design?

5.6 Outline

What will this report contain? In section..

6 Related Work

6.1 Introduction

6.2 others work on the wall

Get some papers...

9 years of the displaywall,

earlier design of the viewer

7 Technical Background/Theoretical framework

Talk about the technicalities... Golang, concurrency vs parallelism,

7.1 The Go programming language

Go is an open source project developed by a team at Google as well as other contributors from the open source community. It was designed to address the problems faced in software development at Google and became a great tool for engineering large software projects.

7.1.1 Why use Go?

As mentioned, one of the goals of this Project is to create a implementation that is maintainable. The current implementation of the image viewer is written by using a combination of several languages and as a result, it is hard to maintain the code. There was a discussion of which programming language would be best suited for the task. Both Python and Go was solid candidates in order to create a maintainable implementation, but the decision landed on the use of Go to perform this task. Unlike Python, (C and C++), the Go programming language is build for concurrency. Go has the advantage of goroutines..

7.1.2 Concurrency build on CSP

Both concurrency and multi-threaded programming have a reputation for being difficult to use. Partly because of complex designs such as pthreads and partly too much focus on low-level details such as mutexes, condition variables and memory barriers. Higher level interfaces enables much simpler code and have been more successful in the past. One of the most successful models for providing high-level support for concurrency comes from CSP, by Tony Hoare. Channels... This is why the concurrency is build on the idea of CSP...

7.1.3 Goroutines instead of threads

Goroutines is a structure of behaviour that makes consistency easy to use in Go. The idea is to multiplex independently executing functions, coroutines, onto a set of threads. When a coroutine blocks, the run-time moves the other coroutines on the same os thread onto a different, runnable thread so they can keep running and not be blocked. The programmer sees none of this, which is exactly the point and makes it very easy to use the interface provided by this language. The result, also called a goroutine, can be very cheap. There is little overhead other than the memory for the stack, which is just a few kilobytes. In order to make the stacks as small as possible, Go uses resizable bounded

stacks. Goroutines are given a few kilobytes, which in most cases is enough, but when it isn't the runtime grows and shrinks the memory for storing stack automatically. This allows many goroutines to run in a small amount of memory. Because of this, you can easy create houndreds of thousands of goroutines in the same address space. If a goroutine was just a thread, the system resources would run out at a much smaller number.
https://golang.org/doc/faq#goroutines

7.2 Concurrency

7.2.1 Concurrency vs. Parallelism

When people hear the word concurrency they often have a tendency to think about the concept of parallelism, a related but quite different term. In programming, concurrency is the organization of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. When you run a program implemented by the use of concurrency, the executing processes is overlapping each other. This means that operations by different processes can work simultaneously, but they doesn't necessarily start at the same time. Parallelism often refers to the technique of making programs fun faster by performing several computations in parallel, literally at the same time. This requires multiple computing units, while the concurrency often refers to techniques that makes a program more usable and requires at least one processing unit. Concurrency can be parallelism, but not the other way around. Concurrency more powerful than parallelism.
https://blog.golang.org/concurrency-is-not-parallelism

8 Design

8.1 Overview

The image viewer is designed with a master-slave solution. The master can either be the frontend node or an ordinary laptop and the slaves is the 28 tiles in the cluster. Images located on either the shared disk space or at a web server is fetched by the slaves into shared memory and blitted on a surface viewed by a projector on the master's command. The master listens for input from the user that changes the position variables of the composed image put together by each tile viewing several smaller images. The tiles used to view images is determined by configuration files.

8.2 Configuration

The viewer is configured using two configuration files. One for the master and one for the slaves. The master uses the configuration file to get the ip-addresses to all the slaves participating in the viewer. The viewer doesn't need to use all 28 tiles. It can use any number of tiles from 1 to 28, but the tiles should be physical neighbours in the total image on the wall.

The slaves all use the same configuration file. This file contains the id of each participating tile and their corresponding physical position in the configuration. This position is given to each tile regardless of its original position in the big picture, but by comparing it's position to the others in the configuration.

As long as they all are neighbours...

The configuration files can be located and fetched from the shared disk or from a given url. The master's configuration file is opened and read only once, before starting to send data to the slaves. The slaves' configuration file is opened and read once by each slave. This is done before they start to fetch images and it is necessary to manage the configuration as an initialization when the viewer is started.

8.3 The master

The master can run either on the frontend node or at any laptop on the same network as the slaves. It's main tasks is to:

1. receive input from the user
2. determine whether the input changes the position variables of the composed image
3. pack changed position variables into a state
4. multicast new states periodically to the slaves in the configuration

At startup, the master launches two infinite running goroutines, one receiving user input and another multicasting new states to the slaves. These goroutines use a unbuffered channel and one way communication to transfer packed states from the goroutine receiving user input to the goroutine multicasting the states.

To interact with the user the master opens an SDL-window constantly listening for keyboard input. The input changes the position variables, that is an x and y position as well as the zoom level. When this happens, these variables is packed into a state and inserted into the state channel. Since the state channel is unbuffered, this goroutine blocks until the state is fetched by the goroutine multicasting it.

The multicast goroutine, will periodically fetch states from the state channel and do a multicast operation. N times per second the state will be sent by a multicast-like operation. First time the master do this operation, it will fetch the configuration file to get a list of ip-addresses to the slaves. It will now do the multicast-like operation, which is actually a loop starting goroutines writing the state into http connections based on each of the slaves' ip addresses from the configuration file.

The benefit of the master's design is that while it multicasts a state fetched from the buffer it can still receive keyboard input. These operations happens concurrently This happens concurrently

receive input -¿ as result, change variables -¿ insert to buffer -¿ (fetch from buffer and start multicast) periodically

Where is the concurrency??

paragraph about data flow

8.4 The slaves

As the master sends the new states to all slaves not knowing which slaves will actually use this information, it is up to each slave to take this information in use. In short, their main tasks is to:

1. listen for states multicasted by the master
2. calculate the image's local position
3. determine if any images is located in it's bounds
4. fetch those images into memory
5. blit them periodically at their local position

As a consequence of the master's behaviour, the slaves is the ones doing the necessary calculation on the states in order to fetch the correct images and blit them on the right position.

When a slave starts up it first fetches the configuration file holding it's virtual position and looks it up by the use of it's unique id. Like the master, the slaves also launch two main goroutines, only they have a bit different task. One goroutine is constantly listening to a http connection for incoming data and the other is responsible for periodically blit images to a SDL window at a calculated position. These goroutines also use a unbuffered channel to transfer received states from the first goroutine to the second.

The first goroutine will always listen to the http connection and for each state it receives, the state will be put into the state buffer. Here it will block until the state is received at the other goroutine where it is unpacked into the position variables again.

The position variables form a global position of the composed image and by using the slave's virtual position from the configuration, it calculates it's boundaries for which the global position have to be inside in order to fetch an image.

E.g. if we set the virtual position of a tile to be (vx,vy) in the configuration, it's boundaries will be from (vx*1024-256, vy*768-256) to (vx*1024+1024, vy*768+768). In a case where the global position is inside this bounds, the local position is calculated. The local position is where the image should be blitted on this tile's SDL surface.

When we know that the position variables place images inside these bounds, the filename of the images to blit is generated. This is calculated using the position variables and for each filename generated, it will try to load the corresponding image into memory. These images will be inserted into a map using the filename as key and a pointer to the image as value. The map is of a certain size and when it is full, old map items must be freed in order to insert new ones.

Depending on the position variables, the filenames generated is not all valid. The protocol is to load the image anyway and if the filename is invalid, the pointer to the image is nil. This pointer is inserted to the map regardless of it's value, but only pointers not equal to nil will be blitted at it's local position. This position is calculated using the global position and offset constants to vary the position of each image.

The design of the slaves is concurrent in several ways. First, a slave can blit an image at the same time as it receives a new state from the master. The goroutine receiving states from the master only blocks as long as the gorutine blitting images is not ready to receive

the new state. Second, the goroutine blitting images is aclually not doing this job alone. Since an image can be blitted at any x position from -256 to 1024 (local position) and any y position from -256 to 768, this makes a maximum of 5 images in width and 4 images in height at a time. Since each image's position is the upper left corner and not the middle of the image, the goroutine always start 20 new goroutines. This is one for each of these potential images, so the calculation of the image's position, the fetching of the image and the blitting of each image can happen concurrently. Before updating a tile's sdl surface all images must be blitted and since different goroutines work with it's own image, all from the same state, we have to wait for all goroutines to finish before the update can take place.

8.5 Data flow

Whatever kind of input method is chosen, the received input will change the position variables kept at the master. Only when at least one of these variables are changed, they all will be marshalled/packed into a state. Every time a new state is paced it will be inserted into a buffer/queue (FIFO) where it will be sent out as a multicast to the slaves periodically.

Input rate The input rate is the amount of received keyboard inputs during one second. This is at its peak when the image is constantly moving...

Variable change rate The amount of times the position variables (x, y and z) is changed by the user input during one second will from now on be referred to as the "vcr" (variable change rate). Depending on the vcr is high or low, it might create a bottleneck in form of a queue before a state is multicasted to the slaves...

Send rate - muticast The send rate is the amount of multicasts the master can complete per second. As the master will send the current state of the position variables periodically, it will have a constant send rate as long as there is enough states. If the send rate is lower than the vcr, the send queue will be filled up and a latency will occur while the input thread is blocked. As the queue grows larger, the time from receiving user input at the master to the corresponding state is sent from the master also increases. Not dealing with this problem will result in a growing latency and (show graph) from tests... To deal with this problem we introduce a empty policy saying that if the a state is tried to be inserted into a full buffer or the buffer grows too large, we empty the whole buffer.

Blit rate Amount of blits per second/FPS. Constant FPS/given... Flow charts

8.6 Network flow

A big design choice was to choose which method the master should use to communicate with the slaves. Several methods was considered when making this choice.

The sequential solution is based on the master sending only one request to a chosen slave. Depending on how this solution is implemented, the slave forwards this request to some or all of its neighbours in vertical, horizontal or both directions. By introducing a rule about not blitting or forwarding a state that equals the last received state, a slave receiving two or more of the same request will not contribute to spamming the other slaves with the same request over and over again. To make this solution most effective the master should choose the tile that is closest to the center due to the configuration.

The forwarding of requests results in a queue of slaves where the first slave is the one receiving from master and the last one is the one furthest from the center. The slaves between the head and tail of the queue is a collection of the slaves forming the shortest route between these two. This approach is said to be sequential because the last slave in the queue must wait until all the salves before it has received and forwarded the request, one by one, to the next slave in the queue.

Unfortunately this approach comes with a few problems. Even if the different queues send their requests concurrently after the head has received it from the master, each slave must wait for the request until the slave above has forwarded it. With a display configuration of 7 tiles in width and 4 tiles in height (maximum amount of tiles), the amount of sends will be 4 after the head received the request from the master. This causes a big latency from master sends the request to the last slave receives is.

Another problem is because of the latency this approach causes, the spamming rule we introduced will only work to a certain degree. If a slave S receives requests from more than one neighbour, let's say A and B, then this can be the case: S receives the request r1A, followed by r2A and then r1B.

What happens is that the delay causes S to receive two different requests from A before the first request is received by B. Since the last request received before r1B was r2A, S will both blit and forward this request twice. Because of the concurrency, this is not a "clean" sequential solution.

A "clean" sequential solution would be to let the master send one request to one slave and let each slave forward only one request further. This was not even considered as an alternative since the last slave in the queue has to wait for all other slaves in the configuration to forward the request. Forwarding the request as much as 27 times would create a even greater delay before the last slave receives the request.

The Multicast solution is quite simpler. The master sends the same request to all slaves by looping over the ip addresses and starting goroutines sending the request to each of the slaves. This is technically not a multicast since the sending of requests is done concurrently, but it is a multicast-like solution.

This will still result in a little delay from the first to the last slave receives the request, but this delay is so little that other factors will have greater impact on the total delay from a request is sent to a request is blitted on the screen. E.g. network traffic at each of the slaves, latency from fetching an image or number of images to fetch.

A disk based solution would be to let the master write the state to a file on the shared disk space and let the slaves read it. In order for this to work, the master must have access to the shared disks, meaning the use of laptops as master is not going to work. Still, this is an interesting approach. All the slaves can open the same file at once for reading, but the master is the only one that can open the file for writing. By writing and reading only one state each time the file is opened and closed, this method would probably create a overhead in opening and closing of the file. To reduce this overhead, states can be written and read in bunches for a certain time period. This would mean that the master and the slaves don't have to open and close the file as many times as before. The problem that might occur with this solution is that since the states is bunched before written, there might be a delay from the user enters keyboard input to the changes is displayed on the wall. A side effect of this is that an image moving across the wall would move constantly for each state in the bunch read, then wait to read the next bunch and then move further. The moving of the image might create a start and stop effect that we don't want.

By comparing the methods mentioned, considering how the latency is scaling when adding more tiles to the configuration, the choice of method to use for data transfer over the network was the multicast solution.

8.7 Memory-cache

current solution - load all images into ram. what about cache? map holding sdl surface pointers to images. when map is full, free pointers on new inserts pre-fetching of images

9 Implementation??

The design section explains the most. Not sure what to write here. Do I need this section?

10 Tests/Evaluation

Do tests!

benchmark

11 Results

Present result from tests by the use of graphs, tables and figures...

12 Discussion

12.1 Unbuffered channel as buffer

over 900 loop cycles per second. Too many variable changes - ok to wait for a state to be received before sending another. If it turns out later that the send rate is too low and we want the master to send states more often, then we can change the design to use a buffered channel instead. When using a buffered channel we can set the size of how many items it can contain and the goroutine inserting items into the buffer will only block until the item is written into the channel.

Channel as buffer: Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

when a channel is full, the sender waits for another goroutine to make some room by receiving you can see an unbuffered channel as an always full one : there must be another goroutine to take what the sender sends.

12.2 Buffered channel

Variables: length of buffer/queue sendrate – how often should we let it send, FPS... empty the whole queue or just a bit?

12.3 Loss of states

We do this because we want the movement of the image to be as accurate as possible according to time with a minimal latency. We can afford to loose some states in this solution, compared to a solution where you send all states, obtain a huge latency over time and cannot afford loosing any states.

12.4 Why SDL?

In this project SDL was chosen as the graphical framework as the author already was familiar with it. Alternatives could be OpenGL where the graphical effects would have been better.

12.5 Limitations of the design

sdl vs openGl - gliding/smooth movement and smooth zoom input method - sdl - could be web socket based cache??

12.6 Single point of failure

This system has many ”single points of failure”. This is partly because of the master - a single node multicasting states changed by the user input - but also because of the tiles. If only one of the tiles i the configuration or a tile's projector goes down, the total image viewed will not be complete. This is possible failures that follows the design of the displaywall, not the viewer and cannot be dealt with creating a new viewer... not the master. what about decentralized solution? several masters? election-¿new master?

13 Further works

openGL?

- cache?
- displaycloud?

14 Conclusion

As we can see from the results and evaluation of the implemented design ... lessons learned...