App Streaming Bringing games to the weak client

Andreas Gustafsson

Degree project in
Computer Science
Second cycle
Stockholm, Sweden 2013
Master Thesis: App Streaming
Bringing games to the weak client

Author: Andreas Gustafsson || andreg@kth.se
Supervisor & Examiner: Johan Hästad || johanh@kth.se
External Supervisor: Fredrik Wallenius: fredrik.wallenius@accedo.tv

October 22, 2013
Abstract

This thesis focuses on the problem of AppStreaming which is: moving the execution of application logic to the server-side and forwarding the User Interface (UI) to a weak client. Also, this proposal will be evaluated and compared to a current solution. The target hardware for the solution is Amazons cloud-based server rental service and weak Digital boxes. Current solutions require a fair amount of server power, and a great deal of bandwidth. The solution proposed in this thesis focuses on a smaller range of applications than the current solution and as such makes assumptions about the nature of the applications and thus exploits these to increase performance. The solution was implemented proof-of-concept style and evaluated with good results. The applications chosen for evaluation (three different browser-games) had much lower requirements on the proof-of-concept implementation than the business implementation of the current solution. The proposed solution is a mix of several well known compression schemes with a few intuitive adjustments.
Sammanfattning

Appströmning - Att spela spel på klena klienter

Chapter 1

Background

1.1 The Smart TV

The Smart TV is to normal TVs what a Smart Phone is to the old "Stupid Phone". This means much higher connectivity to the Internet and a fully fledged operating system with an Application Programming Interface (API) for developers. The Smart TV market has not developed as rapidly as that of Smart Phones but it is on the rise in several countries. The capabilities of Smart TVs are similar to those of the early Smart Phones such as Applications (Apps) that are enhanced versions of the functionality of the old technology. An example of such are Apps providing video-on-demand (VoD). While VoD is not a novelty, the content delivered by these apps are substantially larger than the pay-per-view of yore. Hundreds, if not thousands, of episodes of TV-series and as many full-length movies now lie within arms reach. Most TV-networks have an App of their own, customized to present their own content as perfectly as possible. However this content is not limited to merely TV-series and movies. Included in most Smart TV default setups are a number of games, similar to those of the early- to mid-generation Smart Phones. These games are more popular than previously expected, and Smart TV manufacturers use these games to entice customers to purchase their Smart-TV when choosing a new TV-setup.[3]

Not all Smart TVs consist of a single device. Some consist of a regular TV and a Digital Box. In these cases it is the box that contains the "Smart" part of the setup. This box is often included for free when a customer purchases a subscription plan. As all "free" gifts, the operator wishes them to be as cheap as possible. As a consequence the boxes have fairly weak hardware, barely enough to decode the video stream from the broadcasts. This weak hardware is often not enough to properly run the games that as promised by the operator.[3]

1.2 Target Group

The target group of these games is not the hardcore gamer. The hardcore gamers will buy an Xbox, Playstation or equivalent console. For many years the "typical gamer" was a young male alone in a dark room. This view of gamers has changed during recent years as gaming have become more mainstream. A key factor to this is social media. The popular social media Facebook features many embedded games, such as Farmville.
These games rival that of Blizzard Entertainments World of Warcraft franchise\textsuperscript{1} when comparing number of players per year. The social media makes it simple to measure your own accomplishments with those of your friends. World of Warcraft (WoW) is densely populated with "hardcore" gamers, people who spend a substantial part\textsuperscript{2} of their spare time in the game. The same can not be said for games like Farmville. Farmville and such games are extremely light-weight and rudimentary when compared to games such as WoW. However these are much easier to access (free to play, no giant game-clients that must be downloaded and installed). These "lighter" games are so popular that the "typical gamer" has actually changed from the youth in his dark room to middle-aged women. These lighter games has not pulled gamers from the typical video game, but have instead created an entirely new target group. This is the target group for most games for Smart TVs.

1.3 The problem

One part of the problem is that the hardware of the Digital Boxes are very weak and may not support many of the games. Another part is that the games themselves may be entertaining to play a few times, but quickly fall out of fashion. Thus a need is born: the need to run apps on hardware that is not powerful enough and to circumvent the need to install each game. A naive solution could be to hide all the installation processes from the user and simply uninstall the app upon termination. This would result in loading times upon start-up and not all Digital Boxes have physical memory to spare for such a temporary installation. The solution to this problem is to move the execution of the game from client to the server and simply forward the resulting video stream to the Digital Box. Such solutions already exist, but they require a large amount of bandwidth and processing power. Alternatively, it is possible to reduce the requirements of such solutions by specifically designing the game for this purpose. In either case it becomes necessary to transmit the game state to the client over the Internet. Doing this uncompressed is impossible and even conventional compression is not viable. The focus on this thesis is therefore to minimize bandwidth consumption, as bandwidth consumption is the first road block. The second road block would be to make the server side as efficient as possible, which will not be given as much consideration as the bandwidth consumption in this thesis.

There are a few differences between current solutions and the solution proposed in this thesis. Current solutions focuses on higher quality games whereas solutions proposed in this thesis focus on a subset of very specific types of games. The purpose behind this focus is to be able to exploit some inherent properties of these games. Furthermore, customers are not willing to pay much for these games nor are they intended as a raw source of income. These games are used as an extra means to compete with rivaling Smart TV suppliers. Therefore this heavy consumption of bandwidth and processing requirements of current solutions may not be economically viable.

\textsuperscript{1}As of writing there are approximately twelve million active subscribers to World of Warcraft and many more temporarily disabled.

\textsuperscript{2}6-8 hours a day, more on weekends is not as uncommon as one might think.
Solution: A method to reduce the bandwidth and processing power requirements of forwarding the visual state of basic to medium-style flash games from the server to the (weak) client. The solution should not introduce loss of perceived performance, but may do so if necessary as long as the loss is kept within reasonable boundaries. The method will be evaluated against the current solutions regarding performance and economic viability.

To summarize, the expected outcome of this project is to produce a result in three parts:

1. A solution to efficient AppStreaming of low-end web-based games (i.e. flash games in the range of Tetris to Angry Birds), and an implemented proof-of-concept.

2. To investigate of the economical cost of the proposed solution.

3. A similar evaluation of other currently existing solutions\(^3\).

1.4 Current solutions

The Estonian company Streamtainment is developing a system for streaming games to weak clients. Their target games are of significantly higher quality than the games this thesis focuses on. Their solution is at least a year in the making and access has been given to it in order to measure and evaluate its performance. Performance is measured as the balance between frame rate and the server-footprint per user. Even though the games is of a higher quality than those that are the focus of this thesis, the Streamtainment solution is interesting to compare with the solution proposed in this thesis. Should the proposed solution be cheaper to deploy and yield similar results to Streamtainment’s solution, it would be worthwhile to continue to develop the proposed solution.

---

\(^3\) Only if access to said solutions is given.
Chapter 2

Methods

The following methods were investigated. Some were combined into the final implementation. Some were discarded and some were only partially used.

2.1 Chroma Subsampling

Chroma Subsampling is a method to compress images by reducing the resolution of the colour channels. This works because the human eye is more sensitive to variances in luminescence than variances in colour. It is actually possible to remove up to half of the colour information in both channels without any significant loss of image quality.

The process of Chroma Subsampling consists of two phases. First the image is transformed from the RGB (Red/Green/Blue) format into Y’CbCr. Y’ is the alpha channel, Cb is the blue/yellow channel and Cr is the red/green channel. This phase is the time consuming one. Each pixel in the image decomposed into its RGB components, one byte each. These components are then converted into a Y’CbCr pixel. The second phase is the subsampling process. Each channel is divided into blocks of pixels, commonly 4x2 blocks but 2x2 blocks is also viable. The Y’ channel is usually not subsampled at all since such a loss of resolution would result in a significant loss of image quality. The Cb and Cr channels are subsampled with some predetermined ratio, for example keeping two of the four samples. The subsampling ratios are commonly denoted as J:a:b where J is the width of the sampling horizontally, a is the number of samples in the Cb channel and b is the number of samples in the Cr channel. Usually when Chroma Subsampling is used, the discarded samples are not entirely discarded. Instead the mean value is used. Consider a row of four pixels ABCD from which two should be sampled. The two samples would then be E=mean(A,B) and F=mean(C, D). When reconstructing the image, this row would be EEFF. This row would, when combined with the luminescence channel in its full resolution, be perceived as close to identical to the original. Examples of some common subsampling ratios is illustrated in fig. 2.1 and the effects of subsampling the various channels in fig. 2.2.
2.1. CHROMA SUBSAMPLING

Figure 2.1: The uppermost row is the Luminosity channel, the middle row is the colour channels (blue and red) interposed upon each other. The bottom row is the combined colour and luminosity channels. In 4:1:1, 4:2:0 and 4:2:2, the middle row is subsampled by picking a color and expanding it to a nearby pixel. 4:2:0 is special in the case that all horizontal information is discarded within each sample.

Figure 2.2: Subsampling ratios are illustrated in these images, the leftmost image is unsampled. The upper row of images is the combined channels and the lower row is only the luminosity channels (the image bereft of any colour). The two rows of images illustrate the deterioration of image quality when subsampling colour and luminosity, respectively.
2.2 Lempel-Ziv-Welch

Lempel-Ziv-Welch (LZW) compression is used by the Unix Compress program and in GIF (Graphics Interchange Format)\(^1\) compression. It utilizes the fact that repeating strings of symbols may be coded by look-back pointers to reduce arbitrarily long sequences with fixed length (pointer, quantity) tuples. For instance using sixteen-bit pointers and four-bit quantity indicators enables pointers to sequences of length sixteen up to four kilobytes back in the stream. Images are generally not repetitive in the same manner as running text. In an image consisting of millions of pixels the look-back pointer would have to have a large number of bits. As the length of the look-back buffer increase the compression ratio will decrease since the fixed length of the (pointer, quantity) pairs would increase. Welch also mentioned in his paper that when compressing arrays of floating point decimal numbers, the compression rates were poor. As this is the case of pixel values in images, this method has been considered and given a low priority for implementation and investigation.\(^{[12]}\)

2.3 Huffman Coding

Huffman coding is a common method for compressing data and is a subset of entropy coding. It works by assigning the shortest code words to frequently appearing symbols and longer codewords to the rare ones. This is very useful when many symbols appear frequently, such as in an image. In an image, many of its pixels will have the same or very similar values. With some preparation, many values can be equalized (see section 2.1, section 2.4). The most frequent appearing symbol will be mapped to a short code word, the second most frequent symbol will be mapped to a code word of at least the same length, or one slightly longer. The third most common symbol will be mapped to a code word at least as long as the second was and so on. Note that the length of the code words increase very slowly and are on a magnitude far less than the 32-bit pixels\(^2\). Should the image have very varying pixel values (such as each pixel being unique in the image), the rate of compression would be much worse. In this thesis, each message will be decomposed into a byte array before each byte is Huffman coded.\(^{[6, 7]}\)

2.3.1 Encoding

The simplest encoding algorithm is to construct a Huffman tree and use it to encode the symbols. The tree is then stored alongside the code. The algorithm is as follow:

1. Calculate the probability for each symbol (frequency of appearance divided by length of sequence).
2. Add each to a priority queue (Q). Each element of Q is a leaf of the Huffman Tree. The queue sorts by highest probability first.
3. While \(\text{len}(Q) > 1\):
   
   (a) Poll Q twice.

\(^1\)A type of image format.
\(^2\)ARGB format, 8 bytes each of alpha, red, green and blue information.
2.3. HUFFMAN CODING

(b) Create a new node with these two as children and with a probability equal to the sum of the children.

(c) Add this node to Q (which maintains the priority order).

4. The last remaining node is the root of the Huffman tree.

A priority queue requires $O(\log_2(n))$ per insertion and a binary tree with $n$ leaves has $2n-1$ nodes, this algorithm operates in $O(n\log_2(n))$ time where $n$ is the number of symbols (fig. 2.3). There is a linear-time method using two priority queues, one containing the initial weights and pointers to the associated leaves and the other containing the combined weights and pointers to trees being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the queues and thus minimizing the number of sortings of queues. This is due to that the queue containing trees is likely to be much shorter than the queue containing nodes. [6, 7]

2.3.2 Decoding

The decoding process requires the encoded bit string with its associated Huffman Tree. When compressing images with Huffman encoding in order to send a sequence of images over a network, it might be sufficient to construct the Huffman tree, transmit it once to the client and use the same tree for several successive images. This reduces the transmit overhead significantly, but may result in a loss of compression rate. An example of the decoding process is illustrated in fig. 2.3. [6, 7]

2.3.3 Transmission of Huffman Encoded message

Each message has its own Huffman Tree used to encode and decode it. This tree must be known by both the encoder and the decoder. In the case of encoded text files, the tree is simply stored along with the file and is assumed to be very small compared to the encoded file. In the case of encoding each frame in a video, each frame must be transmitted along with its own Huffman Tree and since sizes of Huffman Trees vary it could consume much bandwidth. There are several ways to code a Huffman tree, described below. Since subsequent frames are similar to each other, as established in section 2.5, the same Huffman Tree is likely to be able to decode several subsequent frames, thus omitting to send a new Huffman Tree for each frame and only send one every few frames. [9]

Frequency transmission

A simple way to transmit the Huffman Tree not requiring much computational power to prepare is to send the symbol frequencies along with their values, then let the receiver reconstruct the Huffman Tree. The transmission is of the following format: $n(f_1, s_1)(f_2, s_2)...(f_n, s_n)$ where $n$ is the number of distinct symbols.

Canonical Huffman Code Book

A coding for the code book that is efficient for transmission over the Internet is the Canonical Huffman Code Book. This coding requires only the transmission of the code
2.4. DISCRETE COSINE TRANSFORM (DCT)  

Chapter 2. Methods

Figure 2.3: There are four distinct symbols, with probabilities denoted in red. To decode a bit string, one pulls a bit from the bit string from left to right and traverses down the tree until a leaf. For instance, the sequence 010111 will result if first a1 (0 leads directly to a1), then to a2 (10) and finally to a4 (111). Note that each code word is of different lengths and there are no length-value sequences.

lengths. Since both transmitter and client knows the inherent properties of the canonical code book, the client require only the code lengths in order to reconstruct each code. The properties of the Canonical Huffman Code Book is the original code book where each code word has the same length as its equivalent in the normal code book, but each code word is sequential. The canonical code book is generated by sorting the code words by first bit length, and then using the symbols natural order. The first code is assigned a new code that consists of as many zeros as the old code was long. Then the code is incremented the code and left-shifted until it is as long as the next code word and assign it to the next symbol. Since both the encoder and the decoder knows the alphabet only the bit lengths needs to be transmitted. This reduces the size of the code book to a fixed-length dependent on the alphabet size. The following pseudo-code will produce a canonical Huffman Code Book [9]:

```python
book = sort(book)
code = 0
while not empty(book):
    print code
    code = (code + 1) << (len(poll(book)) - len(code))
end while
```

2.4 Discrete Cosine Transform (DCT)

Discrete Cosine Transform is a sum of cosine functions, where each cosine function oscillates at different frequencies that represents a finite sequence of data. The DCT is related to the Fourier transforms and is similar to the Discrete Fourier Transform (DFT). The difference between them is that the DCT is only applicable on real numbers, whereas the DFT may handle complex numbers as well. There are eight different DCTs, but only four of those are commonly used and half of those are the primary players in image compression. Those two are the DCT-II and its inverse, the DCT-III. These two are often referred to as simply "The DCT" and "IDCT", respectively.
2.4.1 Usage of DCT in image compression

DCT is used in image compression by dividing the image into eight by eight blocks of pixels. Each block is then divided into the Y’CbCr colour space (just as in section 2.1). Each channel is then transformed by the DCT into cosine coefficients. These coefficients are then quantized by using several techniques including rounding real numbers to integers and dividing by pre-determined weights in order to reduce the quantity of distinct symbols. Human vision is more sensitive to small variances in colour or luminescence over large areas than to variances in the strength of high-frequency brightness. This allows the storing of the high-frequency components in a lower resolution than the lower frequency components. Further compression is usually achieved by entropy coding (such as Huffman code, see section 2.3) the coefficients. [1, 5, 10]

2.5 MPEG-2

A static image is a representation of the distribution of intensities and wavelengths of light in a limited area. A video is a sequence of such images of the same scene in successive time intervals. The human eye can not perceive rapid changes of intensity of light and a range of similar but slightly different images will be perceived as a smooth motion. If the frame rate is less than 15 fps, the sequence appears to stutter. Cinema uses 24 fps, computers 60 and television commonly use 25 or 29.97 fps. [8] The large number of images each second require large amounts of data. Uncompressed CCIR (ITU-r) 601 with resolution of 720 pixels/line and 567 lines (which is a common quality for TV) has a rate of data of close to 300 Mbps (Megabits per second) [4]. Fortunately, there are many ways to reduce this size. MPEG-2 achieves this mainly by not storing each frame separately. Instead, certain frames are chosen at some interval called I-frames and are independent from neighboring frames. They may each be viewed as a full picture. The frames following an I-frame are called P-frames, each describing only the differences between the P-frame and the previous frame (which may be another P-frame). Since the majority of each frame is identical to the one before these static pixels are stored only once in the I-frame and reused in the P-frames. The first frame is always an I-frame, and the following 12-60 frames are P-frames. The longer the segments of P-frames are, the more noticeable any errors become. If an I-frame is broken, this error will affect each following P-frame until a subsequent I-frame is reached. A broken P-frame, on the other hand, will only cause a slight flicker or distortion in the video.

MPEG was originally designed with a few key functionalities: fast forwards search, fast backwards search, limited error propagation and fast image acquisition starting at an arbitrary point. Normal intra-frame coding (the usage of I-frames and P-frames) makes these functionalities difficult to achieve. Therefore the frames are divided into Groups Of Pictures (GOP) and then encode each GOP separately[11].

2.6 ”Collapsed pixels”

This is a technique developed for this thesis. No articles about this technique has been found, nor is anyone claiming credit for it. This may be due to its simplicity. The
2.7. CUDA

CUDA is a parallel computing platform and programming model that utilized the multitude of GPU cores present in NVIDIA graphics cards. A graphics card is tremendously more powerful than a CPU in regard of parallelism while the CPU usually operates at a higher clock frequency. When parallelizing large scale mathematical operations such as image conversions the slower speed of the GPUs over the CPU is alleviated, if not completely circumvented, by having many more truly parallel threads. [2]

2.8 Combinations

In addition to evaluating the methods above, combinations of those compatible with each other was evaluated. For instance, inter-frame differentiation is compatible with the "collapsed pixels" scheme (collapse the differentiation matrix) which is in turn compatible with Huffman Coding (code the \((i, n, v)\) triples). Some of them are on the other hand not compatible with each other; such as first inter-frame differentiation followed by Chroma Subsampling since the differentiation destroys information required by the subsampling algorithm.

2.9 Inherent game properties

The following are in its entirety observations made during the pre-study. The relevant types of games typically have a frame of content that is easily divided into several components:

- **Background.** This is generally a static picture at the lowest level of layers. Sometimes this picture extends beyond the visible borders of the game and the player may pan around in this 'world'.

- **The player avatar.** This is generally a movable object in any shape, most often small compared to background area. It could be as simple as a mouse pointer, but could be as ornate as Super Mario himself.

- **Items.** These may be anything from weapons (if the game is arcade style) or Mahjong tiles (if puzzle style). Generally they can be viewed as a part of the background until interacted with.

- **Enemies.** Non-static avatars, such as Zombies in the popular Plants vs Zombies or the opponents tiles in Backgammon. They either move frequently, or regularly and are mostly similar to the player's avatar.
Most of these components will spend the majority of their time as static objects. Any static pixels will not need to be continuously transmitted to the receiver. Identifying static pixels will reduce bandwidth consumption. A fairly simple scheme to identify the new pixels is to scan through the image and comparing pixel values to the pixel values in the previous frame. Any pixels that do not equal the previous value is new and must be transmitted to the client. This method runs in linear time on the server, but might waste some bandwidth. We have already differentiated the pixels into new pixels and static pixels, and now we must introduce a third category: moved pixels. Moved pixels are a subset of the new pixels, but in the nature of these games lies the fact that pixels tend to move in specific patterns. Clusters of pixels move in the same manner e.g. a chess piece gliding across the board. If one could anchor a cluster to a certain pixel, this cluster may be moved with a simple instruction. Expand this base into as large a rectangle as possible without cutting the edge of the cluster. This entire region could then be transmitted to the client as a simple instruction of four values: \((x_1, y_1), (k_1, k_2)\) where:

- \(x_1\) is the x-value of the pixel to the top left of this bounding box.
- \(y_1\) is the y-value of the pixel to the top left of this bounding box.
- \(k_1\) is the offset in the x-axis to move the top left pixel.
- \(k_2\) is the offset in the y-axis to move the top left pixel.

2.10 Measuring of Streamtainment Solution

Streamtainment is an Estonian company whose main product is a technology to stream high-quality games to low quality clients, more specifically set-top-boxes. Access was given to two machines on the same network, one to act as the server and one to act as the client. Streamtainment installed their software and managed it. Games were given to them that they installed on the machines and started up their solution. The number of concurrent clients was gradually increased while CPU and network traffic was monitored. The test session lasted about two hours. Due to time constraints only two games were tested: One equivalent to the easy game and one equivalent to the hard game. These games are described in more detail in chapter 3.
Chapter 3

Business cases

Three different business cases will be used when measuring the performance of the prototype, each case will be a different type of game, ranging from "easy" to "difficult". "Easy" in this respect does not refer to a game easy to master, but rather a game that requires little bandwidth and processing power to stream to a weak client.

Bejeweled is the "easy" game, where the vast majority of the screen is static (the entirety of the background, borders etc) as well as all tiles not touched with the cursor. A majority of frames will not even change at all from its predecessor since Bejeweled is a game with a low actions per minute (APM) count.

"The world's hardest game" is a moderately difficult game. A large portion of each frame is static but there is a multitude of small moving objects. The game can be described as "the quest to move the Red Square from the first Green Area to the other Green Area while picking up all the Yellow Dots and avoiding the Blue Moving Dots". Similar to the easy game, most of each frame is static but the vast majority of frames will not be identical.

Angry Birds a difficult game since it has several moving objects (the birds, the pigs and falling debris) as well as a moving background (the screen usually pans to the right when firing a bird). The moving background causes the entire image to be completely disjoint from the previous (no pixels remain in the same position).

3.1 Screen Capture

Screen capture is the technique used to extract the video stream frame by frame from the game. This technique is non-trivial, but has already been implemented by Streamtainment and we use their solution. Since this thesis produces only a proof-of-concept no effort will be spent to develop such a technique again.

3.1.1 Assumptions about game environment

The games are all browser-based and customly written for each target device. The games are executed in a browser on the target device. The exact width and height of the target display is known prior to installation. Each game is executed in a browser window where
the game User Interface is the sole component. Any excess screen area is a naked white. When the game is played on a Smart TV, it is known at exactly which pixel coordinate the upper leftmost corner of the user interface is located. This allows for an easy offset calculation to make sure that the Smart TV screen is covered in its entirety by the game. This causes any part of the browser that is "not game" to lie on the outside of the Smart TV screens physical edges, and thus be invisible.

3.1.2 Suggestions for screen capture methods from a headless game

In order to achieve the same behaviour as in section 3.1.1 for this application, one can execute the game in a headless browser and record the graphical user interface. This will allow the game to be executed on another machine than the client and the user interface will be communicated between the host machine and the client device. This recording may be achieved by taking intermittent screenshots. This method may lead to congestion since many screenshot solutions require that the screenshot is stored on the disk, which in this application is unnecessary. The sequence of screenshots may then be quickly trimmed (cutting off any excess browser content) and passed along the actual prototype. The width, height and start-offsets are all parameters determined at startup and will differ for each game and target combination. A problem that may be encountered is if the flash games require some sort of plugin to execute (such as Adobe Flash) which is not necessarily compatible with headless browsers. Such problems are, however, not insurmountable.

A Python Example

The following code snippet starts a headless browser, loads www.google.com and grabs a screenshot before terminating. All credits for the code in listing 3.1 go to Corey Goldberg and his blog\(^1\).

---

Listing 3.1: Original Python Code

```python
#!/usr/bin/env python

from pyvirtualdisplay import Display
from selenium import webdriver

display = Display(visible=0, size=(800, 600))
display.start()

browser = webdriver.Firefox()
browser.get('http://www.google.com')
browser.save_screenshot('screenie.png')
browser.quit()

\(^1\)http://coreygoldberg.blogspot.se/2011/07/python-taking-browser-screenshots-with.html
```
This is simple to extend to grab a screenshot every $\frac{1}{x}$ seconds, where $x$ is the desired frame rate. See listing 3.2 for such an extension.

Listing 3.2: Altered Python Code

```python
#!/usr/bin/env python
from pyvirtualdisplay import Display
from selenium import webdriver

# Added additional imports
from sys import argv
from time import sleep

x = atoi(argv[1])  # Converting the desired frame rate into an integer
display = Display(visible=0, size=(800, 600))
display.start()

browser = webdriver.Firefox()
browser.get('http://www.google.com')

while True:  # Until interrupted
    browser.save_screenshot('screenie.png')  # grab a screenie, disregard the fact that the previous is overwritten
    sleep(1/x)  # wait for 1/x second (may be a floating point number) before grabbing the next screenshot

browser.quit()
display.stop()
```

This implementation is likely far from efficient enough for large scale deployment due to screen shots require some disk activity, but the general principle remains sound.

A JavaScript Example

A more proper suggestion would be to use JavaScript along with a web kit which is specifically designed to handle the execution of headless processes and manage their output. One interesting web kit is Phantom.js\(^2\). The concept is the same as with the python example above. The snippet in listing 3.3 is taken in its entirety from the Phantom.js tutorial page\(^3\).

Listing 3.3: Original JavaScript code

```javascript
var page = require('webpage').create();
```

---

\(^2\) [www.phantomjs.org](http://www.phantomjs.org)

\(^3\) [https://github.com/ariya/phantomjs/wiki/Screen-Capture](https://github.com/ariya/phantomjs/wiki/Screen-Capture)
This may be altered as displayed in listing 3.4 to take a screenshot and pipe it to standard out, to be consumed by a subsequent process.

Listing 3.4: Altered JavaScript Code

```javascript
var page = require('webpage').create();
page.open('http://github.com/', function () {
    page.render('github.png');
    phantom.exit();
});

var fs = require("fs");

function getScreenshot() {
    var base64image = page.renderBase64('PNG');
    fs.write("/dev/stdout", base64image, "w");
    setTimeout(getScreenshot, 1000/x);
}
```
Chapter 4

Solution

As the solution is required to be efficient, it must be able to encode a video stream in near real-time. There are, of course, already methods to accomplish this but all of them require both powerful servers and a fair bit of bandwidth (investigated solution required 100mb/s and 8 CPU cores at 1.2GHz each for twelve concurrent users). Even then, slight buffering is required to provide lag-free display of the content. In this application, buffering is unacceptable due to a significant delay between user input and the effect of that input. The following description is the proposal for a solution prior to implementation. There were some deviations from this, the final architecture is described in section 5.1.

4.1 Proof of concept

4.1.1 Solution Overview

This section explains how the solution is supposed to behave, and the expected input and output formats.

Server

The server application may be viewed as a box into which a headless game may be placed. The box then receives the video stream from the game and performs its transformations on the stream. The output from the game (which is the input to the server application) will be some sort of structure from which a matrix of integers can be extracted. Each element of this matrix is either a 32-bit pixel or a 24-bit pixel on the ARGB or RGB format, respectively. In the latter case, the Alpha value can be assumed to be at its maximum value. Each integer (assumed here to be of a 32-bit length) has the 8 Least Significant Bits (LSBs) to be the value of the blue channel. The following 8 bits is the green channel, the following 8 bits is the red channel. The 8 following the red is the alpha channel. Each pixel looks like this: AAAAAAAA.RRRRRRRR.GG GGGGGG.GBBB BBBBB. The output from the server will be a stream of frames, each encoded and compressed according to the chosen compression scheme.
Client

The host machine of the client software is a very computationally weak client operating at about 300-500MHz but with good capabilities of network I/O (Input/Output). These machines typically have specialized hardware and most have their own C/assembly dialects. This makes the digital boxes difficult to experiment with so to simulate such an environment this project will use an Android device that is underclocked to 400MHz, purely due to the fact that Android is a much more open platform than the Digital Boxes.

The input to the client is the output from the server. The client interprets the message and transforms it back into an image suitable for viewing by a human. The solution is divided into three parts:

1. Recorder. This part captures the visual output of the game. This is a very thin layer that may have to be custom-made for each game.
2. Encoder. This part transforms each frame into a sendable packet. These packets should be as small as possible but still be generated quickly.
3. Client. This part merely receives the packets and updates the Graphical User Interface (GUI).

The recorder has already been created by Streamtainment. They have not released the details of its internal workings, but since it does work no effort was spent on implementing such a mechanism. However, some effort was spent on determining the general principles behind its workings and the difficulty/ease by which to implement it. The Encoder was implemented in Java, since that was the environment used by Accedo (section 4.1.2). The Client was developed and deployed on an Android device as mentioned above (section 4.1.3).

4.1.2 Encoder

The encoder operates as a pipeline in which each frame must traverse before being transmitted to the client. Each step performs one transformation on the frame before forwarding it. For the proof of concept, short videos of three different games were used to benchmark the efficiency of the solution. Audio streaming was ignored since the video information consumed the majority of the bandwidth. The processing and transmitting of user input to control the game was also considered to be insignificant compared to the vast torrent of information that was the video stream. For each connection from a device, a new encoder process was spawned. Each process contained a number of threads and are described in more detail below this short overview:

1. Feeder. This thread merely received frames from the source and fed them to the next.
2. Extractor. This thread consumed the images delivered from the feeder and transformed them into matrices of pixels.
3. Preparer. This thread consumed the matrices produced by the extractor and applied one of two operations: Chroma Subsampling or Discrete Cosine Transformation.
4. Differentiator. Depending on the configuration for the instance it did one of the following: send the matrix directly to the compressor or differentiate the frame first.

5. Compressor. This compressed the image to reduce bandwidth.

6. Transmitter. Transmitted a compressed frame to the client.

Each of these threads worked concurrently, and waited if there is no frame on their step in the pipeline.

**Feeder**

This thread used the Xuggle Java library to read and decode the sample videos. It produced a BufferedImage for each frame which was then injected to the extractor.

**Extractor**

The Extractor consumed the BufferedImages and extracted matrices of integers, where each element was a 32-bit pixel value in the ARGB format.

**Preparer**

This thread consumed the matrices produced by the Extractor and applied one or both of the following techniques: Chroma Subsampling, Discrete Cosine Transformation (DCT). The Chroma Subsampling is much more mundane and is both easier to implement and to understand than DCT. It gives a fair bit of compression with only a slight blurring of delimiting lines within the image. DCT is much more complex and takes far longer time to execute, in fact, it might be too slow for this application. It does, however, result in far better compression rates than Chroma Subsampling with generally less loss to image quality. It is possible to combine these two, but if one does so the Chroma Subsampling must be applied first. This is due to the fact that Chroma Subsampling works with pixel values and "flatten"\(^1\) the DCT will convert the image into cosine functions which obviously are not pixel values. When applying first Chroma Subsampling and then DCT, it is likely that the Subsampling will destroy information that the DCT may have utilized to achieve even greater compression. It is generally considered that using DCT alone when required is the better method.

**Differentiator**

This thread calculated the difference between two frames. It utilized the inherent property of the games that most of the frames have a static content. Only small portions of the visible area were updated in each frame. Any pixels unchanged was not transmitted to the client, thus decreasing the bandwidth. Frames that were completely unchanged were dropped, further decreasing the bandwidth. This thread only had an effect if the Compressor thread was not set to use the UNCOMPRESSED compression mode.

\(^1\)Applies a mean value of Chroma upon a small block of pixels.
Compressor

The Compressor had two modes, and two phases. The first phase was the compression phase, which would be either UNCOMPRESSED, COLLAPSED_PIXELS or HUFFMAN_CODED. The type of compression was dependent on two factors: which of the compression modes that were allowed, and the level of sparseness of the differential matrix. Some sparseness threshold, calculated during runtime determined which of the compression methods will be used. If it was on the sparse side of the threshold, the collapsed pixels technique was used, otherwise the entire matrix was Huffman coded and transmitted in its entirety. The second phase decomposed the message into a byte array suitable for network transportation. Each of these phases may execute somewhat in parallel when required.

Transmitter

This thread consumed the compressed frames and transmitted them to the client over a Java TCP socket.

4.1.3 Client

The client received each frame and displayed it to the user. The client was very weak hardware-wise and was as such divided into only two threads, one to download and one to decode/display the image.
Chapter 5

Results

5.1 Server Hardware

If nothing else is specifically mentioned in conjunction with the presentation of certain results, the results was produced on an Asus UL30vt machine. The Asus UL30vt has an Intel® Celeron® -processor SU2300/743 : 1.2GHz and 1066 DDR RAM.

The architecture described in section 4.1.2 was implemented. However, the architecture was modified and some of the steps were cut during development. The Preparer and the Extractor were removed to reduce synchronization overhead and complexity. The work previously performed by the Preparer was moved to the Compressor and the work of the Extractor was moved to the Differentiator. This modification did not reduce the performance, since these operations are inherently sequential anyway (one can not differentiate un-extracted frames). The final architecture thus became:

1. Server: Differentiator, Compressor, Transmitter.


5.2 Configurations

5.2.1 Discarded Configurations

The following Configurations were tested and discarded as not useful. Each Configuration resulted in a frame rate less than 15 fps, which was the lower threshold of performance.

Plain Huffman Coding

This proved to be too CPU-intense, even though the compression rate was fairly good, the time spent on each frame was way too long which resulted in a low frame-rate (about 4-10 fps depending on the game).

Plain Inter-frame differentiation

This scheme in itself did not reduce the size of a frame at all. What it did do was to quantize the data by creating a matrix consisting of mostly zeroes. The number of distinct symbols in a frame dropped drastically.
Plain Chroma Subsampling

This generally reduced the size of each frame to a third of its original size. Compared to the collapsing of pixels which reduced the size to an average of 10-13% or original size it was found wanting. The conversion from RGB to Y’CbCr colourspace consumed a majority of the time allotted to each frame.

Plain Discrete Cosine Transform (DCT)

This resulted in the best compression rate but was the most time consuming compression schemes considered in this project. Only a partial implementation was completed. Applying the partial solution during test runs turned into poor results, close to one frame every few seconds. Thus the DCT was discarded for being too computationally heavy. This could have been alleviated by using a more powerful host machine but then server costs would rise substantially.

Chroma Subsampling and Huffman Coding

After the pixel matrix was extracted, it was subsampled with a 2:2 ratio. This means that the Cb and Cr channels were divided into blocks of 2 by 2 pixels and each block was replaced with the mean value of the block. This resulted in a row of Y’ values, followed by a row of Cb mean values and a row of Cr mean values. All of these values were then decomposed into byte symbols and Huffman coded. This Configuration gave a fair bit of compression, reducing size to around 20% of original size. Yet again it was noticed that the time spent on each frame was too long. The inter-frame differentiation appeared to be the key to increased compression speed.

Remaining Configurations

The remaining Configurations were combinations of Chroma Subsampling, DCT, and inter-frame differentiation. Neither of these are compatible with each other since each technique destroys information that the others requires to work properly.

5.2.2 Approved Configurations

Config1

This Configuration used inter-frame differentiation and Huffman coding. After the pixel matrix are extracted, the difference between two subsequent frames are calculated before being Huffman Coded.

Config2

This Configuration used inter-frame differentiation, "collapsed pixels" (section 2.6) and Huffman Coding. After the pixel matrix is extracted, the difference between two subsequent frames are calculated. This differential matrix is likely to be sparse\(^1\) which will give

\(^1\)Consisting of mostly zeroes.
the "collapsed pixels" scheme ample room for aggregation. Finally, the aggregated (xcoord, ycoord, value, number) structures were decomposed into byte values and Huffman Coded.

5.3 Easy game

The easy game had a resolution of 1280x720 32-bit pixels. Uncompressed, that was 3.8Mb per frame. At 24 frames per second, that was 88.4Mb per second. As shown in table 5.1 the main difference between the Configurations was the median compression rate where Config2 have twice the compression of config1. Note that the median execution time, 8 ms for Config1 and 12 ms for Config2 was only an increase of a third. Important to note though was that both Configurations waited equally long between frames thus indicating that the compressor was not a bottleneck. Spending that extra time compressing (Config2 over Config1) reduced the median size of transmissions by 36% (table 5.2). Both Configurations deliver similar results from the users perspective (fig. 5.1), though Config2 had a much lower bandwidth consumption but was slightly more CPU intensive.

![Decompressor](image)

**Figure 5.1:** Variations of FPS in the Decompressor for the easy game, this is the framerate perceived by the user. The dip at 10 seconds is due to a rather large event in the game (a large portion of the board was reset). The final dip is because the game was terminating. The rest of the graph indicates an average frame rate of around 17.
Table 5.1: Compression results from the various difficulty/configuration combinations. "Work time" was the time the thread has been working i.e. not idling. Waiting time was the time the thread has been waiting for a blocked resource. These numbers are from running the proof of concept code on a Asus UL30vt (1.2Ghz dual core).

<table>
<thead>
<tr>
<th>Difficulty</th>
<th>Easy</th>
<th>Easy</th>
<th>Medium</th>
<th>Medium</th>
<th>Hard</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>Config1</td>
<td>Config2</td>
<td>Config1</td>
<td>Config2</td>
<td>Config1</td>
<td>Config2</td>
</tr>
<tr>
<td>Mean work time</td>
<td>11ms</td>
<td>20ms</td>
<td>11ms</td>
<td>15ms</td>
<td>18ms</td>
<td>57ms</td>
</tr>
<tr>
<td>Median work time</td>
<td>8ms</td>
<td>12ms</td>
<td>9ms</td>
<td>12ms</td>
<td>13ms</td>
<td>29ms</td>
</tr>
<tr>
<td>Mean waiting time</td>
<td>67ms</td>
<td>67ms</td>
<td>72ms</td>
<td>69ms</td>
<td>66ms</td>
<td>37ms</td>
</tr>
<tr>
<td>Median wait time</td>
<td>53ms</td>
<td>58ms</td>
<td>72ms</td>
<td>58ms</td>
<td>61ms</td>
<td>11ms</td>
</tr>
<tr>
<td>Mean Size (% of original)</td>
<td>1.4%</td>
<td>0.9%</td>
<td>1.1%</td>
<td>0.7%</td>
<td>7.3%</td>
<td>4.5%</td>
</tr>
<tr>
<td>Median Size (% of original)</td>
<td>0.4%</td>
<td>0.2%</td>
<td>1.2%</td>
<td>0.7%</td>
<td>2.7%</td>
<td>1.7%</td>
</tr>
</tbody>
</table>

Table 5.2: Transmission results for the various difficulty/configuration combinations. "Work time" is the time the thread has been working i.e. not idling. Waiting time is the time the thread has been waiting for a blocked resource.

<table>
<thead>
<tr>
<th>Difficulty</th>
<th>Easy</th>
<th>Easy</th>
<th>Medium</th>
<th>Medium</th>
<th>Hard</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>Config1</td>
<td>Config2</td>
<td>Config1</td>
<td>Config2</td>
<td>Config1</td>
<td>Config2</td>
</tr>
<tr>
<td>Mean work time</td>
<td>58ms</td>
<td>62ms</td>
<td>57ms</td>
<td>60ms</td>
<td>59ms</td>
<td>70ms</td>
</tr>
<tr>
<td>Median work time</td>
<td>0ms</td>
<td>0ms</td>
<td>0ms</td>
<td>0ms</td>
<td>47ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Mean waiting time</td>
<td>3ms</td>
<td>2ms</td>
<td>5ms</td>
<td>6ms</td>
<td>1ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Median wait time</td>
<td>0ms</td>
<td>0ms</td>
<td>0ms</td>
<td>0ms</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>Mean size</td>
<td>34kb</td>
<td>22kb</td>
<td>26kb</td>
<td>17kb</td>
<td>173kb</td>
<td>107kb</td>
</tr>
<tr>
<td>Median size</td>
<td>11kb</td>
<td>7kb</td>
<td>29kb</td>
<td>18kb</td>
<td>65kb</td>
<td>40kb</td>
</tr>
<tr>
<td>Bandwidth requirement</td>
<td>564kb/s</td>
<td>348kb/s</td>
<td>423kb/s</td>
<td>258kb/s</td>
<td>2716kb/s</td>
<td>1481kb/s</td>
</tr>
</tbody>
</table>

5.4 Medium game

The medium game had a resolution of 1280x720 pixels. Uncompressed, that was 3.8Mb per frame. At 24 frames per second, that was 88.4Mb per second. For the medium game the difference in compression time was even less than for the easy game, 9 ms (an increase by 1 ms from the easy game) for Config1, and 12 ms (no increase at all) for Config2 (table 5.1). Furthermore, the difference in bandwidth consumption increased, being 423 kb/s for Config1 and 258 kb/s for Config2. Config2’s bandwidth consumption is 40% of Config1’s, just as it was for the easy game. Note that the medium game appeared to yield better results than the easy game, this is discussed in section 6.3.
Figure 5.2: Variations of FPS in the Decompressor for the medium game. The final dip is then the game terminates. The rest of the graph indicates a frame rate of around 17.
5.5 Hard game

The hard game had a slightly higher resolution than the easy and the medium game at 1280x766 pixels (due to its wide screen nature). That makes each frame 3.9Mb (as opposed to 3.8Mb for the other two games) uncompressed and at 24 frames per second requires a download rate of 94Mb/s. The compression time for Config1 did not increase significantly from the medium game as the "collapsed pixels" scheme has a linear complexity to the number of pixels (table 5.1). However, the compression time for Config2 increased dramatically, almost by 380% when compared to the medium game. Furthermore, the bandwidth consumption increased by 642% for Config1 and 570% for Config2 when compared to the medium game (table 5.2). The relevant ratio between the two configs lie at 54%, which was worse than the ratio for both the easy and medium games. The perceived performance for the user is illustrated in fig. 5.3, where Config1 display an alright frame rate, with some disrupting lag spikes, while Config2 display a frame rate that is too low.

![Decompressor](image)

Figure 5.3: Variations of FPS in the Decompressor for the hard game. The dips in the graphs are when a pan happens, which gives a dense differential matrix. The large drop in both graphs both happen when the main event in the game is triggered - the shooting of a bird. This gives a completely saturated differential matrix which takes time to both compress and transmit.
Table 5.3: The number of clients and bandwidth consumption per client when deploying the solutions on a machine equivalent to Amazon’s High-CPU servers.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Proposed</th>
<th>Streamtainment</th>
<th>Proposed</th>
<th>Streamtainment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max #Clients</td>
<td>23</td>
<td>57</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>Bandwidth (per client)</td>
<td>260 kb/s</td>
<td>887 kb/s</td>
<td>13 Mb/s</td>
<td>10.8 Mb/s</td>
</tr>
</tbody>
</table>

5.6 Large scale testing

Testing was performed on a machine with 8 CPU cores of 1.2 GHz each and a 10 GB/s Ethernet connection. This was a server that the solution could be deployed on should it be put into production. Note that this machine is not the same as in table 5.1 and table 5.2. Note also that the games used here are not the same as those used to produce table 5.1 and table 5.2. This is due to some technical constrains on Streamtainment’s solution and the games used are roughly equivalent if not slightly easier.

5.6.1 Proposed Solution

Roughly twenty clients of the easy game consistently consumed about 6 mb/s (260 kb/s each) of bandwidth (table 5.3). At 24 clients the frame rates on all clients suddenly dropped from between 17-26 to 2-4 FPS. The CPU also started to consistently spend about 80% of its time in kernel mode while having spent only a few percent of its time in kernel mode prior to the 24th client. The behaviour of the hard game clients were very similar, the same number of clients with substantially higher bandwidth consumption. At the 24th client the fps dropped from 14-17 fps per client to 0-2 fps per client and the CPU spent roughly 80% of its time in kernel mode.

5.6.2 Streamtainment solution

For the easy game equivalent, the solution ran 57 clients simultaneously, consuming all of the CPU power and about 880b/s per client (table 5.3).

For the hard game equivalent, the solution ran 12 clients simultaneously. When those twelve solutions was running, they consumed 130mb/s (or 10833kb/s per client) and all of the CPU power.

5.6.3 Economical Calculation

When taking the results from section 5.6.1 and section 5.6.2 and applying them to the calculations in table 5.4 and table 5.5 we see that for the easy game, the proposed solution is 1.28 times more expensive than Streamtainment’s solution. This is mainly due to the fact that they could run more than twice the number of concurrent users per server. Furthermore, the total cost per month for Streamtainment’s solution is not wholly accurate. Amazon only have prices for the bandwidth until it exceeds 500TB/month. When this happens, one must contact them and bargain for the price per GB for the excess data traffic. For the easy game, Streamtainment’s solution has 258TB excess data traffic per
month.

For the hard game Streamtainment’s solution is roughly three times as expensive, even without the additional 6.7 PB\textsuperscript{2} of extra data traffic whereas the proposed solution only has an excess of 623 TB. These calculations are based on the number of users during the prime-time (assumed by supervisor to be 50,000 users between 18:00-00:00) and the number of users during the off-time (10,000 users during the remainder of the day). The number of off-time users is likely to be far lower than a constant 10,000 and with Amazon’s cloud computing services it would be possible to adjust the number of servers dynamically as the number of users increase or decrease.

\textsuperscript{2}Petabyte, 1,000 terabytes.
Table 5.4: Economic calculation for deploying the proposed solution versus Streamtainment’s solution with the easy game on servers on Amazon’s Cloud Based Computing Services. OT stands for off-time, PT for prime-time.

<table>
<thead>
<tr>
<th>Easy game</th>
<th>Proposed</th>
<th>Streamtainment</th>
<th>Proposed</th>
<th>Streamtainment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Server costs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-CPU extra large ($/h)</td>
<td>0.66</td>
<td>0.66</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>data in ($/GB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Data out</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>first GB/month ($/Gb)</td>
<td>0</td>
<td>0</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>up to 10TB/month ($/Gb)</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>next 40TB/month ($/Gb)</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>next 100TB/month ($/Gb)</td>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>next 350TB/month ($/Gb)</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td><strong>Data traffic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free GB/month</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>&quot;10TB&quot;/month</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>&quot;40TB&quot;/month</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>&quot;100TB&quot;/month</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>&quot;350TB&quot;/month</td>
<td>98.83</td>
<td>350</td>
<td>350</td>
<td>350</td>
</tr>
<tr>
<td>&quot;RemainingTB&quot;/month</td>
<td>0</td>
<td>257.87</td>
<td>0</td>
<td>257.87</td>
</tr>
<tr>
<td><strong>Data out cost $/month</strong></td>
<td>16741.5</td>
<td>29312,8935</td>
<td>29312.8935</td>
<td>29312,8935</td>
</tr>
<tr>
<td><strong>Measured Performance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>requirement KB/s</td>
<td>288</td>
<td>877.19</td>
<td>288</td>
<td>877.19</td>
</tr>
<tr>
<td>#users/prime-time server</td>
<td>23</td>
<td>60</td>
<td>23</td>
<td>60</td>
</tr>
<tr>
<td>#users/off-time server</td>
<td>23</td>
<td>60</td>
<td>23</td>
<td>60</td>
</tr>
<tr>
<td><strong>Rates of data traffic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OT Data out MB/hour</td>
<td>172800</td>
<td>526300</td>
<td>172800</td>
<td>526300</td>
</tr>
<tr>
<td>PT Data out MB/hour</td>
<td>864000</td>
<td>2631500</td>
<td>864000</td>
<td>2631500</td>
</tr>
<tr>
<td>PT Data out GB/month</td>
<td>155520</td>
<td>473670</td>
<td>155520</td>
<td>473670</td>
</tr>
<tr>
<td>OT Data out GB/month</td>
<td>93312</td>
<td>284292</td>
<td>93312</td>
<td>284292</td>
</tr>
<tr>
<td>Data out MB/hour</td>
<td>17.28</td>
<td>52.63</td>
<td>17.28</td>
<td>52.63</td>
</tr>
<tr>
<td>Data out (GB/month)</td>
<td>248832</td>
<td>757872</td>
<td>248832</td>
<td>757872</td>
</tr>
<tr>
<td>Data out (TB/month)</td>
<td>248.83</td>
<td>757.87</td>
<td>248.83</td>
<td>757.87</td>
</tr>
<tr>
<td><strong>Number of servers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#prime-time servers</td>
<td>2173.91</td>
<td>833.33</td>
<td>2173.91</td>
<td>833.33</td>
</tr>
<tr>
<td>#off-time servers</td>
<td>434.78</td>
<td>166.67</td>
<td>434.78</td>
<td>166.67</td>
</tr>
<tr>
<td><strong>Final Costs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-time cost $/month</td>
<td>154955,59</td>
<td>108002,16</td>
<td>154955,59</td>
<td>108002,16</td>
</tr>
<tr>
<td>Prime-time cost $/month</td>
<td>258260,51</td>
<td>179999,28</td>
<td>258260,51</td>
<td>179999,28</td>
</tr>
<tr>
<td>Data out cost $/month</td>
<td>16741,5</td>
<td>29312,8935</td>
<td>16741.5</td>
<td>29312,8935</td>
</tr>
<tr>
<td>Data in cost $/month</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Environmental Assumptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Days/month</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>#Off-time hours</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>#Prime-time hours</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>#Off-time users</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>#Prime-time users</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
</tbody>
</table>
Table 5.5: Economic calculation for deploying the proposed solution versus Streamtainment’s solution with the hard game on servers on Amazon Cloud Based Computing Services. OT stands for off-time, PT for prime-time.

<table>
<thead>
<tr>
<th>Hard game</th>
<th>Proposed</th>
<th>Streamtainment</th>
<th>Data traffic</th>
<th>Proposed</th>
<th>Streamtainment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Server costs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-CPU extra large ($/h)</td>
<td>0.66</td>
<td>1.2</td>
<td>Free GB/month</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>data in ($/GB)</td>
<td>0</td>
<td>0</td>
<td>&quot;10TB&quot;/month</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>data out</td>
<td>0</td>
<td>0</td>
<td>&quot;40TB&quot;/month</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>first GB/month ($/Gb)</td>
<td>0.12</td>
<td>0.12</td>
<td>&quot;100TB&quot;/month</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>up to 10TB/month ($/Gb)</td>
<td>0.09</td>
<td>0.09</td>
<td>&quot;350TB&quot;/month</td>
<td>350</td>
<td>350</td>
</tr>
<tr>
<td>next 40TB/month ($/Gb)</td>
<td>0.07</td>
<td>0.07</td>
<td>&quot;RemainingTB&quot;/month</td>
<td>623.2</td>
<td>8860</td>
</tr>
<tr>
<td>next 100TB/month ($/Gb)</td>
<td>0.05</td>
<td>0.05</td>
<td>Data out cost $/month</td>
<td>29331.16</td>
<td>29743</td>
</tr>
<tr>
<td>next 350TB/month ($/Gb)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prime-time server cost ($/h)</td>
<td>0.66</td>
<td>1.2</td>
<td>requirement KB/s</td>
<td>1300</td>
<td>10833.33</td>
</tr>
<tr>
<td>Off-time server cost ($/h)</td>
<td>0.66</td>
<td>1.2</td>
<td>#users/prime-time server</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#users/off-time server</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td><strong>Rates of data traffic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OT Data out MB/hour</td>
<td>780000</td>
<td>6500000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT Data out MB/hour</td>
<td>3900000</td>
<td>32500000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT Data out GB/month</td>
<td>702000</td>
<td>5850000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OT Data out GB/month</td>
<td>421200</td>
<td>3510000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data out MB/hour</td>
<td>78</td>
<td>650</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data out (GB/month)</td>
<td>1123200</td>
<td>93600000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data out (TB/month)</td>
<td>1123.2</td>
<td>9360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Final Costs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-time cost $/month</td>
<td>154955.59</td>
<td>539997.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prime-time cost $/month</td>
<td>258260.51</td>
<td>900000.72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data out cost $/month</td>
<td>29331.16</td>
<td>29743</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in cost $/month</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total cost $/month</td>
<td>471878.42</td>
<td>1499484.56</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Measured Performance**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#Days/month</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>#Off-time hours</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>#Prime-time hours</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>#Off-time users</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>#Prime-time users</td>
<td>50000</td>
<td>50000</td>
</tr>
</tbody>
</table>
Chapter 6

Discussion

6.1 The key to the solution

They key technique is the inter-frame differentiation. This quantization technique reduces the range of symbols substantially which in turn allows further compression schemes to achieve much boosted compression rates. This quantization is what enables the use of Huffman Coding even though Huffman on its own was too inefficient to reliably deliver an acceptable frame rate.

6.2 ”Best” configuration

The configurations that utilize the inter-frame differentiation appeared to produce the best frame rates overall, and also to consume the least bandwidth. This effectively excluded all configurations that was incompatible with inter-frame differentiation; namely Discrete Cosine Transform and Chroma Subsampling. This was indicated early on in the development process and as such only a perfunctory implementation of the DCT was attempted. Presumptive measurements of the transform itself (RGB $\rightarrow$ cosine coefficients) indicated that this process would be too slow. After that transformation even more time would be spent on quantization, Huffman coding and transmission.

Remaining configurations was: Config1 and Config2 (described 5.2.2). Of these two, the former gave the best frame rate for each game. However, the latter produced only slightly lower frame rates but consumed much less bandwidth. On the other hand it consumed slightly more processing power. Since it appears as if the bandwidth is the bottleneck for individual sessions, it must be concluded that Config2 holds the most merit. The gap in bandwidth may be increased further by applying the optimizations mentioned in section 2.3.3. When observing the different tables it became apparent that Config2 scaled much better and consistently required less bandwidth at the cost of a slight drop in frame rates.

When observing the frame rate chart Figure 5.3, and using knowledge of how the game behaved during measurements, it was noted that the peaks in the frame rates coincided with when the image was static. As soon as the backgrounds of the games started to pan around the frame rates dropped sharply. This validated the assumption made at the start
6.3 THE EASY & MEDIUM GAME

One may think it is peculiar that the bandwidth consumption presented for the medium game in table 5.2 is lower than the easy game. This is due to how the consumption is calculated, the bandwidth presented is the average calculated by aggregating the data sent during a ten second period and divide it with the time that has passed. The median size of an easy frame is very small, close to a third of the median size of a medium frame. (table 5.1) Most in-game events in the easy game results in a reset of a large portion of the game board (though these events are often separated by at least a second) and as this reset is animated there will be several subsequent frames with a large difference to its predecessor which in turn will hamper the effect of the inter-frame differentiation. These events will make the average bandwidth consumption larger and temporarily bring the "difficulty" of the game to almost the same levels as the hard game. Only few of these events occur during the course of a game. Perhaps Bejeweled does not belong in the easy category and it would have been more appropriate to use a truly static game such as sudoku. The medium game on the other hand does not have such level resets. Whereas the easy game has periods of inactivity followed by a burst of updates, the medium game has a continuous activity of smaller movements in the game environment. It would appear as if the initial assumptions about the difficulties in the games were slightly off track. The initial assumption was that games that spent most of its time in a static state was easier than one in perpetual motion. This observation indicates that the proposed solution works better for continuous small movements than for infrequent large ones.

6.4 THE HARD GAME

This game was a great leap from the medium game, both compression time and bandwidth consumption increased dramatically as expected. Due to the nature of the game (scrolling environment) the differentiation matrices were dense which makes the "collapsed pixels" scheme to produce worse compression ratios. It also increase the size of the Huffman tree (the size of the tree depends on the number of distinct symbols) which will increase encoding and decoding times. The solution was not designed with such a game in mind\footnote{Since such games are not common on Smart TV platforms.} hence the poor results but the game is still a fine illustration of the limit of this solution’s capabilities.

6.5 STREAMTAINMENT

Streamtainment claim to have a theoretical max cap of 297 concurrent sessions with their current implementation, but in actuality they admit it to be 60. This limitation is due to their utilization of CUDA technology. Streamtainment agreed that utilizing CUDA would greatly increase the performance of the solution proposed in this thesis. For this particular
set of games, the solution in this thesis is more efficient even without the use of CUDA or other mass-core parallelism when considering both bandwidth and CPU consumption to fps ratio.

6.6 Parallelism

By using a multitude of processor cores the efficiency of this solution would increase a lot more than just using more powerful CPUs. With the current 1.3GHz dual-core processor the games were playable albeit on a low frame rate. By using a hundred or so cores (achievable by CUDA, for instance, using the GPU clusters for extreme parallelism) one could theoretically prepare and transmit a game at 24 FPS real-time, assuming of course that the bandwidth is not bottle-necking the process.

6.7 Bottlenecks

It was initially hypothesized that it was the bandwidth that would be the bottleneck in this application. However, this was only true in a different aspect. The bandwidth was the aspect of the solution that had to be solved in order to make the solution at all possible. The bandwidth was ultimately the bottleneck for each session individually. Though what turned out to be the bottleneck in large scale deployment was, quite unexpectedly, not the bandwidth nor even the number of available CPU cycles but the number of hyperthreads. Java, which the proof of concept was written in, does not guarantee that the threads created within the virtual machine will each run on an independent hyperthread. Therefore much parallelism that could be utilized (and actually was implemented into the proof of concept code) is not used to its fullest potential. Instead of running each step of the pipeline on a separate hyperthread (and thus running them in *true* parallel). They may actually run on only one hyperthread and the implemented parallelism was only used to overlap internal I/O and message passing thus severely limiting the benefit of such an architecture. This could easily be countered (and should be) by using a more lower level language with more control of the hyperthreads and even using CUDA or other GPU based computational frameworks.

6.8 Economical cost

The calculation in table 5.4 shows that using the proof-of-concept implementation is theoretically economically competitive with Streamtainment’s solution. Consider the Proof-of-concept code is just that, while Streamtainment’s solution is close to ready to be deployed. By refining the solution with the proposals in section 6.11 and section 6.6, especially by the use of CUDA or equivalent technologies the performance of the proof-of-concept implementation should be surpassed by far, emphasizing large scale parallelism over fast CPUs.

It would be cheaper to deploy Streamtainment’s solution for the easy games than the proposed solutions, but not by a long shot. (5.4) The proof-of-concept implementation was a program implemented during a short period of time by someone with little to no
prior experience of such projects and it is holding up fairly well against a product developed by a team of experienced engineers. The proposed solution is only 1.28 times more expensive which indicates that with some further improvements (discussed in section 6.11) it would actually be cheaper than Streamtainment’s solution. For the hard game the proposed solution is actually cheaper than Streamtainment’s solution. \((5.5)\) and with the improvements mentioned above, the gap in pricing will only increase.

The factor that appeared to affect the price the most was the number of concurrent sessions per server (in actuality, the number of servers). Streamtainment claims that their current solution will handle twenty to sixty concurrent sessions for each server depending on which game, while the proof-of-concept handles about twenty regardless of game. Furthermore, the proof-of-concept consumes far less bandwidth. Streamtainment did not wish to divulge too much of their implementation techniques, but this was likely due to the fact that their solution caters to a much wider variety of games than this thesis focuses on. There are no server on Amazon that is optimal to deploy Streamtainment’s solution on. Their solution is utilizing CUDA and there is but one machine on Amazon which support CUDA. That machine has two Tesla NVIDIA cards, each of which has about 2600 CUDA cores, and according to Streamtainment, any cores over 500 will not be used optimally. The server prices for the Streamtainment solutions in table 5.4 and table 5.5 was extrapolated by assuming a somewhat even distribution of pricing between servers. The proof-of-concept implementation, if implemented with CUDA, would likely not be hampered by some 500 core limitation. It should be able to be deployed on an Amazon High-CPU Cluster Server machine, with the number of concurrent users running in at least the high hundreds. This should bring the price down significantly from the prices in table 5.4 and 5.5.

6.9 Identified time-sinks

Profiling the execution of the most promising Configurations reveals that the implementation of the Huffman Coding algorithm was far from optimal. However, most of the time "wasted" was spent by Java’s internal buffering and copying of objects. Furthermore, the upload rate of the test machine was limited, a dedicated server would have a much higher upload rate. It was also observed that the CPU-consumption was much lower in the "hard" difficulty than in the lower difficulties due to the compression taking roughly the same amount of effort, but the resulting compression was worse which further compounded the transmission bottleneck.

6.10 Skewed Results

The method used to simulate the game execution and sampling consumes much more computational power than the real-life implementation would have, and as the measurements has been performed on a computer with much lower performance than a dedicated server would be (dual-core Intel U7300 1.3 GHz and a generic on-board-MoBo network card). Even so, the games are fully playable with such a frame rate, even under these conditions.
6.11 Further improvements

By writing the client side in hardware-specific C or assembly (as is common with digital box firmware) the client will work much more efficiently. By implementing the server side in C or C++ will reduce much of the overhead in Java, mainly type-casts and array buffering. Java impose (or may impose, depending on the situations) various dynamic optimizations of method calls and loops. Sometimes this may not be optimization but instead only a waste of time due to untimely decisions to optimize code that is soon abandoned. Java might accidentally erroneously optimize something, for instance a set of recursive calls just prior to the termination of the call sequence. This will waste time optimizing. The same applies for loops and in-lining of methods. By substituting the language to C or C++ you gain more control of execution, and in such a delicate application as this (were optimization is more important than ease of development), such languages are suitable. It is also known that Java Socket I/O imposes several buffer copies prior to transmission, which increases the transmission overhead significantly.

In the proposed solution the game is sampled at 24 fps and each frame is then compressed and transmitted to the client. The compression and transmission takes some time which cause a small delay each frame. This delay is accumulative and will eventually result in a state where the player reacts to a state that has long since passed which renders the game unplayable. What is needed is therefore some sort of protocol that adjusts the sampling rate of the games on the server side, depending on the frame rate the client receives.
Chapter 7

Conclusions

This proof-of-concept implementation is at a playable performance fps wise. Applying the optimizations suggested earlier especially the CUDA or equivalent technology and the solution proposed (Config2) should perform admirably both performance wise and economically. The calculation in 5.4 and table 5.5 backs this claim. Streamtainment shows that using CUDA for mass-parallelization of operations on images is very effective. The use of CUDA is what enables them to have three times as many concurrent users on the same hardware as the proposed solution. The next phase of development for the proposed solution would be to refine the implementation and architecture with the use of GPU computing of matrix operations in order to increase the number of concurrent sessions per server. For this range of games Streamtainment’s solution is not preferable to the proposed solution. The proposed solution as a concept is superior to Streamtainment’s solutions concept for this range of games but currently the implementation of Streamtainment is unsurprisingly superior to the proof-of-concept code.
Chapter 8

Bibliography


