Image encoding evaluation in remote desktop systems
- A framework for measuring the encoding performance in TigerVNC

Bildkodningsevaluering i fjärrskrivbordssystem

Adam Halim

Supervisor : Martin Funkquist
Examiner : Peter Jonsson

External supervisor : Frida Flodin
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtidiga ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Adam Halim
Abstract

Remote desktop solutions have widespread adoption across the world, allowing people to connect to a computer remotely from anywhere in the world. One widely used solution is TigerVNC which uses the RFB protocol for communication between a client and server. TigerVNC supports several encoding types, which use different techniques to compress image data. Currently, there is a lack of a performance evaluation frameworks for VNC software that makes it possible to measure the performance of not only different encoders, but also the performance of the system that chooses which encoding to use for different parts of the image. This thesis presents a framework that was developed to evaluate the performance of TigerVNC server in real-world scenarios. The framework includes a tool that records an X session losslessly and a benchmarking suite that processes a recorded session, providing data regarding execution time and compression ratio. Benchmarks were run using several encoding settings with different recorded sessions representing real-world scenarios. Results show that TigerVNC server has a good tradeoff between compression ratio and execution time. The work done in this thesis lays a foundation on which future research can be done, leading to improvements in the TigerVNC project.
Acknowledgments

I want to thank the entire team at Cendio AB for a great time and a warm welcome to the office, with special thanks to my supervisor Frida Flodin for helping me and discussing ideas crucial to this study. I also want to thank my examiner Peter Jonsson and my supervisor Martin Funkquist for their valuable feedback during this project. Finally, I want to express my gratitude towards my opponent Anna Larsson for her great insights which notably improved the quality of this thesis.
# Contents

**Abstract** iii

**Acknowledgments** iv

**Contents** v

**List of Figures** vii

**List of Tables** viii

1 **Introduction** 1
   1.1 Motivation 1
   1.2 Aim 2
   1.3 Research questions 2
   1.4 Approach 2
   1.5 Delimitations 2
   1.6 Contributions 2

2 **Background** 3
   2.1 Data compression 3
   2.2 X Window System 6
   2.3 Remote desktop software 7
   2.4 Remote framebuffer protocol 7
   2.5 VNC 9
   2.6 TigerVNC 10
   2.7 Performance metrics 11

3 **Related work** 13

4 **Framework design** 16
   4.1 Recording an X session 16
   4.2 X session benchmark 18

5 **Testing setup** 20
   5.1 Test scenarios 20
   5.2 Benchmarking 24
   5.3 Test environment 24

6 **Results** 25
   6.1 Default settings 25
   6.2 Varying compression 28
   6.3 Test case comparison - MPx/s 32
   6.4 Upper bound compression 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Discussion</td>
<td>37</td>
</tr>
<tr>
<td>7.1 Results</td>
<td>37</td>
</tr>
<tr>
<td>7.2 Method</td>
<td>38</td>
</tr>
<tr>
<td>7.3 The work in a wider context</td>
<td>41</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>8.1 Research questions</td>
<td>42</td>
</tr>
<tr>
<td>8.2 Future work</td>
<td>43</td>
</tr>
<tr>
<td>Bibliography</td>
<td>44</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Huffman tree for the set of symbols $S = \{a, b, c, d, e, f, g, h\}$ ........................................ 5
2.2 Zig-zag pattern used in JPEG compression ................................................................. 7
2.3 High-level overview of TigerVNC architecture ......................................................... 10

4.1 DAMAGE events containing overlapping rectangles .................................................. 17
4.2 DAMAGE events coalesced into a single region. The cross hatch pattern represents regions that have not changed ......................................................... 17

5.1 Screenshot from the word processing case ............................................................... 21
5.2 Screenshot from the 3D CAD case ............................................................................ 21
5.3 Screenshot from the photo editing case ................................................................. 22
5.4 Screenshot from the video editing case ................................................................. 22
5.5 Screenshot from the video streaming case ............................................................ 23
5.6 Screenshot from the web browsing case ............................................................... 23

6.1 Execution times and compression ratios in the word processing test case for all encoders, using compression levels 0-9 ................................................................. 29
6.2 Execution times and compression ratios in the 3D CAD test case for all encoders, using compression levels 0-9 ................................................................. 29
6.3 Execution times and compression ratios in the photo editing test case for all encoders, using compression levels 0-9 ................................................................. 30
6.4 Execution times and compression ratios in the video editing test case for all encoders, using compression levels 0-9 ................................................................. 30
6.5 Execution times and compression ratios in the video streaming test case for all encoders, using compression levels 0-9 ................................................................. 31
6.6 Execution times and compression ratios in the web browsing test case for all encoders, using compression levels 0-9 ................................................................. 31
6.7 Encoded MPx/s for the Tight encoder ................................................................. 32
6.8 Encoded MPx/s for the ZRLE encoder ................................................................. 33
6.9 Encoded MPx/s for the default encoder ................................................................. 33
6.10 Encoded MPx/s for the RRE, Hextile and JPEG encoder ........................................ 34
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Huffman code table</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Encoding types defined in The RFB Protocol standard</td>
<td>8</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of the test scenarios benchmarked in this thesis</td>
<td>20</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of benchmark for the word processing test case using the default settings</td>
<td>26</td>
</tr>
<tr>
<td>6.2</td>
<td>Summary of benchmark for the 3D CAD test case using the default settings</td>
<td>26</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary of benchmark for the photo editing test case using the default settings</td>
<td>27</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary of benchmark for the video editing test case using the default settings</td>
<td>27</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary of benchmark for the video streaming test case using the default settings</td>
<td>28</td>
</tr>
<tr>
<td>6.6</td>
<td>Summary of benchmark for the web browsing test case using the default settings</td>
<td>28</td>
</tr>
<tr>
<td>6.7</td>
<td>Brute force results using the default settings, showing execution time, compression ratio, and how many of the rectangles were encoded by each encoder</td>
<td>35</td>
</tr>
<tr>
<td>6.8</td>
<td>Brute force results using default settings compared against the default encoder</td>
<td>35</td>
</tr>
<tr>
<td>6.9</td>
<td>Brute force results using compression level two for Tight and ZRLE, showing execution time, compression ratio, and how many of the rectangles were encoded by each encoder</td>
<td>35</td>
</tr>
<tr>
<td>6.10</td>
<td>Brute force results using compression level two for Tight and ZRLE compared against the default encoder</td>
<td>36</td>
</tr>
</tbody>
</table>
This thesis was conducted in collaboration with Cendio AB at their offices in Linköping, which currently maintain the TigerVNC project. The introduction chapter of this thesis serves as an overview of the research presented in the following chapters. It includes motivation and context for the study, as well as the research question and objectives.

1.1 Motivation

The use of remote desktop applications has had widespread adoption in recent years. This allows users to access and work on a computer remotely from anywhere in the world. With this kind of technology, traditional work practices have changed, allowing people to work from home or when traveling. The remote machine does not even have to be a physical one, but can instead be a virtual desktop or application. This has great benefits, allowing companies great flexibility with their employees by enabling them to work from home and hire anyone from anywhere in the world. With more and more users working remotely, performance can take a hit, which decreases the users’ experience. To reduce the load on a network, images are typically compressed before they are sent. This reduces the file size of each image and bandwidth used by a system, but as compression is computationally complex, it comes at the cost of increased CPU usage.

There is a plethora of remote desktop services and protocols available to choose from, each using different techniques to deliver a good user experience. Some examples of these services and protocols are Simple Protocol For Independent Computing Environments (SPICE), Remote Desktop Protocol (RDP), and the Remote Framebuffer Protocol (RFB). The RFB protocol is used by the VNC family of software to send and receive image data and is responsible for, among other things, negotiating which image encoding should be used by the server and client. The protocol is standardized and requires the support of several encoding types, but is also extensible so that new encodings can be added with relative ease. While multiple encoding types are available, RFB does not specify that each image must be sent using the same encoding. This means that a single image can be divided into multiple smaller images, where each sub-image is encoded independently.

1. https://www.spice-space.org/
1.2 Aim

This thesis aims to analyze the performance of TigerVNC [37], an open-source VNC client and server application. In particular, it is the server-side system that chooses which encoding algorithm should be used for each image. A testing framework is developed which looks at key performance metrics. Data sets are constructed that represent real-world remote desktop usage scenarios.

1.3 Research questions

1. How can a testing framework be developed for TigerVNC that measures the performance in real-world scenarios?
2. How effective is TigerVNC at choosing the appropriate compression algorithm?
3. What is the effect on performance, measured in execution time and compression ratio, when varied input data is used?

1.4 Approach

In this thesis, a literature study is performed to investigate current methods of evaluating the performance of remote desktop systems. Furthermore, a testing framework is being developed for TigerVNC server. The framework will be used to test various configurations of the encoders in TigerVNC, which lays a foundation on which analyses can be made. With this foundation, it is easier to test how TigerVNC performance reacts to changes made in different parts of the system. By analyzing the performance of different implementations, smarter choices can be made in future versions of TigerVNC, which improves performance for users.

1.5 Delimitations

Image quality can be measured in objective metrics, such as peak signal-to-noise ratio (PSNR) and structural similarity index measure (SSIM). While these metrics are useful when comparing different image compression techniques, they are of lesser importance in this thesis. Most image compression algorithms, such as JPEG, are designed with natural photographs in mind and are highly effective in preserving image quality as perceived by humans. However, computer-generated imagery differs greatly from real photographs and may exhibit noticeable artifacts as a result of lossy compression algorithms. While a compressed image might look good using objective metrics, an artifact can disproportionately negatively impact the image quality perceived by a human. Another aspect that is omitted in this thesis is client interactability. Due to time constraints, metrics regarding input and display update latencies are not being looked at, and only the server implementation will be tested.

1.6 Contributions

This thesis presents a testing framework for evaluating the performance of TigerVNC server. Included are also data sets that represent real-world remote desktop usage scenarios. The contributions include:

- A testing suite for evaluating CPU performance and compression ratios for TigerVNC server.
- A foundation for future improvements.
- New data sets that represent real-world remote desktop usage scenarios.
2 Background

This chapter provides relevant background information and theory needed to understand the research presented in this thesis.

2.1 Data compression

Data compression is used to reduce the size of computer files by reducing the amount of data needed to represent a file. By encoding a file in a way that gets rid of redundant information, the same file can be reconstructed using less data[15]. This can be done without losing any information in the encoding process and is then called lossless compression. If information is lost in the encoding process leading to a degradation of quality, the compression is said to be lossy.

2.1.1 Shannon’s source coding theorem

Lossless data compression is a well-studied area of research spanning over seven decades, back to Shannon’s paper from 1948[36]. The notion of entropy is introduced in his paper, which measures the average of how much data is contained in a random variable. Shannon invented a formula to quantify the entropy of a signal of discrete, independent random variables:

\[ H(X) = -K \sum_{i=1}^{n} p_i \log_2 p_i, \]  

(2.1)

where \( H(X) \) is the entropy for a random variable \( X \), \( p_1, ..., p_n \) denote the set of probabilities for \( x \), and \( K \) is a positive constant. The value \( H \) is the minimum amount of bits needed to code a symbol and can be used to determine the limit for data compression without any information loss. Concerning image data, entropy is regarded as the amount of redundancy present in the image. It is not uncommon for images to have both statistical and spatial redundancies, meaning certain pixel values and patterns are likely to occur multiple times in the same image. As Shannon’s formula is only based on statistical and not spatial information, calculating the Shannon entropy for an image is not a trivial task.
2.1. Data compression

2.1.2 Image compression

When an image is compressed using lossless compression, no pixel data is lost and the picture looks identical after the compression process. However, if the image is compressed using lossy compression, the picture will degrade in quality after compression. Although storage capacities and network speeds keep increasing year by year, the need to compress data is still necessary. This is especially true for image and video files. A digital image, not using any compression to store each pixel value, would typically need 3 bytes to represent a single pixel using 24-bit colour depth (8 bits per colour channel). For an image with a typical desktop resolution of $1920 \times 1080$, that would require $1920 \times 1080 \times 3 \approx 6$ MB of storage. By using lossless image compression, such as PNG, the filesize can be drastically reduced without sacrificing image quality. To compress images even further, lossy compression such as JPEG can be used instead.

2.1.3 Video compression

Uncompressed video data requires substantially more data compared to images. Continuing on the same example as before, a video with a resolution of $1920 \times 1020$ with 24-bit colour depth at a frame rate of 30 frames/s would require roughly 187 MB per second of video. With current technology, the data rate is unfeasible for everyday use. Fortunately, video compression can achieve much higher compression ratios than images. This is thanks to the temporal redundancy that is present in videos; the difference between frames in a sequence can be very small, which video compression algorithms exploit.

2.1.4 Huffman coding

In 1952, David Huffman invented a coding technique that strives to reach the lower bound entropy defined by Shannon. The technique is referred to as Huffman coding and follows the intuition that symbols that occur frequently can be coded with shorter codes than symbols that occur less frequently. Huffman coding ensures that the codes generated provide the shortest average code length for the source set.

To generate Huffman codes for a source symbol set, each symbol in the set is mapped to a binary tree according to a set of rules. The tree is constructed by first creating leaf nodes for each symbol in the set along with their weight which represents the probability of occurrence or frequency in a message. The second step is finding the two nodes with the lowest weight and creating a new parent node, with the combined weight of the children. The branches of each child are labeled as 1 and 0. Next, this step is repeated by finding the next two nodes with the lowest weight and creating a new parent node. When there is only one root node left, the tree generation is complete and each symbol’s code is found by traversing the tree from the root to the symbol and appending the labels on each branch. For example, adopt the following symbol set

$$S = \{a, b, c, d, e, f, g, h\}$$

with their corresponding frequencies

$$p(a) = 30, p(b) = 10, p(c) = 20, p(d) = 6,$$
$$p(e) = 9, p(f) = 7, p(g) = 3, p(h) = 15.$$  \hspace{1cm} (2.2)
2.1. Data compression

The Huffman tree for the set of symbols can be seen in Figure 2.1. By traversing the tree, the codes for each symbol can be constructed, as seen in Table 2.1.

![Huffman Tree](image)

**Figure 2.1:** Huffman tree for the set of symbols \( S = \{a, b, c, d, e, f, g, h\} \).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Huffman Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>001</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>01</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
<td>0001</td>
</tr>
<tr>
<td>e</td>
<td>9</td>
<td>1111</td>
</tr>
<tr>
<td>f</td>
<td>7</td>
<td>1110</td>
</tr>
<tr>
<td>g</td>
<td>3</td>
<td>0000</td>
</tr>
<tr>
<td>h</td>
<td>15</td>
<td>110</td>
</tr>
</tbody>
</table>

**Table 2.1:** Huffman code table for the probabilities found in Equation (2.2).

2.1.5 zlib

zlib[11] is a widely used data compression library that uses the DEFLATE algorithm for compression[10]. The DEFLATE algorithm uses a combination of LZ77[40] together with Huffman coding[16] to compress data.

2.1.6 Run-length encoding

Run-length encoding (RLE) is a simple lossless compression algorithm which encodes data by replacing repetitive sequences of the same data with a tuple that contains a value paired with the length of the sequence[5]. For example, the string "AAABBBBBBCCBBA" can be...
encoded as "3A6B2C2B1A". RLE is also applicable to image compression, where runs of pixel colour values are stored to reduce redundancy in the image.

2.1.7 Discrete Cosine Transform

The discrete cosine transform (DCT)\cite{3} is a method used to represent a discrete signal as a matrix of cosine functions by transforming it from the spatial domain to the frequency domain. The DCT is very prevalent in signal processing and is commonly used in image and video compression.

2.1.8 JPEG

JPEG uses the DCT as part of its compression by first dividing the image into several sub-blocks, typically sized 8 \times 8. Each block can be seen as a discrete signal with 64 evenly spaced data points which can be used as an input to the DCT which transforms it to the frequency domain\cite{38}. Following the transform is a quantization process, where each data point is divided by a corresponding value from a predefined quantization matrix. The values are rounded to the nearest integer, which introduces some round-off errors meaning the process is inevitably lossy\cite{2}. There is no defined quantization matrix defined in the JPEG standard, but the standard does provide two reference matrices. The matrices are derived from subjective image studies conducted by Lohscheller in 1984\cite{21}.

After the quantization process, JPEG performs differential encoding on the top-left value, which is called the DC coefficient and represents the average value of all pixels in the block. Differential encoding is calculating the difference between an element and the previous one: \( \text{DIFF}_i = D C_i - D C_{i-1} \)\cite{2}. The point of encoding the DC coefficient based on the previous one is to take advantage of the spatial correlation between adjacent pixels present in natural images. As a result, the matrix is likely to contain mostly zeroes which is easy to compress efficiently. The 63 remaining values are called the AC coefficients and they are encoded by first ordering them in a zig-zag pattern from the top left to the bottom right, as seen in Figure 2.2. The zig-zag pattern is used as non-zero values are concentrated to the top-left region of the block\cite{38}. After re-ordering the matrix, all nonzero AC coefficients are encoded using RLE. The coefficients are coded by pairing the number of preceding zeroes with the coefficient. For example, the sequence 6 2 0 0 1 0 0 2 is encoded as (0, 6), (0, 2), (2, 1), (3, 2). In the final step, all coefficients are encoded with Huffman coding. Decoding JPEG is done by performing all steps in reverse, using the inverse DCT in the final step.

2.2 X Window System

The X Window System (X) is a system that allows for the drawing of graphics and window management. It started development in 1984 at MIT as a response to the need for a windowing system by two groups at the university. The latest protocol version is 11 (X11) and is maintained by the X.Org Foundation\cite{3}.

DAMAGE extension

The X DAMAGE extension is used to keep track of changes made to any window or pixels. Whenever anything is drawn on the screen, affected areas can be fetched as DamageNotify events that contain at least all modified pixels in the framebuffer\cite{4}. A DamageNotify contains only the region of the framebuffer that is damaged, but not the actual pixel data within the framebuffer.

\[ \text{https://x.org/wiki/} \]
\[ \text{https://www.x.org/releases/X11R7.5/doc/damageproto/damageproto.txt} \]
2.3 Remote desktop software

With the use of remote desktop software, a user (client) can gain access to a desktop environment running on another computer (server) remotely. The software is responsible for transmitting the image from the server to the client and sending the client’s keyboard and mouse inputs to the server. As previously mentioned, there are a lot of software and products that fall under the remote desktop category, differing in protocols used and features present.

2.4 Remote framebuffer protocol

The remote framebuffer (RFB) protocol is used to transfer images and user input to and from a client and server, allowing remote access to graphical systems or applications. It is a very simple protocol that works directly with the framebuffer, which makes it runnable on all windowing systems and applications. The protocol specification is maintained by RealVNC and is published on RFC Editor, although a community protocol specification is available as well. Extending the protocol beyond its specification is easily done through the use of pseudo-encodings. For example, a client and server can agree to use an encoding type that isn’t supported by the specification. The client will send a pseudo-encoding request and will assume that the server doesn’t support it unless it explicitly responds that it does. Servers simply ignore pseudo-encodings that they don’t support. Following is a brief description of how the protocol works.

Initial handshake

When a connection is initialized between a client and a server, a handshake sequence is performed where the client and server agree on the protocol version. Additional parameters such as the width and height of the framebuffer are also sent.

https://github.com/rfbproto/rfbproto
2.4. Remote framebuffer protocol

Display protocol
Communication regarding what to display on a client screen is done through the use of a simple graphics primitive, essentially saying "put a rectangle of pixel data at a given x,y position"[54]. By sending many of these primitives in sequence, the framebuffer can be updated. The client is responsible for demanding updates from the server explicitly, which means the protocol can adapt to changes in network and/or processing speed. The protocol, apart from the initial handshake sequence, is asynchronous, and if multiple requests are piled up on the server side, one update can be made to fulfill them all.

Input protocol
Input handling is handled relatively simply, mimicking the use of a keyboard and mouse. Whenever the user presses a key or button, or when the pointer is moved, an input event is forwarded to the server which performs the input action.

Image encoding
There are twelve encoding types documented in the RFB standard which can be seen in Table 2.2. These encoding types are used when sending rectangles as part of a FramebufferUpdate. Multiple rectangles can be part of a single update, and rectangles can be encoded using different encoding types in the same update. Out of the twelve encoding types, the TigerVNC server supports Raw, CopyRect, RRE, Hextile, ZRLE and Tight.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Raw</td>
</tr>
<tr>
<td>1</td>
<td>CopyRect</td>
</tr>
<tr>
<td>2</td>
<td>RRE</td>
</tr>
<tr>
<td>4</td>
<td>CoRRE</td>
</tr>
<tr>
<td>5</td>
<td>Hextile</td>
</tr>
<tr>
<td>6</td>
<td>zlib</td>
</tr>
<tr>
<td>7</td>
<td>Tight</td>
</tr>
<tr>
<td>8</td>
<td>zlibhex</td>
</tr>
<tr>
<td>16</td>
<td>ZRLE</td>
</tr>
<tr>
<td>21</td>
<td>JPEG</td>
</tr>
<tr>
<td>50</td>
<td>Open H.264</td>
</tr>
<tr>
<td>-260</td>
<td>Tight PNG</td>
</tr>
</tbody>
</table>

Table 2.2: Encoding types defined in the community RFB Protocol standard specification.

Raw
The Raw encoder is very simple, it does not perform any kind of compression and simply sends updates in the form of raw image data.

CopyRect
The CopyRect encoding is used to copy parts of the framebuffer that is available on the client to another position. For example, if a window is moved across the display, the window position, size, and new position can be sent to the client instead of re-transmitting the entire window.
2.5. VNC

RRE

Rise-and-run-length (RRE) encoding is a two-dimensional version of RLE encoding[32]. It works by first dividing an image into smaller regions and finding subrectangles in the region that consist of a single colour. Regions are encoded using a single background colour, followed by the subrectangles in the region.

Hextile

Hextile encoding is a variant of RRE encoding and works by first splitting up the image in $16 \times 16$ tiles. Each tile is encoded using either raw encoding, or RRE, depending on the contents of each tile[32].

ZRLE

zlib run-length encoding (ZRLE) is a combination of zlib compression and RRE[32].

Tight

Tight encoding is versatile and has support for three compression methods: BasicCompression, FillCompression, and JpegCompression. The BasicCompression method has three filters: CopyFilter, PaletteFilter, and GradientFilter. Using CopyFilter means no filter is used, and raw bytes are compressed. With PaletteFilter, pixels are indexed and stored in a palette that can contain up to 256 colours. By using a colour palette, each pixel can be encoded using a single byte instead of three. This filter is only used if the number of different colours in a rectangle is between 2-256. The last filter, GradientFilter, uses a type of differential encoding to filter the image so that it becomes easier to compress. For example, a simple one-dimensional greyscale gradient $[0, 1, 2, 3, 4, 5, 6, 7]$ can be encoded as $[0, 1, 1, 1, 1, 1, 1, 1]$, which is trivial to compress. The next method, FillCompression, is very simple and is used for rectangles that consist of a single colour. It simply sends the pixel value that corresponds to the colour of the rectangle along with the rectangle size and position. Lastly there is JpegCompression which is used for rectangles with more than 256 colours and if the JPEG quality level pseudo-encoding is sent by the client. It works by simply sending JPEG-encoded image data. The final step in Tight encoding is to encode all data using zlib compression.

2.5 VNC

Virtual network computing (VNC) is a family of software which consists of a thin client and a server that uses the RFB protocol to communicate. The original source code was originally developed by the Olivetti & Oracle Research Lab (ORL) in 1998[33], and later released to the public under a GPL license. As GPL allows for modifications and redistributions, and modified versions that are published have to use the same GPL license, forks of the original source code have developed over the years. Some of the most popular VNC versions today include the original VNC which today is called RealVNC[9] and forks like TurboVNC[10] and TigerVNC[11].

[33] https://www.gnu.org/licenses/old-licenses/gpl-2.0.html
[34] https://www.realvnc.com/
[35] https://www.turbovnc.org/
[36] https://www.tightvnc.com/
2.6 TigerVNC

TigerVNC is a platform-independent VNC server/client application. The project is free and open source and is currently the most starred VNC server/client project on GitHub\footnote{https://github.com/TigerVNC/tigervnc}. High performance is achieved through clever use of image encoding even in low bandwidth situations, originally based on the Tight encoder\footnote{https://tigervnc.org/doc/Xvnc.html}. Following is a high-level overview of the architecture of TigerVNC.

2.6.1 High-level architecture

Unlike a traditional VNC server, TigerVNC uses its own X server called Xvnc\footnote{https://tigervnc.org/doc/Xvnc.html} instead of relying on an existing X server. Xvnc uses a virtual display instead of a physical one and can only be seen by connecting through a VNC client. By listening to draw calls being made to the virtual framebuffer, TigerVNC marks regions that have been changed similarly to how the X DAMAGE extension, described in Section 2.2, can be used. These marked changes are sent to an update tracker, which keeps track of which regions have changed since the last update was sent to the client. When the server sends an update to the client, only the regions in the framebuffer that have changed since the last update are sent. An overview of how the TigerVNC server and client work can be seen in Figure 2.3.

![Figure 2.3: High-level overview of TigerVNC architecture.](image)

EncodeManager

Before an image is encoded and sent to the client, it is divided into smaller rectangles of varying sizes. The rectangles are then encoded and transmitted to the client, which then updates its own framebuffer when all rectangles for a single update have been decoded. To determine how to split up the changed regions, TigerVNC uses a module called the EncodeManager which employs a heuristic derived from the Tight encoder\footnote{https://tigervnc.org/doc/Xvnc.html}. It is the EncodeManager that analyses the changed regions and decides how to split it up and encode it.
Solid coloured rectangles

The `EncodeManager` starts by identifying blocks that consist of a single colour. These blocks are transmitted using the `LastRect` pseudo-encoding and can be more efficient to encode compared to conventional image encoding as only the rectangle size, position, and colour need to be transmitted. Following is the algorithm TigerVNC uses to identify solid coloured blocks:

1. Start with a 16 x 16 block in the top left corner of the rectangle.  
2. Check if the block is solid. If false, continue searching in 16px increments, row-wise first, until a solid block is found.  
3. When a solid 16 x 16 block is found, keep extending it by 16px row-wise and column-wise until the combination that gives the largest area of a solid colour is found.  
4. Do the same thing as step 3, but extend the area one pixel and in one direction at a time instead of one block at a time.  
5. Finally, the solid-coloured rectangle is encoded and sent to the client.

Multi-coloured rectangles

The remaining rectangles that aren’t entirely composed of a single colour are encoded differently. To decide which encoder to use for each remaining rectangle, TigerVNC uses the following heuristic:

1. The dimensions of the remaining region is first assessed to determine if it should be split into sub-rectangles before encoding. If the width is less than or equal to 2048px and the area is less than or equal to 65536px, the rectangle can be encoded directly and the next step can be skipped.  
2. If the region’s width is greater than 2048px, it is split into 2048 x 32 rectangles, and for widths less than 2048px, each rectangle will have the same width as the region and a height of 65536/width.  
3. The encoding of each rectangle starts with an analysis of the number of colours present in the rectangle, using the `analyseRect` function. Using the default settings in TigerVNC client, the system will default to JPEG encoding if the number of colours is greater than 256. In all other cases, Tight encoding is used.

2.7 Performance metrics

When trying to measure the performance of an application, or an encoder as is the case in this thesis, the question of how to measure performance comes to light. Measuring CPU performance for instance is not always a trivial task, especially on a concurrent multi-core system. Sources of error, such as those coming as a result of CPU scheduling, have different effects on code execution time depending on how it is measured. As such, when measuring CPU time, which is how long the program takes to run on the CPU, can be different from how much real time has elapsed (wall time). While both are interesting metrics, distinguishing the difference is important. In this thesis, CPU time will be measured.

An important performance metric for an encoder is its compression ratio; the ratio between the size of data before compression and after compression. Encoding an uncompressed
image of 6 MB down to 2 MB would have a compression ratio of $6/2 = 3$. Weighing the compression ratio against CPU performance is interesting as higher compression ratios usually are more demanding to compute, and finding a balance can be difficult.

Another well-used metric for encoders is the number of frames encoded per second (FPS). For most scenarios, the FPS combined with the resolution is a sufficient metric to accurately describe the performance of an encoder. However, a problem arises if the encoder has to encode images of different sizes; larger images are generally more demanding to process. A metric that can be used instead that scales with different image resolutions is how many pixels per second are processed (Mpx/s). For a remote desktop application such as VNC, using Mpx/s instead of FPS is a better metric as not all parts of the screen are updated (encoded) each time the framebuffer is updated.
This chapter introduces previous relevant research related to the topic of this thesis. Plenty of research has been done in the area of remote desktops, with some work focusing more on the client/user, and others focusing more on the server performance.

Endo et al. [13] argue that latency and interactability are good metrics to measure the performance of interactive systems. Work previous to theirs focused mostly on throughput when measuring performance, which they claim limits what kind of analyses can be performed. They introduced new methods to measure and benchmark performance regarding input latency. In their experiments, they measured the latency and CPU usage of several input events, such as maximizing a window. Several shortcomings are brought up by the authors, a major one being the lack of application source code. Without the source code, it was difficult to make precise time measurements, and time had to be measured through the operating system’s API calls.

Schmidt et al. [35] propose the Stateless Low-level Interface Machine (SLIM) architecture; a low-level protocol that can be used to display GUI applications remotely with good performance. SLIM is very similar to VNC, as both use protocols that are independent of any windowing and operating system. A key metric in their experiments was interactability, and their tests included running applications for image processing, web browsing, video streaming and 3D games. The compression ratio of the protocol was also measured to figure out the performance in low bandwidth scenarios. Unlike Endo et al. [13], latency measurements could be instrumented as the authors wrote the SLIM driver themselves. This allowed for very precise time-logging of messages. The paper concludes that in their testing, the limiting factor for user experience was server bottlenecks, and by encoding display updates, performance is good even with fairly low bandwidths.

A new methodology to study the performance of thin-client systems was created by Nieh et al. [26]. Their findings were that their new methodology, slow-motion benchmarking, is a better tool to use to analyze performance and fixes shortcomings of previous industry standards. Compared to previous methods, slow-motion benchmarking focuses more on the user experience, rather than the server’s performance. For example, a previous approach was to measure a video’s frame rate running on the server. Since what the server is displaying and what the client is seeing necessarily are not the same, the measurement isn’t indicative of real-world performance. Another approach mentioned is to monitor network traffic between a server and a client. By looking at the time between a web page download was initiated, and
the time the web page was displayed on the client, user experience could be measured by latency and interactivity. To combat these shortcomings, the authors came up with a new methodology; slow-motion benchmarking. This new method works by introducing a delay on the server side for display updates, such as loading a web page or displaying a new frame of a video. Introducing a delay ensures that display updates on the client are fully complete for each frame sent from the server. What this means in practice is that every display update from the server should be fully completed and displayed on the client. While running these benchmarks, latency and data transferred are measured for each display update. According to the authors, the new methodology improves previous methods by making tests more accurate and repeatable. Furthermore, there is no need to make changes to the source code to perform any benchmark, which makes it appropriate to use with most remote desktop software. The authors also argue that slow-motion benchmarking is closer than previous methods to real-world usage such as web browsing.

In another study by Lai and Nieh [20], slow-motion benchmarking was used to measure the performance of different remote desktop systems in both WAN and LAN environments. Three categories were tested: latency, web browsing and video playback. What they found was that in general, WAN performance is acceptable even in high latency situations, but results varied heavily between remote desktop systems. The authors state that focusing on improving latency over bandwidth is more important for WAN remote desktop systems.

More recent studies have not only looked at traditional thin-client server applications but have also applied the same techniques to virtual desktop environments. In a paper written by Berryman et al. [6], a benchmarking toolkit was made to evaluate virtual thin-client server environments. The toolkit, named VDBench, is meant to measure the server capacity, network performance, and user experience metrics like response time and subjective picture quality. Unlike previous methods, the study utilizes constrained memory and network capacities on the server side to analyze virtual desktop performance. To model real-world desktop usage, an automated randomized process was developed to perform several “application tasks”. The tasks include launching applications in random order, and performing different tasks in each application, such as opening a spreadsheet using Microsoft Excel, filling in data and generating graphs. The authors argue that the toolkit, VDBench, can be used by service providers to help with determining how much hardware should be accommodated to fit their workloads.

Similarly to Berryman et al. [6], the authors of [27] created a benchmarking framework for virtual desktop infrastructure (VDI) systems. In their paper, VMWare’s VDI solution[1] was used. The framework includes tools to generate real-world VDI workloads, performance metrics collection, as well as analysis tools. Performance metrics include server-client response time, application latency, framerate, CPU and RAM usage. The authors argue that end-user experience is important in the adoption of VDI and that measuring performance is not a trivial task. The difficulty stems from the diversity of VDI systems; the use of different hardware, operating systems, software, and different client needs. With the use of their framework, the authors wanted to create a way to easily build automated benchmarks using virtual desktops with predetermined workloads. Different system configurations can be used with the workloads to compare and optimize a system to fit a use case and workloads can be very diverse and are simple to implement using their API. Some of the workloads mentioned in the paper include 3D CAD applications, such as SPECapc for Autodesk 3DS Max[2] and machine learning applications such as object recognition using the CIFAR-10 dataset[19].

Alali et al. [4] stress the importance of using both subjective and objective measures to evaluate the performance of a zero-client connected to a VDI system. Their benchmarks include techniques used by previous research, such as slow-motion benchmarking[26] and latency measurements used in VDBench[6], as well as subjective methods used by Casas et al.[7, 8], where participants were instructed to perform numerous tasks on a virtual desktop.
In both studies, the ACR-9 Mean Opinion Score (MOS)\textsuperscript{[31]} scale was used for subjective metrics. Tests were repeated using different packet loss rates (PLR), ranging from 0\% to 10\% and what they found was that the MOS at a PLR of 0.5\% and up did not differ much, with the exception of video applications that had a much greater negative effect on the MOS.

Several studies have been conducted comparing the performance of the encoders in a VNC implementation. In 2001, Kaplinsky\textsuperscript{[18]} compared his newly developed Tight encoder against the Hextile and Zlib encoders. In his study, a tool called \texttt{rfbproxy}\textsuperscript{3} was used to capture traffic between a VNC server and a client such that the images transmitted between them could be re-constructed. By recording a session, it could be sent through a VNC server again using different encoders, allowing for performance comparisons between them. The study concluded that the Tight encoder showed good compression results, albeit with a higher execution time.

A study using the same methodology as Kaplinsky was conducted by the VirtualGL Project in 2008\textsuperscript{[30]}. In addition to the datasets provided by Kaplinsky, the study added a dataset of 3D application workloads, which included applications from the SPECviewperf\textsuperscript{9}\viewset\textsuperscript{4}. The study compared TightVNC to TurboVNC and the encoders were compared using varying compression levels and JPEG quality settings, using compression ratio and compression time as metrics. Experimentation with the Tight colour palette was conducted which gave improvements in both compression ratio and compression time in most cases. In the end, a palette threshold of 24 was deemed to be the best.

The VirtualGL project conducted a similar study in 2011\textsuperscript{[29]}, this time comparing TigerVNC to TurboVNC using varying compression levels and compression time. It was concluded that a compression level greater than two had diminished effects and levels greater than six showed no improvements at all.

\textsuperscript{3}\url{http://cyberelk.net/tim/software/rfbproxy/}
\textsuperscript{4}\url{https://www.spec.org/gwpg/publish/vp9_rel.html}
The testing framework developed as part of this thesis is split into two parts. First, a way of creating reproducible tests was developed through the use of a tool that records an X session losslessly. The recorded session could then be used for testing and analysis, which is the second part of the framework. It is possible to choose which encoders to test and which settings to use for each encoder. After running different tests on multiple recorded sessions, an analysis will be conducted.

4.1 Recording an X session

The purpose of recording an X session is to capture real inputs and store them in a manner that can reproduce the session at a later time. When reproducing the session again, the aim is to make the input as close as possible to how it looked when the session was first recorded, making it possible to "re-play" the session again. From the VNC server’s point of view, reading the session file should be indistinguishable from listening to a real X server. The recording tool is built to follow these criteria.

4.1.1 X11 DAMAGE

The recorder works by first connecting to an existing X server. After the connection has been established, the program waits and listens for DamageNotify events, as described in Section 2.2. When an event is detected, the damaged region is copied from the framebuffer, encoded, then stored to disk. Remember that DamageNotify events only contain information about which parts of the framebuffer have changed, but not the actual data in the framebuffer at the time the event was detected. This means that special measures have to be taken if multiple DamageNotify events are detected at the same time. Consider the scenario where three DamageNotify events are detected and the regions affected in the events are overlapping. As the events are processed one by one, future events need to be taken into consideration since the affected region will contain information from future updates. A scenario like this is illustrated in Figure 4.1. To combat this from happening, the recorder aggregates all events when they are detected and combines all damaged regions into a bounding rectangle that contains all damage, as seen in Figure 4.2.
4.1. Recording an X session

4.1.2 Update interval

When using the recorder tool, it is necessary to supply a desired frame rate to the tool which specifies how long to wait between each frame update. For example, if the desired frame rate is 60 frames per second (FPS), the recorder will attempt to detect damaged regions and store them each $\frac{1}{60}$ th second. To achieve the desired frame rate, the tool keeps track of how much time it takes to perform a frame update. If the time taken is less than the frame interval, the program sleeps until it is time for the next update. However, if the time taken exceeds the frame interval, the program does not sleep but instead looks for the next update immediately. Updates that are not encoded in time are noted as delayed frames and can be an indicator of the quality of a recorded session. Following is the pseudo-code for the recorder tool’s main loop:

```cpp
interval = 1 / FPS;
while (true) {
    start = now(); // Start timer
    vector<XEvent> events;
    while (XPending()) // Load all XEvents
        events.push(XNextEvent());
    if (events.empty()) // If no events were detected, try again
        continue;
    // Detect overlaps & aggregate if necessary, then store to disk.
    handleEvents(events);
    end = now(); // Stop timer
    encodeTime = end - start;
    // Sleep until next update.
    if (encodeTime <= interval) {
        sleepTime = interval - encodeTime - sleepDuration;
        sleep(sleepTime);
        sleepDuration = sleepTime;
    } else {
        sleepDuration = 0;
    }
}
```

4.1.3 File format

The file format used by the recording tool is a very basic one and has support for multiple image encodings. It is meant to mimic how the RFB protocol handles framebuffer updates.
Header

First, there is a header that tells which encoding is used for each image, followed by the size of the framebuffer for the X session that was recorded, and lastly the frame interval, i.e. the time between each frame.

Image metadata

What follows is some metadata for the next image in the sequence, such as the image size, the position, as well as some statistics for the captured frame. The statistics include:

- **timeBudget** – How much time will pass until the next frame.
- **encodingTime** – How long it took to encode the frame in milliseconds.
- **margin** = **timeBudget** − **encodingTime**, i.e. how much time after encoding is left until the next frame update. A negative number indicates how much we exceeded our desired frame interval.

The **timeBudget** can be used to induce an artificial delay between each frame update when reading the file, allowing a real-time replay of the recorded session.

Image

Lastly, the encoded image is stored. The image can be re-read by reading exactly as many bytes as is specified in the image metadata. Subsequent updates are stored in the same manner; their metadata is followed by the encoded image.

4.1.4 Encoding

The recorder supports a variety of image encodings, with the default being libjpeg-turbo\(^1\), which was used for all recorded sessions in this thesis. The JPEG settings used were a quality level of 100, and a subsampling setting of zero, meaning no subsampling. These settings should provide lossless encoding, except for round-off errors caused by the quantization process in the DCT transformation as described in Section 2.1.8.

4.2 X session benchmark

The second part of the testing framework is responsible for processing a recorded session file and running benchmarks on it. Following is a high-level overview of how it works.

1. Load session file & parse header.
2. Load benchmark settings.
3. Initialize encoders & framebuffers according to settings.
4. Read next image in sequence.
5. For each encoder, load image into framebuffer & encode image.
6. If there are images left in session file, go to 4.
7. For each encoder, save output to disk.

Running the tool is very simple and only requires the input file as an argument. For example, running the benchmark suite on a file called `session.txt` is done by running 

```bash
./suite session.txt
```

\(^1\)\url{https://libjpeg-turbo.org/}
4.2 X session benchmark

4.2.1 Benchmark settings

Benchmarks can run multiple times on the same session file using different settings. The settings that are available for each benchmark are which encoders should be supported and which encoder is preferred. For the Tight and ZRLE encoders, a compression level can be set. For the JPEG encoder, the JPEG quality level is also supplied. The supported compression and quality levels are 0 – 9.

Default settings

The default settings in TigerVNC, assuming there are no bandwidth limitations, is to prefer using Tight encoding whenever possible, except for areas in the framebuffer that have more than 256 colours. In those cases, JPEG encoding is used with a quality level of eight, which is equivalent to using quality level 92 in libjpeg-turbo. The quality level 92 is based on research made by The Virtual GL project[29, 30] and should produce perceptually lossless images. This setting will be benchmarked in this thesis and is referred to as the "default encoder".

4.2.2 Benchmark measurements

As mentioned in Section 2.7, the performance metrics that are evaluated in this thesis are CPU time, compression ratio, and encoded Mpx/s.

CPU time

The CPU time is measured by timing how long calls to the `writeUpdate()` function take, which is called once per framebuffer update.

Compression ratio

To measure compression ratio, each encoder keeps track of the size of each raw image that is being encoded, as well as the size of the encoded image. This measurement is taken each time a rectangle is encoded.

Megapixels per second

Calculating the number of pixels encoded per second is trivial and is essentially another way of expressing CPU time. It is done by simply dividing the pixel count by the time it took to encode those pixels. This metric can give better insight into how different encoders perform in given varied input data, as some inputs are more difficult to encode than other.

This chapter presents the sessions that were benchmarked, as well as the motivation for why and how the sessions were chosen. The benchmarking method, experiments conducted, and the testing environment are also described.

5.1 Test scenarios

A total of six different use cases were recorded and benchmarked. The use cases are comparable to related studies that perform remote desktop performance tests. The software used for the use cases are typically divided into categories like "general office workloads", which include things like using a word processor, spreadsheets, reading PDFs, web browsing and reading email. Video, photo and 3D CAD workloads are also common use cases. The six use cases in this thesis are based on the benchmarked applications used in previous studies. A table showing the summary of the use cases can be seen in Table 5.1.

<table>
<thead>
<tr>
<th>Use case</th>
<th>Application</th>
<th>Time (minutes)</th>
<th>Size (Gb)</th>
<th>Frame updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word processing</td>
<td>vscode</td>
<td>13.9</td>
<td>1.4</td>
<td>12346</td>
</tr>
<tr>
<td>3D CAD</td>
<td>Blender</td>
<td>10</td>
<td>5.2</td>
<td>4500</td>
</tr>
<tr>
<td>Photo editing</td>
<td>GIMP</td>
<td>3.2</td>
<td>0.2</td>
<td>1048</td>
</tr>
<tr>
<td>Video editing</td>
<td>Kdenlive</td>
<td>7.5</td>
<td>5.1</td>
<td>7830</td>
</tr>
<tr>
<td>Video streaming</td>
<td>mpv</td>
<td>10.4</td>
<td>26</td>
<td>18550</td>
</tr>
<tr>
<td>Web browsing</td>
<td>Firefox</td>
<td>9.8</td>
<td>2.8</td>
<td>3528</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the test scenarios benchmarked in this thesis.

5.1.1 Word processing – Visual Studio Code

The word processing use case was captured using Visual Studio Code (vscode) and lasted for a total of 14 minutes. In this use case, the session captured writing source code for the benchmarking application used in this thesis. The recording frame rate was 60 FPS and a total of 12346 framebuffer updates were captured. A screenshot from the word processing case can be seen in Figure 5.1.

[https://code.visualstudio.com/](https://code.visualstudio.com/)
5.1. Test scenarios

5.1.2 3D CAD – Blender

For the 3D CAD use case, the application Blender was recorded for a total of ten minutes. In this scenario, a demo bundle from Blender was used called Cube Diorama. The bundle includes a scene with an unfurnished room, as well as premade objects and materials that can be placed directly in the scene. Also included is an image of the room furnished, using some of the objects included in the bundle. The recorded session for this scenario starts with the unfurnished room and continues with an attempt to re-create the furnished room from the included example image. As with the previous use case, the targeted frame rate was 60 FPS, and a total of 4500 framebuffer updates were captured. An important note, in this case, is that Blender is a GPU-intensive program, and with the hardware used resulted in a low update frequency for the application. This means fewer updates were recorded compared to other use cases. A screenshot from the 3D CAD case can be seen in Figure 5.2.

Figure 5.1: Screenshot from the word processing case.

Figure 5.2: Screenshot from the 3D CAD case.

[https://www.blender.org/](https://www.blender.org/)
5.1.3 Photo editing – GIMP

GIMP\textsuperscript{4} was used for the photo editing scenario. The session lasted for roughly three minutes and consists of following the tutorial "Tone Mapping and Shadow Recovery using GIMP’s ‘Colors/Exposure’\textsuperscript{5}. It is a simple tutorial where you learn some basic color correction by applying some filters to an image of a tree. The session was recorded with a target frame rate of 60 FPS and includes 1048 framebuffer updates. A screenshot from the photo editing case can be seen in Figure 5.3.

![Screenshot from the photo editing case.]

5.1.4 Video editing – Kdenlive

The video editing use case was done with the use of Kdenlive\textsuperscript{6}. In this scenario, the quick start guide\textsuperscript{7} is followed, and the session took roughly seven minutes. A target frame rate of 60 FPS was used, and a total of 7830 framebuffer updates were captured. A screenshot from the video editing case can be seen in Figure 5.4.

![Screenshot from the video editing case.]

\textsuperscript{4}https://www.gimp.org/
\textsuperscript{5}https://www.gimp.org/tutorials/Tone_Mapping_Using_GIMP_Levels/
\textsuperscript{6}https://kdenlive.org/
\textsuperscript{7}https://userbase.kde.org/Kdenlive/Manual/QuickStart
5.1.5 Video streaming – mpv

The media played mpv[8] was used for the video streaming use case. The session consists of recording a full-screen video playback of Blender Foundation’s Big Buck Bunny short film[9], running at 1080p at 60 FPS. A total of 18550 framebuffer updates were captured. A screenshot from the video streaming case can be seen in Figure 5.5.

Figure 5.5: Screenshot from the video streaming case.

5.1.6 Web browsing – firefox

The web browsing use case was performed using Firefox[10] and consists of browsing random articles on Wikipedia[11]. A frame rate of 60 FPS was used, and the session lasted for ten minutes. A total of 3528 framebuffer updates were captured. A screenshot from the web browsing case can be seen in Figure 5.6.

Figure 5.6: Screenshot from the web browsing case.

[8] https://mpv.io/
[10] https://www.mozilla.org/firefox
5.2 Benchmarking

To gather performance measurements, the six recorded sessions were run through the testing suite several times using different settings.

5.2.1 Comparison against default encoder

The first experiment uses all available encoders in TigerVNC: Raw, RRE, Hextile, Tight, JPEG and ZRLE encoding. The pseudo-encoding LastRect was enabled for all encoders. For the JPEG encoder, the default quality level of eight was used. In addition to the six encoders, a seventh benchmark was run using the default encoder. The encoders that use zlib compression, Tight, ZRLE, and the default were benchmarked using all available zlib compression levels 0 – 9.

5.2.2 Brute-force upper bound compression

To calculate what the upper limit for compression is, an experiment using a brute forcing method will be used. In this experiment, each rectangle will be encoded once by each encoder, and the metrics from the encoder with the highest compression ratio are saved. This means that the upper bound limit for compression ratio can be calculated, along with the execution time.

5.3 Test environment

All benchmarks in this thesis were run on a typical desktop machine, using common open-source tools.

Hardware

- **CPU**: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz.
- **GPU**: Intel(R) UHD Graphics 630.
- **RAM**: 16GB DDR4 @ 2667MT/s.
- **SSD**: Samsung PM891a 512GB.

Software

- **Operating system**: Fedora 37.
- **X server**: Xephyr (xorg-server 1.20.14).
- **TigerVNC version**: 1.13.

Build system

- **CMake version**: 3.10.2.
- **GCC version**: 5.5.0.
6 Results

This chapter presents the results of all benchmarks. Section 6.1 presents the results for the benchmarks using the default settings, which compares the encoders to each other in each test case. In Section 6.2, results are shown for the benchmarks using compression levels 0-9. Following, a comparison between each test case can be seen in Section 6.3, which shows the difference in encoded MPx/s for each encoder. Lastly, Section 6.4 shows the results for the brute-force experiment, which marks the upper-bound compression ratio for the EncodeManager.

6.1 Default settings

In this section, the benchmark results are presented for all seven encoder settings using their default settings. The defaults are a compression level of two for the Tight encoder, a compression level of six for the ZRLE encoder, and a quality level of eight for the JPEG encoder. Changing the compression level setting only affects the Tight, default, and ZRLE encoders, meaning the results for the Raw, RRE, Hextile and JPEG encoders are the same regardless of the compression setting. In the tables, the symbol $M$ denotes the median value, and the mean, median, and standard deviation values are not calculated for results without any delays. The Raw encoder does not perform any compression and is only shown for reference.

Word processing

First, the benchmark for the word processing scenario results are looked at. The RRE, Hextile, and default encoders all display similar results in terms of execution times. However, looking at compression ratios we see that RRE performs the worst, followed by Hextile which compresses more than twice as much data, and the default which in turn compresses twice as much as Hextile. JPEG and Tight are about twice as slow, with JPEG compressing 30% more than Hextile, and Tight being even better, compressing 78% more than JPEG. ZRLE is much slower than the others but has the best compression ratio, beating Tight by 8% while being more than twice as slow. ZRLE has many more delayed updates than the rest, with a mean delay of 12.87 ms, meaning that some frame updates took longer to encode than what would be necessary for real-time encoding of the session. A summary of the results can be seen in Table 6.1.
6.1. Default settings

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Time (ms)</th>
<th>Ratio</th>
<th>Delay %</th>
<th>µ (ms)</th>
<th>M</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>2.38</td>
<td>1.00</td>
<td>0.03</td>
<td>0.58</td>
<td>0.81</td>
<td>0.32</td>
</tr>
<tr>
<td>RRE</td>
<td>13.01</td>
<td>6.87</td>
<td>2.11</td>
<td>2.15</td>
<td>1.99</td>
<td>1.38</td>
</tr>
<tr>
<td>Hextile</td>
<td>17.18</td>
<td>15.06</td>
<td>3.09</td>
<td>2.84</td>
<td>2.67</td>
<td>1.93</td>
</tr>
<tr>
<td>Tight</td>
<td>30.42</td>
<td>34.99</td>
<td>5.76</td>
<td>5.44</td>
<td>4.87</td>
<td>3.74</td>
</tr>
<tr>
<td>JPEG</td>
<td>28.36</td>
<td>19.64</td>
<td>5.94</td>
<td>4.37</td>
<td>3.67</td>
<td>3.12</td>
</tr>
<tr>
<td>ZRLE</td>
<td>70.83</td>
<td>37.65</td>
<td>13.41</td>
<td>12.87</td>
<td>11.61</td>
<td>8.12</td>
</tr>
<tr>
<td>Default</td>
<td>14.71</td>
<td>31.45</td>
<td>2.50</td>
<td>2.74</td>
<td>2.55</td>
<td>1.75</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of benchmark for the word processing test case using the default settings.

3D CAD

The results are more varied for the 3D CAD test case than the previous one and are summarized in Table 6.2. The two fastest encoders are JPEG and the default, being close in execution times. The default encoder manages to compress 18% more data compared to JPEG while being 16% faster. RRE and Hextile also come close in execution times, but interestingly, RRE has a compression ratio of 0.77, meaning it increased the number of bits per pixel for the image stream which is worse than using the Raw encoder. The Tight encoder and ZRLE encoder stick out with their longer execution times, ZRLE taking more than twice as long to execute compared to Tight but managing to compress 13% more data. Both encoders also have lots of delays, with ZRLE having a median delay three times greater than Tight.

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Time (ms)</th>
<th>Ratio</th>
<th>Delay %</th>
<th>µ (ms)</th>
<th>M</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>2.78</td>
<td>1.00</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RRE</td>
<td>92.95</td>
<td>0.77</td>
<td>7.02</td>
<td>2.70</td>
<td>2.89</td>
<td>1.50</td>
</tr>
<tr>
<td>Hextile</td>
<td>107.98</td>
<td>1.93</td>
<td>11.96</td>
<td>4.15</td>
<td>4.17</td>
<td>2.50</td>
</tr>
<tr>
<td>Tight</td>
<td>255.05</td>
<td>4.51</td>
<td>33.74</td>
<td>28.54</td>
<td>29.76</td>
<td>8.58</td>
</tr>
<tr>
<td>JPEG</td>
<td>51.68</td>
<td>14.67</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ZRLE</td>
<td>562.68</td>
<td>5.09</td>
<td>44.17</td>
<td>82.40</td>
<td>93.21</td>
<td>32.05</td>
</tr>
<tr>
<td>Default</td>
<td>43.38</td>
<td>17.31</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of benchmark for the 3D CAD test case using the default settings.

Photo editing

The photo editing test case shows the fastest encoders to be the default and Hextile. In terms of compression ratio, the default beats Hextile by a factor of 4.5. RRE and JPEG fall slightly behind in execution times, with RRE having the lowest compression ratio and JPEG being the second best. Tight and ZRLE once again show slower execution times, with ZRLE being twice as slow as Tight, but both perform equally in compression. A summary of the results can be seen in Table 6.3.
6.1. Default settings

Table 6.3: Summary of benchmark for the photo editing test case using the default settings.

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Time (ms)</th>
<th>Ratio</th>
<th>Delay %</th>
<th>μ (ms)</th>
<th>M</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>0.24</td>
<td>1.00</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RRE</td>
<td>2.11</td>
<td>2.19</td>
<td>5.25</td>
<td>2.34</td>
<td>1.66</td>
<td>2.02</td>
</tr>
<tr>
<td>Hextile</td>
<td>1.81</td>
<td>5.89</td>
<td>2.96</td>
<td>3.10</td>
<td>3.69</td>
<td>1.72</td>
</tr>
<tr>
<td>Tight</td>
<td>5.98</td>
<td>11.10</td>
<td>14.69</td>
<td>8.13</td>
<td>7.58</td>
<td>6.48</td>
</tr>
<tr>
<td>JPEG</td>
<td>2.63</td>
<td>19.08</td>
<td>5.63</td>
<td>3.86</td>
<td>2.83</td>
<td>2.50</td>
</tr>
<tr>
<td>ZRLE</td>
<td>10.30</td>
<td>11.09</td>
<td>23.76</td>
<td>16.67</td>
<td>11.79</td>
<td>9.82</td>
</tr>
<tr>
<td>Default</td>
<td>1.47</td>
<td>26.35</td>
<td>2.39</td>
<td>2.36</td>
<td>2.45</td>
<td>1.43</td>
</tr>
</tbody>
</table>

Table 6.4: Summary of benchmark for the video editing test case using the default settings.

Video editing

In the video editing test case, the default encoder has the fastest execution time. Following are the RRE, JPEG and Hextile encoders. While RRE is the second fastest encoder, it performs much worse in terms of compression. JPEG manages to compress almost seven times more data compared to RRE while being 16% slower. The default encoder performs the best in terms of compression ratio, followed by JPEG. Once again, the Tight and ZRLE encoders have much slower execution times, ZRLE being roughly twice as slow as Tight performing similarly in compression. While both Tight and ZRLE have a majority of their frame updates delayed, the delay time is much greater for the ZRLE encoder. The results are summarized in Table 6.4.

Video streaming

Results for the video streaming test case are displayed in Table 6.5. Compression ratios in this case, and similarly in the 3D CAD case, are much lower than the others. The REE encoder performs terribly with a compression ratio of 0.57. Hextile, having a similar execution time as RRE also performs badly, but still manages to compress 31% of the data. JPEG and the default encoder perform the best, both in terms of execution time and compression ratio, with JPEG 5% slower and compressing 3% less data. Lastly, Tight and ZRLE have much longer execution times and boast comparably subpar compression ratios. These encoders also have almost all frame updates delayed, with ZRLE having double the delay time compared to Tight.
6.2 Varying compression

This section presents the benchmark results using all compression levels from 0-9. The encoders that are affected by compression levels are the Tight encoder, ZRLE, and the default which uses both Tight and JPEG. RRE, Hextile and JPEG will be displayed in the figures as well for reference but will be constant across all levels.

6.2.1 Word processing

Figure 6.1 shows the execution times and compression ratios for compression levels 0-9. We can immediately notice two things; at level zero, Tight and ZRLE barely do any compression at all, and at level nine, execution times increase immensely with very small improvements in compression ratio. The figure shows that the ZRLE encoder always has a slower execution time compared to Tight, except for compression level nine, and that the encoders perform very similarly in compression, trading blows between compression levels. Interestingly, the compression ratio for ZRLE decreases between levels three and four. The default encoder is consistently faster than the others and has a consistent execution time until a compression
level of seven or higher. Going past level two does not give any significant gains in compression for the default encoder. For Tight and ZRLE, we see very small gains in compression ratio as higher levels are used, but with execution times that increase significantly.

![Figure 6.1: Execution times and compression ratios in the word processing test case for all encoders, using compression levels 0-9.](image)

### 6.2.2 3D CAD

For the 3D CAD test case, seen in Figure 6.2, we see a steeper increase in execution time as the compression level increases, but without any significant gains in compression ratio past level two. Once again, ZRLE is the slowest, with Tight following not too far behind. The default is the fastest and outclasses the other encoders in compression. For ZRLE and Tight, a compression level greater than one does not increase the compression ratio significantly.

![Figure 6.2: Execution times and compression ratios in the 3D CAD test case for all encoders, using compression levels 0-9.](image)
6.2.3 Photo editing

The results for the photo editing test case can be seen in Figure 6.3 and display a similar trend as in the 3D CAD case. For the default encoder, a compression level greater than two does not increase the compression ratio, and the execution time remains constant from level zero to six. ZRLE is slightly slower than Tight across all levels except nine and they both perform equally in compression. Going past level two does not benefit compression for ZRLE and Tight.

![Figure 6.3: Execution times and compression ratios in the photo editing test case for all encoders, using compression levels 0-9.](image)

6.2.4 Video editing

The same patterns are visible for the video editing case, shown in Figure 6.4, as in the photo editing and 3D CAD case. The compression ratio for the default encoder does not increase significantly past level two, and the execution time remains constant except between levels six and eight where it doubles. For both ZRLE and Tight, the compression ratio increases by 10% and 12% respectively between levels one and nine, while the execution time in the same span increases by 454% and 545%.

![Figure 6.4: Execution times and compression ratios in the video editing test case for all encoders, using compression levels 0-9.](image)
6.2.5 Video streaming

In the video streaming test case, shown in Figure 6.5, the results are very similar to the 3D CAD case. A difference here is that the compression ratio does not increase past level one. The compression ratio for the default encoder is just slightly lower than in the 3D CAD case and just under half the ratio for ZRLE and Tight. The difference between the default and the rest is also much greater both here and in the 3D CAD case.

Figure 6.5: Execution times and compression ratios in the video streaming test case for all encoders, using compression levels 0-9.

6.2.6 Web browsing

Figure 6.6 shows the results for the web browsing test case and shows similar results to the word processing case. Tight, ZRLE and the default encoder are closer in compression ratio across the levels compared to the other cases. Unlike the word processing case, the default encoder has the highest compression ratio and lowest execution time across all compression levels except level nine.

Figure 6.6: Execution times and compression ratios in the web browsing test case for all encoders, using compression levels 0-9.
6.3 Test case comparison - MPx/s

While execution time is an interesting metric in itself, it is difficult to use it to compare the results between test cases; each test case differs in length, number of pixels encoded, and the pixel data itself. This section presents a comparison between the encoders using encoded MPx/s as a metric.

6.3.1 Tight encoder

The encoded MPx/s for compression level zero are much higher in all compared to the other levels, but have very low compression ratios, as shown in Section 6.2. To make changes easier to see, the results have been split up into two graphs, seen in Figure 6.7, one where all compression levels are shown and another where compression level zero is excluded. Following results will have level zero excluded as well. We can see that the photo editing and video editing cases have the same performance through all levels. In all cases, there is an almost linear decrease in performance going from levels 1-8, with an exception between levels 2-3 where performance barely changes. The 3D CAD and video streaming cases seem to be more difficult to encode as they have much slower performance than the others. For level two, the word processing case is encoded seven times more quickly than the video streaming case.

(a) All compression levels.
(b) Level zero excluded.

Figure 6.7: Encoded MPx/s for the Tight encoder.

6.3.2 ZRLE encoder

Figure 6.8 shows the results for the ZRLE encoder and is very similar to the Tight encoder results. Where they differ is that there is no interruption in performance decrease between levels 2-3 and that the performance for the photo editing case is slightly better than the video editing one.
6.3. Test case comparison - MPx/s

6.3.3 Default encoder

Several things differ between the default encoder and the previous two results, as shown in Figure 6.9. First, the gap between the slowest and fastest test case is much smaller, showing more consistent performance. Another difference is how the MPx/s rate for all cases except for 3D CAD and video streaming seems to follow the inverse square law between levels 1-6. From levels 6-8, steep declines can be seen in performance and no changes between levels 8-9.

Figure 6.8: Encoded MPx/s for the ZRLE encoder.

Figure 6.9: Encoded MPx/s for the default encoder.
6.3.4 RRE, Hextile and JPEG encoders

As the RRE, Hextile and JPEG encoders are unaffected by compression level, the results are aggregated into a single graph, seen in Figure 6.10. One apparent thing is that the JPEG encoder performs very consistently regardless of input data. This is in contrast to RRE and Hextile, which show huge disparities between the test cases.

![Figure 6.10: Encoded MPx/s for the RRE, Hextile and JPEG encoder.](image)

6.4 Upper bound compression

By using each encoder once every time a rectangle is encoded, the encoder with the highest compression ratio can be picked to calculate the upper bound in compression ratio. This section presents the results of this brute-force approach, first using the default settings which is using a compression level of two for Tight, level six for ZRLE, and a quality level of eight for JPEG, shown in Section 6.4.1. The experiment was run again using a compression level of two instead of six for ZRLE, the results of which can be seen in Section 6.4.2.

6.4.1 Default settings

Table 6.7 shows the results for the brute-force experiment using the default settings and includes, in addition to execution time and compression ratio, the ratio of which each rectangle was encoded by each encoder. We can see that, for most rectangles, Tight and ZRLE are the best in terms of compression ratio. One exception is the video streaming case, where JPEG is used most of the time. Interestingly, the Raw encoder is the best in very few cases, probably where small rectangles are encoded and the overhead of the other encoders decreases the compression ratio.
6.4. Upper bound compression

<table>
<thead>
<tr>
<th>Use case</th>
<th>Time (s)</th>
<th>Ratio</th>
<th>Raw</th>
<th>RRE</th>
<th>Hextile</th>
<th>Tight</th>
<th>JPEG</th>
<th>ZRLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word processing</td>
<td>58.09</td>
<td>38.02</td>
<td>0.16</td>
<td>0.34</td>
<td>38.41</td>
<td>43.05</td>
<td>1.68</td>
<td>45.82</td>
</tr>
<tr>
<td>3D CAD</td>
<td>60.52</td>
<td>17.83</td>
<td>0.00</td>
<td>0.06</td>
<td>8.71</td>
<td>32.69</td>
<td>31.59</td>
<td>26.95</td>
</tr>
<tr>
<td>Photo editing</td>
<td>2.14</td>
<td>28.91</td>
<td>0.01</td>
<td>0.15</td>
<td>2.54</td>
<td>32.17</td>
<td>16.54</td>
<td>48.59</td>
</tr>
<tr>
<td>Video editing</td>
<td>103.58</td>
<td>33.06</td>
<td>0.65</td>
<td>0.56</td>
<td>6.53</td>
<td>30.20</td>
<td>16.75</td>
<td>45.31</td>
</tr>
<tr>
<td>Video streaming</td>
<td>180.70</td>
<td>15.86</td>
<td>0.02</td>
<td>0.33</td>
<td>2.89</td>
<td>24.42</td>
<td>59.30</td>
<td>13.04</td>
</tr>
<tr>
<td>Web browsing</td>
<td>85.05</td>
<td>24.86</td>
<td>0.06</td>
<td>0.20</td>
<td>8.51</td>
<td>46.09</td>
<td>5.50</td>
<td>39.64</td>
</tr>
</tbody>
</table>

Table 6.7: Brute force results using the default settings, showing execution time, compression ratio, and how many of the rectangles were encoded by each encoder.

A comparison can be seen between the upper bound brute-force results and the default encoder in Table 6.8. On average, an increase of 12% in compression ratio can be seen across all test cases. This comes at an average increase of 2.2 times in execution time. The execution time for the video streaming case decreased by 5% while the compression ratio remained practically unchanged.

<table>
<thead>
<tr>
<th>Use case</th>
<th>Time (vs. default)</th>
<th>Ratio (vs. default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word processing</td>
<td>3.95</td>
<td>1.21</td>
</tr>
<tr>
<td>3D CAD</td>
<td>1.40</td>
<td>1.03</td>
</tr>
<tr>
<td>Photo editing</td>
<td>1.46</td>
<td>1.10</td>
</tr>
<tr>
<td>Video editing</td>
<td>2.07</td>
<td>1.17</td>
</tr>
<tr>
<td>Video streaming</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td>Web browsing</td>
<td>3.55</td>
<td>1.20</td>
</tr>
</tbody>
</table>

Table 6.8: Brute force results using default settings compared against the default encoder.

6.4.2 Compression level two

Using a compression level of two for ZRLE, we can see lower execution times across the board without a large decrease in compression ratio. The word processing case suffers the most in terms of compression, being 6% worse, but at the same time, the execution time is decreased by 60%. Except for the results for the 3D CAD case, and the video streaming case which hardly changed, execution times are roughly halved.

<table>
<thead>
<tr>
<th>Use case</th>
<th>Time (s)</th>
<th>Ratio</th>
<th>Raw</th>
<th>RRE</th>
<th>Hextile</th>
<th>Tight</th>
<th>JPEG</th>
<th>ZRLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word processing</td>
<td>23.05</td>
<td>35.90</td>
<td>0.16</td>
<td>0.53</td>
<td>10.31</td>
<td>57.54</td>
<td>3.25</td>
<td>28.20</td>
</tr>
<tr>
<td>3D CAD</td>
<td>44.29</td>
<td>17.76</td>
<td>0.00</td>
<td>0.06</td>
<td>9.59</td>
<td>38.03</td>
<td>32.27</td>
<td>20.05</td>
</tr>
<tr>
<td>Photo editing</td>
<td>1.33</td>
<td>28.61</td>
<td>0.01</td>
<td>0.17</td>
<td>2.73</td>
<td>40.34</td>
<td>16.83</td>
<td>39.92</td>
</tr>
<tr>
<td>Video editing</td>
<td>55.56</td>
<td>32.09</td>
<td>0.59</td>
<td>0.93</td>
<td>7.95</td>
<td>41.55</td>
<td>18.09</td>
<td>30.90</td>
</tr>
<tr>
<td>Video streaming</td>
<td>179.96</td>
<td>15.85</td>
<td>0.02</td>
<td>0.44</td>
<td>3.79</td>
<td>26.10</td>
<td>59.46</td>
<td>10.19</td>
</tr>
<tr>
<td>Web browsing</td>
<td>41.16</td>
<td>24.25</td>
<td>0.06</td>
<td>0.26</td>
<td>9.50</td>
<td>59.69</td>
<td>6.15</td>
<td>24.35</td>
</tr>
</tbody>
</table>

Table 6.9: Brute force results using compression level two for Tight and ZRLE, showing execution time, compression ratio, and how many of the rectangles were encoded by each encoder.
Compared to the default encoder, an average increase of 10\% in compression ratio can be seen, while the average execution time increases by 21\%. The execution time increases not only in the video streaming case, as when using the default settings, but also in the photo editing case which sees a speedup of 10\%, and an increase in compression of 9\%. A summary of the results can be seen in Table 6.10.

<table>
<thead>
<tr>
<th>Use case</th>
<th>Time (vs. default)</th>
<th>Ratio (vs. default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word processing</td>
<td>1.57</td>
<td>1.14</td>
</tr>
<tr>
<td>3D CAD</td>
<td>1.02</td>
<td>1.03</td>
</tr>
<tr>
<td>Photo editing</td>
<td>0.90</td>
<td>1.09</td>
</tr>
<tr>
<td>Video editing</td>
<td>1.11</td>
<td>1.14</td>
</tr>
<tr>
<td>Video streaming</td>
<td>0.94</td>
<td>1.00</td>
</tr>
<tr>
<td>Web browsing</td>
<td>1.72</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 6.10: Brute force results using compression level two for Tight and ZRLE compared against the default encoder.
This chapter discusses the various parts of the thesis and is split into three sections. Section 7.1 discusses the results of the experiments conducted in this thesis. Following, Section 7.2 discusses the methodology and design of the testing framework and experiments. Lastly, Section 7.3 discusses the thesis in a wider context.

7.1 Results

This section provides a discussion of the results of the experiments conducted in this thesis, which includes a comparison between the encoders, a comparison between each test case, as well as a discussion of the brute-force experiment.

7.1.1 Encoder comparison

The results show that the default settings in TigerVNC client, in almost all cases, have the lowest execution time and highest compression ratio beating all other encoders. The Tight and ZRLE encoders only managed to beat the default slightly in compression ratio for the word processing test case, while being twice as slow. As usual when it comes to data compression, one has the difficult task of weighing execution time against compression ratio. The only encoder that isn’t available to choose in TigerVNC client is RRE and with good reason. It consistently performed the worst in compression, at times being even worse than Raw encoding.

It is interesting to see how large the improvements are by using both Tight and JPEG, instead of using just one of them, both in terms of execution time and compression ratio. The largest improvements are seen in the video editing case, shown in Figure 6.4 where the default beats JPEG by 53% and Tight by 117% in compression ratio at level two.

7.1.2 Test case comparison

In Section 6.3, we can see a comparison between the test cases for all encoders. Generally, the encoders’ performance are highly dependent on the input data. We can see similarities between the video editing and photo editing cases, seen in Figure 6.3 and Figure 6.4 respectively. The results are almost identical, suggesting that the recorded sessions share similar
characteristics, one being that both have static, mostly solid, user interfaces with smaller regions with high colour density.

The results for the 3D CAD case and video streaming are also similar, albeit to a slightly lesser extent, as shown in Figure 6.2 and Figure 6.5. There is a larger gap between the default and JPEG to the other encoders compared to the other cases. Both cases are similar in that there are constant changes in them, giving frequent framebuffer updates. Furthermore, both also have a higher density in colours, although the 3D CAD case has a lot of static UI elements with large solid chunks.

Lastly, we can see some similarities between the word processing and web browsing test cases, seen in Figure 6.1 and Figure 6.6. Tight and ZRLE perform well in compression in both cases. The web browsing case consisted of browsing random articles on Wikipedia, consisting mostly of a solid white background and text. This is very similar to the word processing case, which also mostly consisted of a solid background with text.

7.1.3 Brute-force experiment

The results in Section 6.4 show that there is potential for both increased compression ratios and lower execution times in TigerVNC. While a scenario like this was only shown once for the photo editing case in Table 6.10, we still saw significant increases in compression ratio in all cases except for video streaming.

Compression is generally a tradeoff between execution time and compression ratio, where higher compression ratios tend to lead to exponentially higher execution times. Using the default settings, Section 6.4.1 shows that the upper bound for compression is, on average, 12% higher than the current implementation. This comes at a cost of being 2.2 times slower, which might be worth it if one has an excess of computing power. In most cases, this tradeoff is probably not worth it. Considering these results, the current EncodeManager manages to keep a good balance between compression ratio and computing time.

The default encoder uses a combination of the Tight and JPEG encoders, but results presented in Table 6.7 and Table 6.9 show that the other encoders, notably ZRLE and Hextile, do account for a substantial amount of encoded rectangles. To improve the EncodeManager in TigerVNC, more research needs to be done to investigate what makes a rectangle more likely to be compressed better by each encoder.

7.1.4 Lossless encoding

Lossy compression in general achieves higher compression than lossless compression. The JPEG encoder is the only one that can not do lossless encoding, which makes the comparison in compression ratios a bit unfair and worth keeping in mind. However, the question is if this even matters in most cases. The quality level used by default in the JPEG encoder, level eight, is said to be perceptually lossless based on current research. As objective image quality metrics were not studied in this thesis, it is difficult to say how much of the images were degraded. When testing and debugging the framework during development, no visual difference between the encoders could be seen. This subjective observation suggests that a quality level of eight is good enough for most users. An exception could be where perfect image replication is extremely important, such as in medical applications. In those cases, using a strictly lossless encoder, at the cost of performance, is a better choice.

7.2 Method

This section discusses the methodology used for the design of the recording tool, as well as the experimentation method used. Alternative methods are brought up, together with the limitations of the current implementation.
7.2 Method

7.2.1 Recorder tool

The problem of needing to record an X session is not a new phenomenon and there are many available solutions already. For example, most screen recording software can be used to capture an X session, such as the open source software OBS[^1] or x11grab[^2] in FFmpeg. The difference between how typical screen recorders work, and how TigerVNC works, is that screen capturing software capture the screen at a set frame rate and always encodes the entire framebuffer at each interval. This is in contrast to TigerVNC, which only sends an update when a change is detected, and only sends the part of the screen that is changed.

rfbproxy

A way to record an X session, which several other studies have used for their performance comparisons [17, 29, 30, 39], is rfbproxy[^3]. The program acts as a proxy between a VNC client and server and stores the RFB protocol messages sent from the server to a file, which can later be parsed to decode framebuffer updates. The aforementioned studies that used rfbproxy did so by first decoding each rectangle, and then encoding them again using different settings and taking measurements. One flaw with this approach is that it can only be used to compare the encoders on a rectangle-by-rectangle basis, and not based on the system that determines how the rectangle should be split up. This means that changes made to the logic that splits up rectangles are impractical to benchmark. Instead of comparing rectangle encoding performance, framebuffer updates would have to be fully reconstructed using all corresponding rectangles before re-encoding them again.

Use of rfbproxy also requires one to run both a VNC client and server. Running a VNC client and server to record an X session is inefficient, especially when an X server is already available locally. Capturing the X server directly without going through VNC yields more framebuffer updates captured per second, which gives the recorded session file a closer representation of how the session looked like locally when it was recorded.

Update tracking

The recorder tool was built to tackle these weak points, with the goal of being able to record a session and play it back in a way that makes it indistinguishable from the original session. This gives benchmarks strong replicability compared to running benchmarks on a live session. Depending on the hardware and what is being recorded, the recorder can capture a lossless session in a way mimicking RFB framebuffer updates. However, it does not reflect how Xvnc detects changes to the framebuffer. Xvnc does not use X DAMAGE events but instead hooks on draw calls made by the GPU. It also uses the ComparingUpdateTracker[^4] class, which is used to compare the contents of rectangles in the framebuffer, looking at each pixel and detecting areas that are unaffected. For example, if an application reports that the entire window has changed when only a small UI element has been altered, the ComparingUpdateTracker will only mark the UI element as changed, even if the entire window is marked as updated. This is not something that the recorder tool does, which does affect the results presented in this thesis, especially for the 3D CAD and video editing cases. The application used in those cases always reported the entire screen as changed whenever any updates were made to the screen, despite that the changed regions in the application mostly included smaller regions and not the static UI elements. If the recorder tool implemented the ComparingUpdateTracker, the framebuffer updates in the recorded session

[^1]: https://obsproject.com/
[^2]: https://ffmpeg.org/ffmpeg-devices.html#x11grab
[^3]: http://cyberelk.net/tim/software/rfbproxy/
would be much smaller, which would not only affect the execution time results, but could also affect the compression ratio and encoded MPx/s results.

Limitations

One of the encoding types defined in the RFB protocol is the CopyRect encoding, described in Section 2.4, which is used when part of the framebuffer is copied somewhere, such as when moving a window. This encoding is supported by TigerVNC server and client, which reduces execution time and bandwidth usage as data that is already available to the client does not have to be re-sent from the server. Unfortunately, due to time constraints, a CopyRect-like feature is not supported by the recorder. This could have affected the compression ratio results negatively.

7.2.2 Benchmarking tool

The benchmarking tool is highly configurable and can be set up to use any encoders and settings available in TigerVNC. In the benchmarks conducted in this thesis, all available encoders were used with their default settings, along with the default setting in TigerVNC client which combines Tight and JPEG encoding. While this allows for a good comparison between the encoders’ performance, it does not reflect how the system works in the real world. TigerVNC server uses a heuristic that attempts to respect a client’s desired encoding, but it can deviate in certain circumstances (if the client also supports the encoding). For example, if a client supports the LastRect pseudo-encoding and Tight encoding, the Tight encoder will always be used to encode solid rectangles, regardless of what encoding the client prefers. This is not something that is reflected in the experiments, which affects the results as the Tight encoder is by far the most efficient encoder to use for solid rectangles. The JPEG encoder is greatly negatively affected by this as well, as encoding solid rectangles with JPEG first and then the rest of the region is in almost all cases less efficient than just encoding the entire region.

Reliability

The benchmarking tool is designed to process and encode a session file as quickly as possible, regardless of the timing of the frame updates in the session file. This puts a workload on the machine running the benchmarks that is not necessarily representative of real-world usage. The processor used was an Intel(R) Core(TM) i7-9700 CPU that supports Intel Turbo Boost Technology, which adjusts the CPU frequency based on how heavy the workload is. When the benchmarks are run, an image from the session file is loaded and decoded between each framebuffer update. Loading the image from disk is an I/O bound operation that does not load the CPU greatly, which can cause the CPU to lower its frequency between updates. A study by Mazouz et al. looked at the frequency transition latency in Intel processors and found that the time it takes to switch between frequencies was in the order of microseconds. One might think that considering a single framebuffer update is in the order of milliseconds, this should not affect performance much. But, while the transition is done quickly, it is still up to the operating system to decide when a transition should be done. How fast these transitions occur is not something that was investigated and they could have affected the results. Preferably, the benchmarks should have been run multiple times to give more measurements. This is something that decreases the reliability of this thesis.

Real-world representation

The goal of the recorded test cases was to capture a wide variety of real-world use cases that have different enough characteristics to show how varied inputs affect the encoders. To evaluate how representative they are is difficult and an attempt was made in the literature study of this thesis. Similar studies that used different test cases meant to represent real-world usage, some of which referenced in Section 5.1, did not put much effort into motivating why their choices were made. Maybe anything a user does in a VNC session could be regarded as being real-world usage, but this classification is not very helpful. Instead, it might be better to categorize the characteristics of the application, such as the colour count, colour concentration, update frequency and update size. Finding several scenarios that fit these categories might yield more distinct results than just using the scenario itself as a category. Regarding the test cases used in this thesis, the results do show a large disparity both between the encoders and between test cases. This fact might be more important than evaluating how representative they are of real-world usage.

Perhaps even more important is the simplicity of using the testing framework. Recording new session files is very simple, and anyone can use the tools to create their own test bench which matches their day-to-day usage. With a wide enough test bench, tweaks to the EncodeManager can be evaluated broadly, which can lead to changes that give general improvements in most cases.

7.2.3 Source criticism

The literature used in this thesis is varied, including using peer-reviewed articles from well-known journals and conferences, technical specifications, published textbooks, and online resources. A higher priority was put on using peer-review articles whenever possible, deferring to technical specifications and published textbooks for well-established technical details. Some of the sources include online resources, such as GitHub in the case of the RFB protocol specification, and TurboVNC.com for the studies conducted by The VirtualGL Project.

7.3 The work in a wider context

TigerVNC is the most popular VNC server on GitHub and comes packaged with popular Linux distributions such as Red Hat Enterprise Linux, Fedora, Arch Linux, Debian, Ubuntu, and many more. It has widespread use in both individual and enterprise settings. The open-source nature of the work done in this thesis does not only have the potential to benefit TigerVNC, but other projects as well. The work is not only beneficial to remote desktop applications but also to compression techniques in general. Data compression is a well-studied area and is something that not only benefits users directly, but indirectly it can affect society as a whole. Better compression means less hardware can be used, which means investments can be made elsewhere and a lesser environmental impact.
The goal of this master’s thesis was to analyze the performance of the system in TigerVNC server that is responsible for choosing the appropriate encoding for each framebuffer update. A testing framework was developed on which several experiments were run, giving insights into the performance of the system.

8.1 Research questions

How can a testing framework be developed for TigerVNC that measures the performance in real-world scenarios?

A testing framework was developed for TigerVNC which can be used to capture an X session losslessly to later be played back on the server. From the server’s perspective, playing this session file is indistinguishable from a real X session. Provided that a recorded session is representative of a real-world scenario, the framework can be used to gather performance data that reflect what the performance would be in a real-world scenario.

How effective is TigerVNC at choosing the appropriate compression algorithm?

The experiments conducted in this thesis looked at all available encoders in TigerVNC and compared them against the heuristic in TigerVNC server that is responsible for choosing the appropriate compression algorithm in each framebuffer update. Using the default setting as a baseline, the results show that TigerVNC is very effective at choosing the best compression algorithm, both in terms of execution time and compression ratio. There was one exception where the default fell behind in compression ratio, but managed to have a lower execution time. This was corroborated by a brute-force experiment which calculated what the highest compression ratio is achievable by TigerVNC. The experiment showed that TigerVNC, using default settings, has a good balance between compression ratio and execution time.

What is the effect on performance, measured in execution time and compression ratio, when varied input data is used?

The results show that varied input data greatly affects performance. In several cases, differences in compression ratio varied by a factor of ten, and execution time by a factor of five.
The variance could be explained by the different nature of the test cases, such as image composition and application usage. We observed similarities in performance between test cases that might be related to similarities in the characteristics of the test cases.

8.2 Future work

The framework developed for this thesis lays the groundwork on which future studies can be made. In addition to the measurements taken on the server side, the framework is built so that a real client could connect to it, which would make it possible to perform latency measurements, as many of the studies presented in Chapter 3. If one was to implement new encoders, tweak existing ones, or tweak the EncodeManager, the framework together with a premade test bed could be used as a baseline to evaluate changes in performance.

Due to time and resource constraints in this thesis, only limited test scenarios were analyzed, and only parts of the total data gathered were analyzed. Furthermore, the recorder tool could have been implemented using the ComparingUpdateTracker, described in Section 7.2, which would make the recorder session files match the exact way Xvnc works in TigerVNC. Other things, like investigating how artificial constraints such as a bandwidth limitation would have affected the performance, or experimenting with different framerates instead of using 60, would also have been interesting.
Bibliography


[28] “Performance evaluation of VDI environment”. In: 2017. DOI: 10.1109/INTECH.2016.7845102


