Christopher Wheelhouse

Network Packet Trace Visualisation at Scale

Computer Science Tripos, Part II
St. John's College

June 8, 2014

Proforma

Name:	Christopher Wheelhouse
College:	St John's College
Project Title:	Network Packet Trace Visualisation at Scale
Examination:	Computer Science Tripos, Part II, June 2014
Word Count:	10070
Project Originator:	Malte Schwarzkopf and Matthew P. Grosvenor
Supervisor:	Malte Schwarzkopf and Matthew P. Grosvenor

Original Aims of the Project

The aim of this project was to implement a tool that could visualise network packet traces that were of the order of 100 million packets. This includes the plotting of the packet timelines of multiple traces and of latency graphs between different traces, with plotting completed at interactive timescales.

Work Completed

Overall the project was a success, with all success criteria being met and several optional extensions also implemented. The ability to decode the custom format of trace files was implemented. From this the two main graphs, the packet timeline and the latency graph, were plotted. The plotting of both graphs were to be completed with sub second delays. Additionally options to change of the view of the graphs were added. Overall the current implementation provides a significant increase in speed over other solutions.

Special Difficulties

None.

Declaration

Contents

Chapter 1

Introduction

In this dissertation I set out to create a network packet trace visualiser for a custom format trace file, with a size of the order of 100 million packets. The visualiser should be able to output two graphs: a timeline of a network packet trace and a latency graph between two traces. Each graph should be drawn within an interactive timescale, less than a second, and additional statics should be calculated to help analyse the traces. This has been achieved and thus the aims set out in the project proposal have been met.

1.1 Motivation

Data centres have become ubiquitous. As large volumes of data have become more readily available, data centres are to needed to analyse and process this data in an efficient manner. One key concern in a modern data centre is the need to transfer data from one node to another. As a data centre inherently relies on distributed applications, they often generate significant amounts of network traffic between nodes, especially as it is often quicker to retrieve data from another node's main memory rather than accessing one's own hard drive.[?] But it is often difficult to provide dependable network latencies as network contention at switches often creates latency spikes, and due to the distributed nature an increase in latency from one machine could send a ripple effect through the network. [?]

The Computer Laboratory's Systems Research Group is currently developing a bufferless network architecture, R2D2. R2D2 aims to create a resilient, realtime network through reducing latency. This is done by moving buffers to the end point of the network as well as sacrificing bandwidth. Hence any packets are able to traverse the network quickly without being delayed at a switch's queues.[?]

The research into R2D2 requires that network packet traces can be analysed. Part of this analysis looks into how packets behave in a network, especially the characteristics of sequences of packets, rather than individual packets. "Packet trains"[?] are a series of packets in quick succession. The researchers are interested in the traffic pattens that are created by different applications and whether packet trains are created. Packet trains can cause latency spikes for an application if two different packet trains intersect at a switch.

1.2 Challenges

The trace files are provided as a custom tightly packed binary file. Each file is captured at line rate from a 10Gbit/s link. Thus each trace potentially contains of the order of 100 million packets at approximately several gigabytes in size. For each packet in the custom trace file 20 bytes are required to store the required meta-data of the trace. This means that a 100 million packet trace will occupy approximately 1.9GB. The size of each trace is a major complication, both in terms of the number of packets and memory requirements of each trace. Thus the data structures and algorithms used to plot the trace must be taken into special consideration to gain a level of efficiency that is acceptable in both time and space complexity.

Big $\mathcal{O}$ complexities become obvious if we consider the problem: if we take a binary trace file of 100 million packets and convert it into a Java in-memory representation that can easily be manipulated there will be a large memory overhead compared to the original file, possibly a doubling in size. Thus if we take two traces, if we are comparing them against each other, then they can potentially use 8GB of memory just to get them into a usable format. Similar overheads are also present when dealing with the number of packets in the trace. If we have to calculate the position in a graph for each point in a 100 million packet trace file and then plot that position, the cost would be too large.

1.3 Related Work

Many mature graphing tools exist but none are particularly well suited to plotting graphs on this scale. Tools like matplotlib and gnuplot end up plotting each point in the data set. Thus for a 100 million point data set they would plot a graph at a granularity that could never be seen. This problem would also be compounded if we wished to change the zoom level of the graph, all the points at the new zoom level would have to be replotted, hence making the tools unsuitable for interactively analysing a trace file.

Chapter 2

Preparation

In this chapter I give an overview of the relevant parts of the R2D2 project that were needed to implement the visualiser, along with how packet traces were captured. I also briefly look into how other plotting tools currently currently work. Then I give the requirements needed for the project. Finally I discuss the tools chosen for the implementation and software engineering techniques that were used over the course of the project.

2.1 R2D2

The R2D2, Resilient Realtime Data Distributor, project is a bufferless and switchless network designed with data centres in mind. It aims to provide hard latency guarantees, using commodity hardware.[**?**]

Typically packets are sent into the network from different sources, as soon as they hit a switch the packets get put onto incoming buffer queues. This would not normally be a problem however if the network is congested and an incoming packet arrives at a full buffer, the packet may be dropped or queued until the buffer from that point has been emptied, potentially taking many RTTs. Hence the latency for that packet, and similar packets will increase.

Another problem with buffers happens when many different streams of packets combine at a switch. Take two different sources of packets that are sending data to the same server. When they intersect at a switch one stream will be blocked waiting for the others to traverse the switch. At scale this contention creates latency spikes.

R2D2 aims to solve this problem by moving the buffers out of the network. With the buffers pushed to the end points a software network scheduler is used to co-ordinate access to the network. The switches are also replaced by passive crossover points, in a 1 to N connectivity fashion, with the packets being broadcast going from 1 to N or the packets are merged in the N to 1 direction. A key point is that this can be implemented on traditional hardware.

For the analysis of the network, packet traces are captured at different tapping points in the network for comparison. Figure 2.1 shows how two traces are captured. Servers 1 and 2 are both sending packet streams to server 3. These streams intersect at the network switch and get forwarded on to server 3. Capture point 1 records the trace of packets before it has been intersected with the traffic from server 2. That trace then serves as a baseline for the trace from the second capture point, as the two traces can be compared to see how transitioning through the switch has affected network behaviour, e.g. time to traverse the switch, packet ordering.

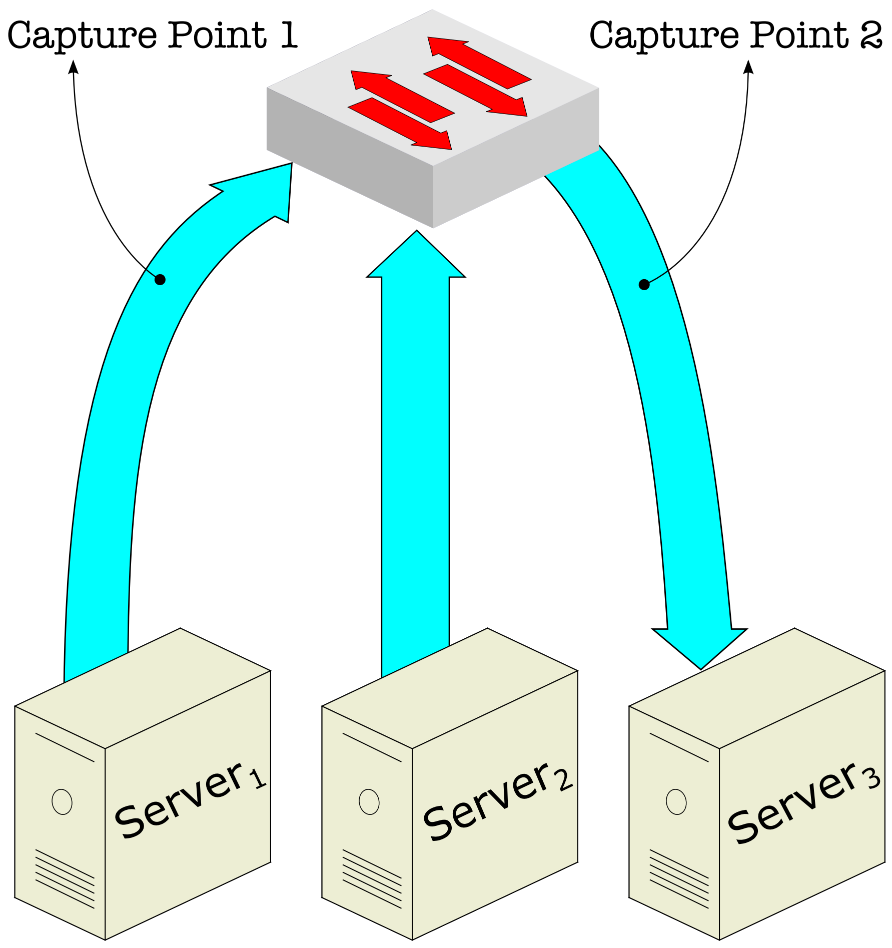


Figure 2.1: The packet capture setup: Servers 1 and 2 are sending packets to server 3. While the Capture Point 1 and Capture Point 2 are recording the trace.

2.2 Existing Plotting Tools

Before working on the visualiser I investigated the leading existing plotting tools: gnuplot[1], matplotlib[2] and MATLAB[3] to see I could use any of them to fulfil any of the requirements of this project.

[1]http://www.gnuplot.info/
[2]http://matplotlib.org/
[3]http://www.mathworks.co.uk/products/matlab/

While each of the tools are well established and are well supported they couldn't provide the performance that was required. None of the tools explicitly supported the plotting of one dimensional data, i.e. the packet timeline. As evaluated in Section 4.4 the tools plot the required graphs far too slowly and do not provide any easy ways to improve the performance of the plotting times over the initial performance of their standard libraries. MATLAB and matplotlib, through python, have the ability for some advance general purpose computing that may of helped in plotting a trace file but the bottleneck encountered when plotting was far too great. Thus none of three tools were used in the implementation of the this project.

2.3 Requirements Analysis

The main objectives of my project are as follows:

1. Decode the network packet trace file into a form that a Java program can easily manipulate.
2. Implement the latency graph and packet timeline and ensure that any replot of the two graphs happen with less than sub second delays such that the graphs can be visually analysed at interactive time scales.
3. Build a user interface to allow a user to more easily interact with the program.
4. Calculate additional statistics from the trace that could help with the analysis.

2.4 Choice of Tools

2.4.1 Programming Language

There was no specific library or programming language function that was needed to complete the project, thus there was a large choice of languages that could be used for the implementation. The main language that was used to write the visualiser was Java, specifically Oracle JDK7 SE[4]. No features outside of the standard library were used. Java was chosen for a number of reasons:

- Due to the requirements of the project is was prudent to use a high level, object oriented language and where most of the common functions needed would be part of the standard library.
- I had experience programming with Java. Thus while languages like C and C++ may have provided a small amount of improved performance, for example with I/O operations, the lack of familiarity would of hindered the project. Choosing Java meant I could focus on good software engineering practices.
- Java had a mature and stable UI toolkit. It included support for custom graphics, Java 2D, and also general GUI components, Swing. The toolkit also provided

[4]`http://docs.oracle.com/javase/7/docs/index.html`

support for advance features including supporting concurrency for some of the longer running processes that the visualiser would have to run.

2.4.2 Development Environment

Development was completed on OS X Mavericks and all code was written using Sublime Text 2. Two separate version control repositories were used for this project. The first was a private git repository that is hosted on GitHub[5]. All code written for this project was pushed to this repository. The second was a public git repository that was used for all LaTeX files relating to this dissertation. Backups were done on a weekly basis using Dropbox[6]. Although Dropbox is a commercial service backups were also done to my personal external hard drive, in case their servers went down.

2.5 Software Engineering Techniques

Due to the uncertain way in which to achieve the requirements listed in Section 2.3 I chose to develop this project in an iterative and incremental way using the spiral model. Each iteration usually starting with a meeting with my project supervisors where we would plan out the next incremental stage of the project, and resulted in adding, refining or modifying a feature of the program. After a feature was added or modified the program was unit tested to make sure that the feature worked correctly with the rest of the program and functioned as expected. The code was then committed to my repository.

Throughout the implementation of the project I aimed to follow good software design principles including a high degree of cohesion, low coupling and logical handling of errors. I was able to use the language features that Java provided to help with this such as modifiers and the class based structure to provide encapsulation and Java's powerful exception handling constructs to catch errors.

2.6 Summary

In this chapter I have described the necessary work that was needed to be done before I implemented the visualiser. A summary of R2D2 and the potential tools that could be used to visualise the impact of R2D2 was given. Finally the tools chosen for the project were described along with the development methodology.

[5]`https://github.com`
[6]`https://www.dropbox.com/`

Chapter 3

Implementation

This chapter describes the implementation of the network packet trace visualiser. There are three components to the program:

- *Trace Decoding*

  This does the initial work in transforming the tightly packed binary packet trace file into a form that the rest of the program can manipulate easily. This is needed otherwise the components below would have to manipulate the binary file whenever they need to plot the graphs.

- *Packet Timeline*

  The second component is responsible for plotting the main visualisation graph. It includes some pre–processing of the traces, before they are plotted, to calculate what part of the traces to plot.

- *Latency Graph*

  The third component plots the latency between common packets in two separate traces. Like the packet timeline this also needs to do some pre–processing before the graph can be plotted. From two traces, the common packets between each trace have to be found and then their latency's can be computed.

This chapter describes the implementation of all three components and also several optimisations that were also explored to increase the efficiency of the components. Figure 3.1 shows a high level overview of the visualiser, showing the steps taken to get to the output graphs.

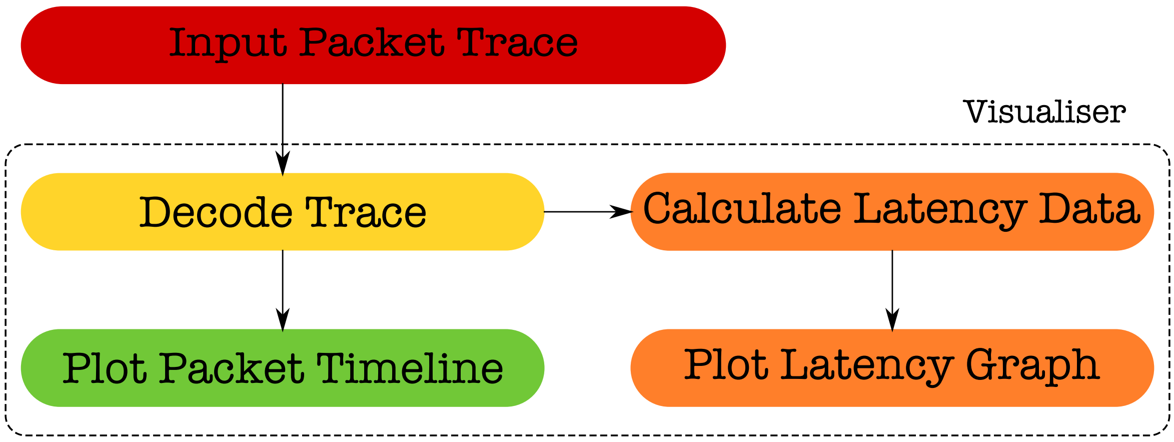


Figure 3.1: An overview of the processing completed in the visualisation tool.

3.1 Trace Decoding

The first step in displaying any visual information was to decode the packet trace file into a Java readable form, as the project relied on this it was critical that this step was implemented early in the project. Each trace is a custom, tightly packed binary file containing summary information about the trace and meta-data for each packet.

The start of the file has a header with the format:

```
uint64_t start_time; //Start time of the trace in nanoseconds
uint64_t end_time;   //End time of the trace in nanoseconds
uint64_t samples;    //Number of sample packets in the file
```

Figure 3.2: The first three variable in a trace file.

The remainder of the file is a list of sample packets, each 20 bytes, and with the following format:

```
uint64_t timestamp;   // Timestamp in nanoseconds
uint32_t length :16 bits //Length of the packet
uint32_t dropped:8  bits //Packets dropped behind this one
uint32_t type   :8  bits //Type of the packet
uint64_t hash;   // 64bit unique identifier for each packet
```

Figure 3.3: The format of each packet.

When decoding the first problem encountered was that each value in the trace is stored as an unsigned integer and as of Java SE7[1] there was no support for unsigned integers.

[1]From Java SE8 there is now support for unsigned 32–bit *int* and 64–bit *long* arithmetic but it was released too late to be useful for this project.

Thus for each variable in the trace I had to decide on a primitive or object that could hold the value. I then verified that it was able to handle the data it was given - e.g. it wouldn't overflow, and that its memory footprint was minimized.

The `length` variable has a maximum value of $2^{16}-1$ and can easily be stored as an *int* primitive. The length variable is accessed relatively often, the `dropped` and `type` fields on the other hand are rarely accessed, as a result it is suitable to store them as two *byte* primitives. Even though both variables are unsigned bytes there is a one-to-one mapping between them and the two's complement structure that Java uses for *byte*, allowing at runtime for the correct values to be translated back.

For `timestamp`, `start_time` and `end_time` they are stored as unix time, in nanoseconds. This means that they can be safely stored in a Java *long*[2]. The final variable `hash` is be stored as a Java *long*. Even though the maximum value of *long* is smaller than the unsigned 64 bit integer, if `hash` is greater than $2^{63}-1$ then it will cause the *long* value to overflow into the negative numbers. This is fine as it maintains the uniqueness property that is required from the hash.

For the variables `length`, `dropped` and `type` a mask also has to be applied to force Java to keep the correct values. All three are read and then cast to a suitable data type, Java's casting sign extends the values meaning that any values read off that are negative will be negative when cast to the correct data type. For example if `dropped` is `11111111` then when it is cast to a short it will turn to `1111111111111111` but applying the mask will return it to its correct value `0000000011111111`.

The second problem with the trace is that it is encoded in little endian format while Java I/O is in big endian. Thus the byte order must be reversed for each variable.

Once the file is read each variable is stored in one of five arrays at a common index. Rather than storing each variable encapsulated in a packet object they are stored separately for performance reasons, as discussed in Section 4.3.1.

Arrays are used to store the packets due to the time and memory considerations. An array has an extremely small memory footprint in Java. The program often has a need to have random access to the data, thus ruling out any pointer based data structures. With an array there is a $\mathcal{O}(1)$ cost in accessing and setting the data at a particular index. The constant cost for getting and setting combined with the fact that the data is provided in-order also makes this the fastest data structure for this application.

3.2 Packet Timeline

While the Trace file has been decoded and placed in a memory efficient data structure with fast access times to each element there is still more background work to perform before the packet timeline can be plotted.

As described in Section 2.3 the visualisation has to support one or more traces. When the traces are plotted the time shown should be the intersection of the time interval

[2]The maximum value of a *long*, $2^{63}-1$, is sufficient to hold time in nanoseconds until April 2262, far past the lifetime of this program.

of each trace. Algorithm 3.1 calculates the new intersection for each trace added. The algorithm is given the alignment times of the set of traces that have already been aligned and the position in the data structure described in Section 3.1 of each time for each trace. This alignment information is needed to quickly plot the traces.

Algorithm 3.1: Calculating the new alignment information when a new trace is given. Trivially for base case when the trace given is the only trace: the intersection times are the start time and the end time, while the corresponding positions of the times are the first and last packets.

```
AlignTrace(trace):
    Result: Once complete the start and end times of the intersection and the
            packet locations for the intersection times for each trace are know.
    commonTraces = List of all previously processed traces;
    timeStart, timeEnd = Start and end time of the commonTraces intersection;
    traceStart, traceEnd = Start and end time of the given trace;
    /* If the condition is true then there is no overlap in times
       between this trace and commonTraces, the list of traces.      */
    if $traceStart \geq timeEnd \vee traceEnd \leq timeStart$ then
        Intersection = false;
    else
        /* Calculate the new intersection times                      */
        if $traceStart > timeStart$ then
            timeStart = traceStart;
        end if
        if $traceEnd < timeEnd$ then
            timeEnd = traceEnd;
        end if
        /* Calculate the positions in each trace of the start and end
           intersect times.                                          */
        foreach t in commonTraces do
            t.startIntersectPosition = t.search(timeStart);
            t.endIntersectPosition = t.search(timeEnd);
        end foreach
    end if
```

Once the first and last packet of each trace that is within the common time interval of the traces are found the packet timeline can be plotted. Every relevant packet is drawn on a canvas as shown in part a of Figure 3.4, if there is no intersection between traces each packet in every trace is plotted as in part b. A packet is plotted by drawing a line between the co–ordinates associated with the start and end time of the packet. The end time is just: $startTime + packetLenth * 0.8$. The packet length in bytes is translated into the time domain by first converting to bits and then multiplying by 0.1 (ns per bit), as the line speed is assumed to be 10 gigabits per second. The co–ordinates of each packet can be calculated by scaling the times with regards to the total time to be plotted.

As is expected plotting of the order of 100 million points is a very time consuming operation, approximately 60 seconds for a trace containing 70 million points. Several optimization strategies are explored in Section 3.4.

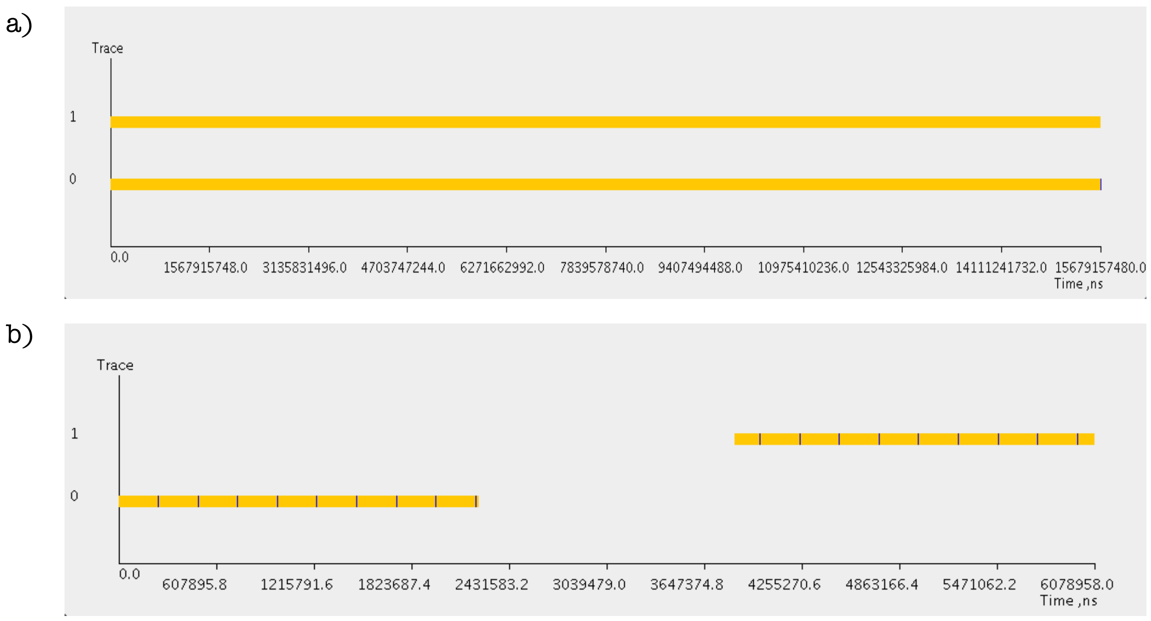


Figure 3.4: Part a shows the default view after each trace has been decoded and all pre–processing has been completed. Trace one starts before trace two and trace two finishes after trace one but only the intersection of the two traces times are plotted. Part b on the other hand shows the same two traces truncated to only include the first 10000 packets each.

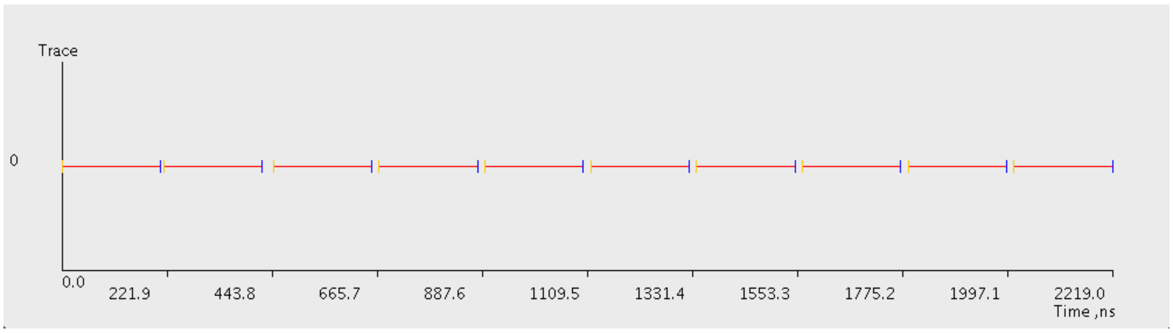


Figure 3.5: The packet timeline on a micro scale only showing the first 10 packets, the time difference can clearly be distinguished this is compared to the macro scale of Figure 3.4.

The packet timeline also requires that there is an ability to zoom and traverse the traces. As an input two times are given to the program, a start and end time – i.e. the time interval that should be shown. These input values and find the corresponding packets number for each time, for each trace, translating from the time domain to the packet domain. If the time end time is before the start of the trace nothing is shown, likewise if the start time is after the trace ends nothing is shown. If the start time is before the start of the trace or the end time is after the end of the trace then the packet numbers default to zero and the trace length respectively. To find the packet numbers in the normal case a binary search is performed. Thus taking $\mathcal{O}(lg(n))$ time to find each packet number, where n is the number of packets. So for a 100 million packet trace each translation will take no more than 27 operations. Once the start and end packets are found the traces are plotted as before.

3.3 Latency Graph

The next graph to be plotted is the Latency Graph. This tracks the time difference between two input traces. It compares the hashes of the packets in one trace to the hashes of the packets in the second trace, if they match then the two packets are the same. The time difference between the two packets is calculated and this is then plotted.

Algorithm 3.2: Calculating the positions of common packets between two traces.

```
FindCommonHashes(traceA, traceB, windowSize):
    Data: Two traces are given as inputs along with the search window size.
    Result: Once complete the positions in both traces of each hash common to
            both traces are returned
    latencyPosition = ∅;
    /* Find the first common hash of each trace.                         */
    positionA, positionB = FindFirstCommonHash(traceA, traceB);
    while positionA < length(traceA) do
        currentHash = Hash of traceA at positionA;
        windowStart = positionB − windowSize/2;
        windowEnd = positionB + windowSize/2;
        Check windowStart and windowEnd are within the bounds of traceB
        /* For each packet inside the window check if the hash of that
           packet matches the hash of the current packet from traceA. If
           there is not a match then it is assumed that the packet does
           not occur in that trace.                                      */
        for i = windowStart to windowEnd do
            hashB = Hash of traceB at i;
            if hashB == currentHash then
                latencyPosition.add(positionA,i);
                positionB = i;
                break;
            end if
        end for
    end while
```

Algorithm 3.3: Find the first packets common to both traces.

```
FindFirstCommonHash(traceA, traceB):
    Data: Two traces are given as inputs.
    Result: Once complete the positions in both traces of the first common packet
            are returned
    commonHash = null;
    positionA, positionB = null;
    setA, setB = $\emptyset$;
    /* Calculate the first common hash and its position in the first
       trace.                                                          */
    for $i = 0$ to $max(length(traceA), length(traceB))$ do
        hashA = Hash of traceA at i;
        hashB = Hash of traceB at i;
        setA.add(hashA);
        setB.add(hashB);
        if setB.contains(hashA) then
            positionA = i;
            commonHash = hashA;
            break;
        else if setA.contains(hashB) then
            positionB = i;
            commonHash = hashB;
            break;
        end if
    end for
    /* Calculate the hash's position in the other trace.               */
    if positionA == null then
        for $i = positionB$ to $length(traceA)$ do
            hashA = Hash of traceA at i;
            if commonHash == hashA then
                positionA = i;
                break;
            end if
        end for
    else if positionB == null then
        for $i = positionA$ to $length(traceB)$ do
            hashB = Hash of traceB at i;
            if commonHash == hashB then
                positionB = i;
                break;
            end if
        end for
    else
        The traces do not intersect.
    end if
    return postionA, positionB;
```

As there is no mapping from one trace to another each packet common to both traces needs to be found. Algorithm 3.2 takes a packet from one trace and compares the hash to each packet in a sliding window, from the other trace, until it finds a match or searches the entire window. If it finds a match then the latency between the two packets is calculated, if it doesn't find a match it assumes that packet is not in the second trace. The sliding window is needed as packets can arrive out of order at a node and also a hash value is not guaranteed to be globally unique but only locally unique. A consequence of the hashes only being locally unique means that other methods to match hashes are excluded, for example the hash values can not be sorted and then compared.

Algorithm 3.2 relies on the location of the first common packet of each trace being given to it, this is calculated by Algorithm 3.3.

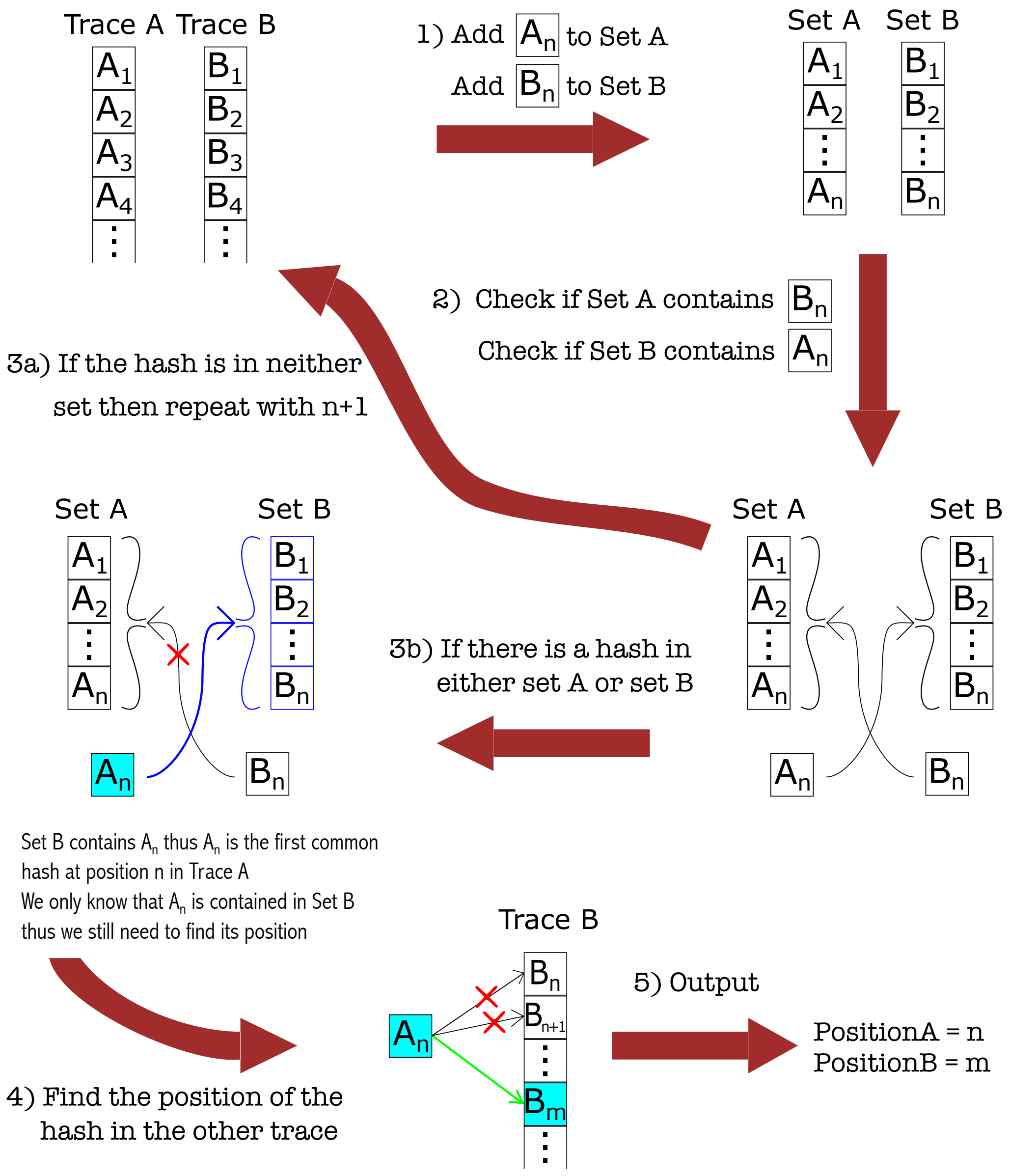

Trace A Trace B
A_1 A_2 A_3 A_4
B_1 B_2 B_3 B_4
1) Add A_n to Set A
Add B_n to Set B
Set A Set B
2) Check if Set A contains B_n
Check if Set B contains A_n
3a) If the hash is in neither set then repeat with n+1
3b) If there is a hash in either set A or set B
Set B contains A_n thus A_n is the first common hash at position n in Trace A
We only know that A_n is contained in Set B thus we still need to find its position
4) Find the position of the hash in the other trace
Trace B
B_n B_{n+1} B_m
5) Output
PositionA = n
PositionB = m

Figure 3.6: A flow graph of the method to find the first common packets of two traces.

Algorithm 3.3 takes the first hash from trace A and adds it to a HashSet for trace A, it then takes the first hash from trace B and adds it to a HashSet for B. Then the hash from A is check against the HashSet for trace B and vice versa the hash from B is check against the HashSet for trace A. If the HashSet B contains the hash from A then the first packet common to both traces is known and the location of that packet for trace A is also known. This is also true in the opposite direction. Thus we can just perform a search until the packet is found in trace B. If neither hash is contained in the opposite HashSet then the next packet is taken and the steps repeated. This is shown in the form of a flow diagram in Figure 3.6

Once the latency for the common packets are calculated each point can be plotted on the latency against time graph, as shown in Figure 3.7.

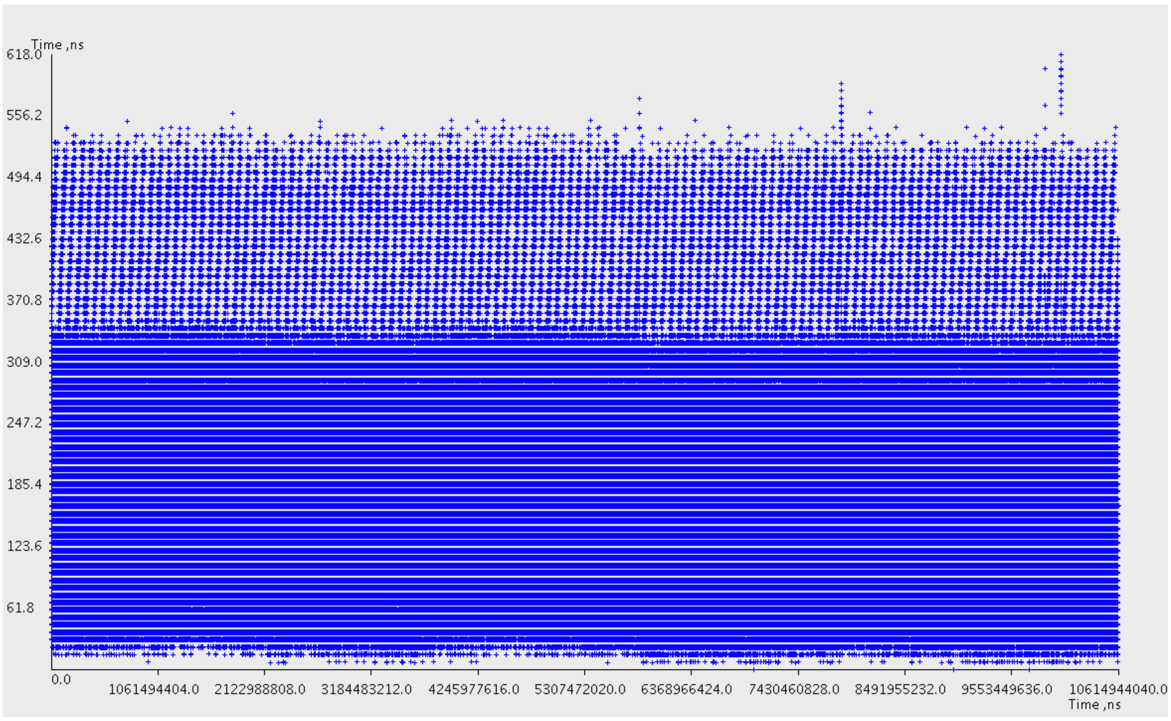


Figure 3.7: The default view of the latency graph.

3.4 Optimisations

The naive packet timeline and latency graph plotting algorithms are very slow. Plotting a single point on the graph is a relativity costly action. Thus plotting every single point is wasteful, especially as there are many more points than pixels so a majority of points will be needlessly redrawn over previous points. This section describes the different optimisations, and refinements, that could be applied to the graph plotting algorithm to speed up the plotting operation and reduce the amount of points plotted.

3.4.1 Constant Sampling

The first technique plots points at constant intervals. It only plots one in every one thousand packets, essentially throwing away or ignoring 99.9% of the total trace. For the packets that are not drawn the logic needed to determine the points' locations and the actual drawing operations are skipped. While this is fine for plotting large uniform traces at a low zoom level, it is obviously not a valid strategy. There are far too many cases where this strategy will fail to show an accurate high level structure of the trace. The accuracy lost far outweighs any plotting speed gain. Although this approach is flawed it does help to give a baseline as to the speed up that can be achieved by not plotting or selectively plotting points on the graph. The speed up that this technique can provide is intrinsically linked to the size of the trace, where as any final solution would ideally not be as dependant on the size of the trace.

3.4.2 Selective Sampling

The previous strategy traded a lot of accuracy for some time performance gain. This next optimization aims to maintain the accuracy of the graph, while also attempting to significantly speed up the plot times. The optimization first calculates the position of the point on the graph and then calculates if a point has already been drawn there, if it has the position for the next packet is calculated, otherwise the point is plotted. Thus skipping the drawing operations for approximately 99.999% of the trace, assuming a 100 million packet trace plotting on 1000 pixels. It should be noted that although the drawing operations may be skipped the location of each packet must still be calculated. With this strategy there needs to some consideration of the time complexity of the logic used to decide if a point should be drawn, if the logic takes longer to execute than the actual drawing operation then there is little to be gain from using this.

To plot a packet on the timeline the start co–ordinate and end co–ordinate are calculated, as integers. There are a number of properties of the Trace's data that allows the simplification of the decision logic:

1. If the current packet's end co–ordinate is the same as the previous packet's end co–ordinate then the packets occupy the same pixel.
2. The start of the packet has a "happens–before" relation with end of the packet.
3. As he trace is ordered by time and also plotted sequentially then we know that the start co–ordinate of the current packet can't be before the start co–ordinate of the previous packet.
4. From points 2 and 3 we know the start co–ordinate of the packet must be between the start co–ordinate of the previous packet and the end co–ordinate of the current packet.

If we fix the previous packet's start and end co–ordinates and the current packet's end co–ordinate then there is only three possible positions in which the current packet's start co–ordinate can be in: before the previous packet's end co–ordinate, in the same pixel as the previous packet's end co–ordinate and after the previous packet's end co–ordinate. This is shown in Figure 3.8

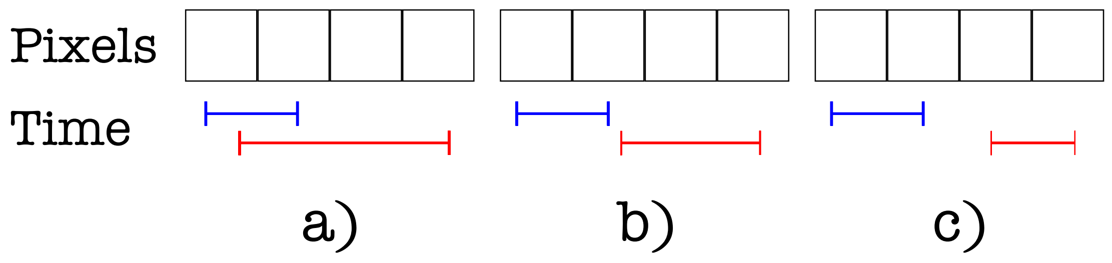


Figure 3.8: The three states the start co–ordinate of a packet can be in when the previous packet's co–ordinates and the current packet's end co–ordinates are fixed. Each state shows the pixel that the packet's times would correspond to, linking the time and pixel domain.

If the first state, a, is true then it is likely that there is an error with the underlying data, in normal operation this state should not happen. For state a then as the start co–ordinate is before the previous packet's end co–ordinate then the start of the packet must of already have been drawn thus whether the rest of the packet is drawn must be dependant on the end co–ordinate. State b is similar to the first state, as the two co-ordinates are the same then the start of the packet must already of been drawn, hence if the rest of the packet is drawn depends on where the current packet's end co–ordinate is. Finally if we are in state c then the whole packet must be drawn, but due the the "happens–before" relationship then the current packet's end co–ordinate must be after the previous packet's end co–ordinate. Thus there is only a reliance on the end co–ordinate of the previous packet and the end co–ordinate of the current packet to see if current packet has to be drawn.

Where previousEnd is the end co–ordinate of the previous packet and currentEnd is the end co–ordinate of the current packet then if the condition below is true then the current packet does not have to be redrawn.

```
previousEnd == currentEnd
```

3.4.3 Bitmap Plotting

The first two optimizations have focused on the packet timeline. They have not been applicable to the latency graph due to its second dimension, there is no function that maps the previously plotted point to the current. On the other hand this optimization is relevant to both graphs.

This optimization creates a bitmap of the graph, one dimension along the x–axis for the packet timeline and a second dimension along the y–axis for the latency graph. When a point is plotted the co–ordinate in the bitmap is set. If the co–ordinate was already set before the point was plotted then the drawing operation is skipped.

This optimization was implement in two separate ways, a faster implementation using *boolean* arrays, or a more memory efficient version using Java's `BitSet`.

3.4.4 Trace Traversal

This final optimization traverses the packet trace trying to find the next point that can be plotted that will visually change the graph. This optimization only works with the packet timeline. Each pixel on the drawing canvas represents a time period For a very zoomed in plot only showing a few packets, a pixel will only be worth a fraction of that packet's time. While a zoomed out plot showing the shape of the graph, a pixel could be worth tens of thousands of packets. This optimization tries to avoid doing any processing on the tens of thousands of packets a pixel represents once that pixel has been plotted.

When a packet is plotted the pixel value after that plotted packet is taken. This is then translated back into a time value, so now we have the start time that the next pixel represents. With this time a search can be preformed on the packet trace to find the position of that time in the trace. The packet corresponding to the position is then plotted. If the position is the same or less than the current plotted packet, for example due to the zoom level, then the next packet after the current packet is plotted. Thus when plotting a lot of packets there is no need to sequentially iterate through the trace.

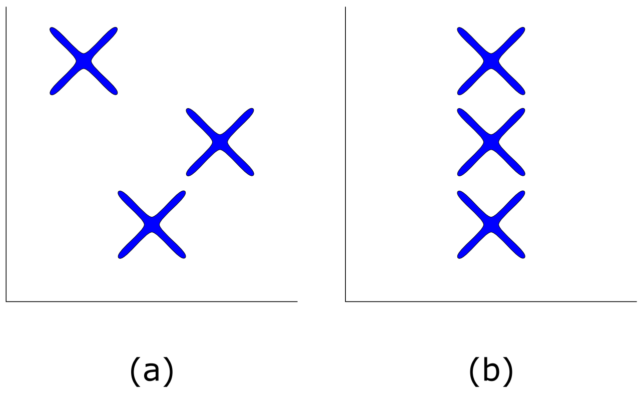
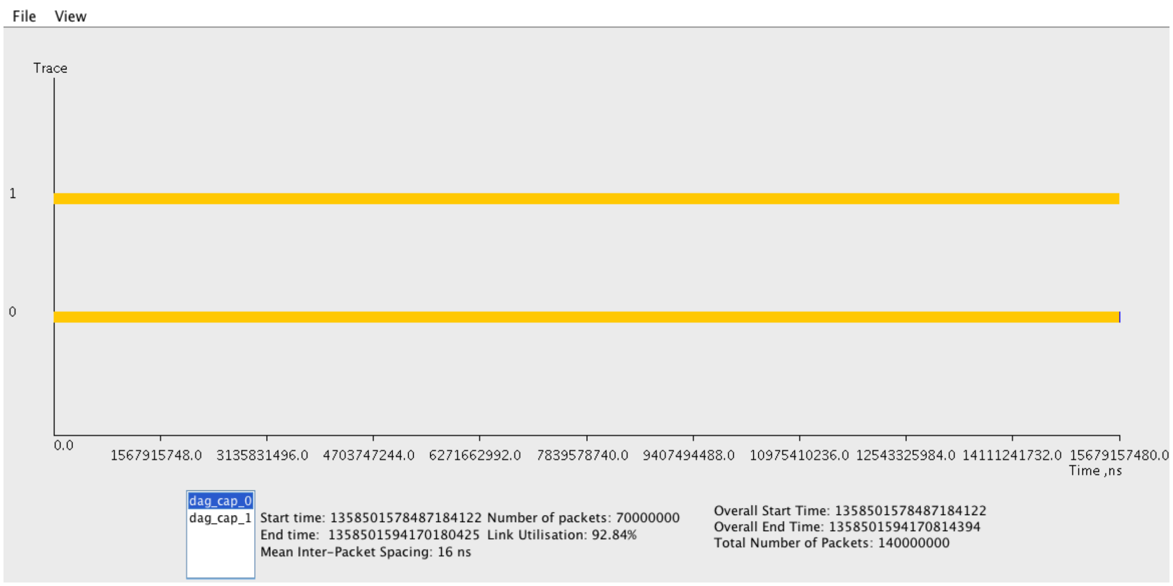


Figure 3.9: The second dimension makes it more difficult to optimise the plotting. Part (a) shows three points at a zoom level. Part (b) shows the same three point at a lower zoom level, the three points now have the same x–axis co–ordinate due to not being able to differentiate the different time values at that scale. Even though in (b) they have the same x–axis co–ordinate the lack of ordering in the y–axis means there is not a fast way to see if a point will be plotted in a different place on the y–axis, other than just searching through each packet in the trace.

This technique will unfortunately not work for the latency graph due to the second dimension, the latency values are not sorted. Thus there is no way to tell if a point plotted with a certain x–axis value will not be plotted again with the same x–axis value but a different y–axis value. This is shown in Figure 3.9

3.5 User Interface

Figure 3.10 shows the normal view of the graphical user interface. It includes some additional information about the packet traces, for example the mean inter-packet spacings and the link utilization percentage which are calculated in the trace decode phase.



Start time: 1358501578487184122 Number of packets: 70000000
End time: 1358501594170180425 Link Utilisation: 92.84%
Mean Inter-Packet Spacing: 16 ns

Overall Start Time: 1358501578487184122
Overall End Time: 1358501594170814394
Total Number of Packets: 140000000

Figure 3.10: The main user interface.

3.6 Summary

This chapter described the implementation of my project and justified the design decisions made. I first explained the method for decoding a packet trace file, looking at how the file was read and the data structures it was stored in. Next the packet timeline was described, this included the pre–processing required to align multiple traces and the zooming functions. I then presented the latency graph and the calculations required to find packets common to each trace. Next several optimisations were described that were applicable to the packet timeline and the latency graph. Chapter 4 demonstrates the effectiveness of each optimisation and the significant speed ups obtained. Finally a brief description of the user interface is given.

Chapter 4

Evaluation

This chapter presents the results of the work completed, as documented in Chapter 3, and compares it to the project's goals. Each method of implementing a feature is evaluated, justifying the design decisions made in the previous chapter.

Throughout this evaluation the all benchmark times are run on an Apple MacBook Air running OS X 10.9, with 8GB of Memory and a 1.7GHz Intel i7 processor. The same trace file has been used consistently through out this chapter, it consists of seventy million packets uniformly distributed over the time frame of the file and has a total size of 1.4GB. When a second trace is needed, for example when plotting the latency graph, a different trace is used, along with the original. The second trace file is again uniformly distributed across its total time and is captured at a different point in the network setup, in Figure 2.1 point A would be the original trace while point B would be the second trace. Thus the second trace has packets common to the original trace file and packets from a new source.

4.1 Overview

The success criteria of the project, as described in the project proposal, were fully satisfied and are outlined below:

1. *The network packet trace file must be decoded.*

   This high priority goal was completed at the start of the project, as the result of the project relied being able to read the trace. The implementation was described in Section 3.1. The correctness of the decoded trace is evaluated in Section 4.2 and the time need to decode the traces are evaluated in Section 4.3.1

2. *A data representation to store the decoded file must be researched and designed.*

   This goal is described in Section 3.1.

3. *Algorithms must be designed such that the packets can be visualised and analysed at interactive time scales, i.e. the trace must be re-drawn with approximately less than sub-second delays.*

The final goal is detailed in Sections 3.2, 3.3 and 3.4. This was the bulk of the project, with the evaluation of the performance undertaken in Section 4.3.

The following optional extensions to the project were also implemented:

1. *Implement a user interface.*
2. *Add support for additional statistics that can help with analysing the trace.*

   Both the user interface and the additional statistics are described in Section 3.5.

4.2 Testing

As this project relied on the correct information being decoded from the trace file unit tests were made for the verification of the decoded data. To achieve this a Bash script was written that verified the the output of the first one hundred samples of the trace against a file of correct values that was provided by my supervisors. This was done for two separate traces.

The correctness of the graphs were also verified. This was completed manually by comparing selected outputs from existing plotting tools to my implementation. I was also able to verify the general shape that the graph should take though my knowledge of the packet trace's characteristics.

4.3 Performance

For the project to be successful the performance of the program had to be better than any other alternative plotting tool that could plot the two graphs. This section evaluates the different sections that were required to plot the to graphs.

4.3.1 Trace Decoding

Decoding the can be separated into two parts: i) the performance of Java I/O in reading the file and; ii) the data structure that the trace is stored in.

File Reading

The efficiency in decoding the packet traces was considered heavily. With packet traces of the order of one hundred million packets, several gigabytes per file, it was essential that the complexity of I/O operations were kept to a minimum and that the optimal buffer sizes were chosen when reading the trace file. For that reason several ways in which Java can read files were implemented, each outputting the correct trace at the end, and compared on the fastest times.

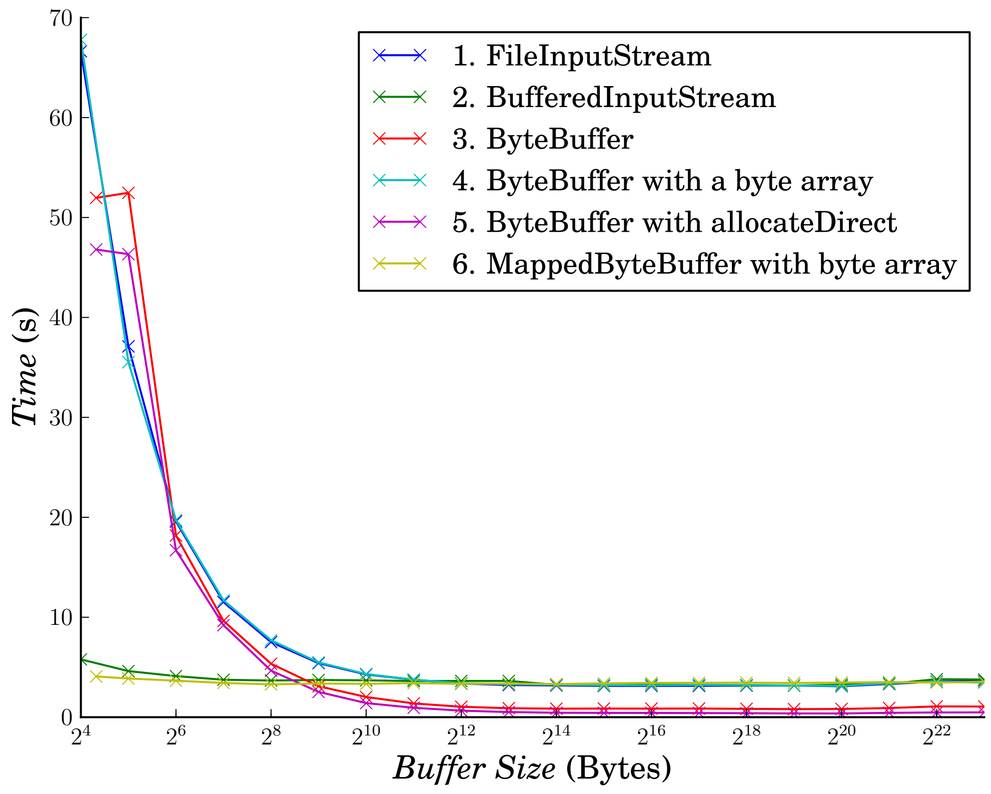


Figure 4.1: Time taken to read a seventy million packet trace file against the size of buffers.

Seven ways of performing I/O in Java were considered:

1. `FileInputSteam` with byte array gets, the array varies in size.
2. `BufferedInputStream` with a variable internal buffer size.
3. `FileChannel` reads which fill a `ByteBuffer` created using the *allocate()* method and then using dedicated get methods to retrieve the values, e.g *getlong()*.
4. `FileChannel` reads which fill a `ByteBuffer` which is backed by a local byte array.
5. `FileChannel` reads which fill a `ByteBuffer` created using `allocateDirect()`. This attempts preform native I/O operations, avoiding copying to intermediate buffers.This uses dedicated get methods.
6. `FileChannel` with `MappedByteBuffer` with byte array gets.
7. `FileChannel` with `MappedByteBuffer`, using dedicated get methods.This is the only option where a buffer size can not be manipulated.

Figure 4.1 shows the times for the first six ways of reading the seventy million packet trace file. The buffer sizes started at 16 bytes, except for item 3, 5 and 6 which due to implementation reasons had a minimum buffer size of 20 bytes. Then timings were measured every doubling of the buffer size. The cliff on the left of the graph is due to the file being read by only a few bytes at a time. For example `FileInputSteam` when only reading one byte a time, not shown in the graph, took on average just under 15 minutes to read the whole file. The seventh I/O operation took 0.6 seconds to complete.

The plot levels out at around 2KB with larger buffer sizes not making too much of a difference. Figure 4.2 shows a clear distinction between using `ByteBuffer` and the other I/O operations. With the top half flattening out at about three seconds, while the two `ByteBuffer` implementations flatten out to about 0.4 seconds to read the whole file. The best implementation was the fifth option with a global minimum of 0.37 seconds at a buffer size of 512KB.

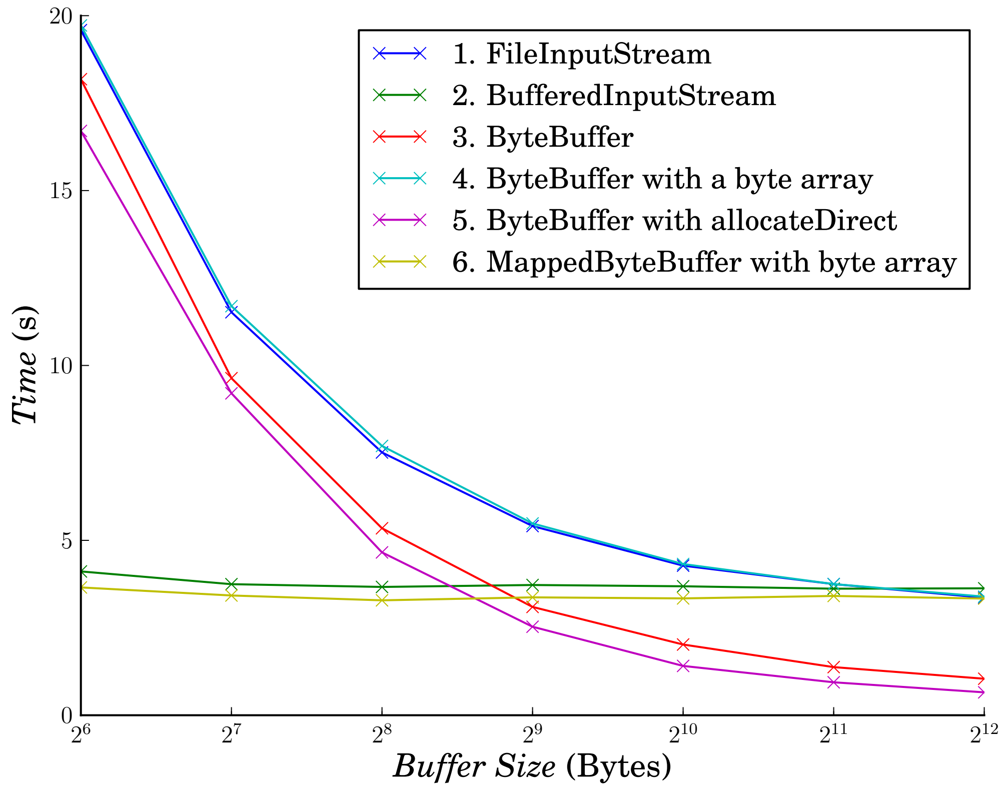


Figure 4.2: Zoomed in view of Figure 4.1.

Using this information a more accurate buffer size could be found for the fifth I/O implementation the direct`ByteBuffer`. This is shown in Figure 4.3. With a minimum at approximately the 65KB mark.

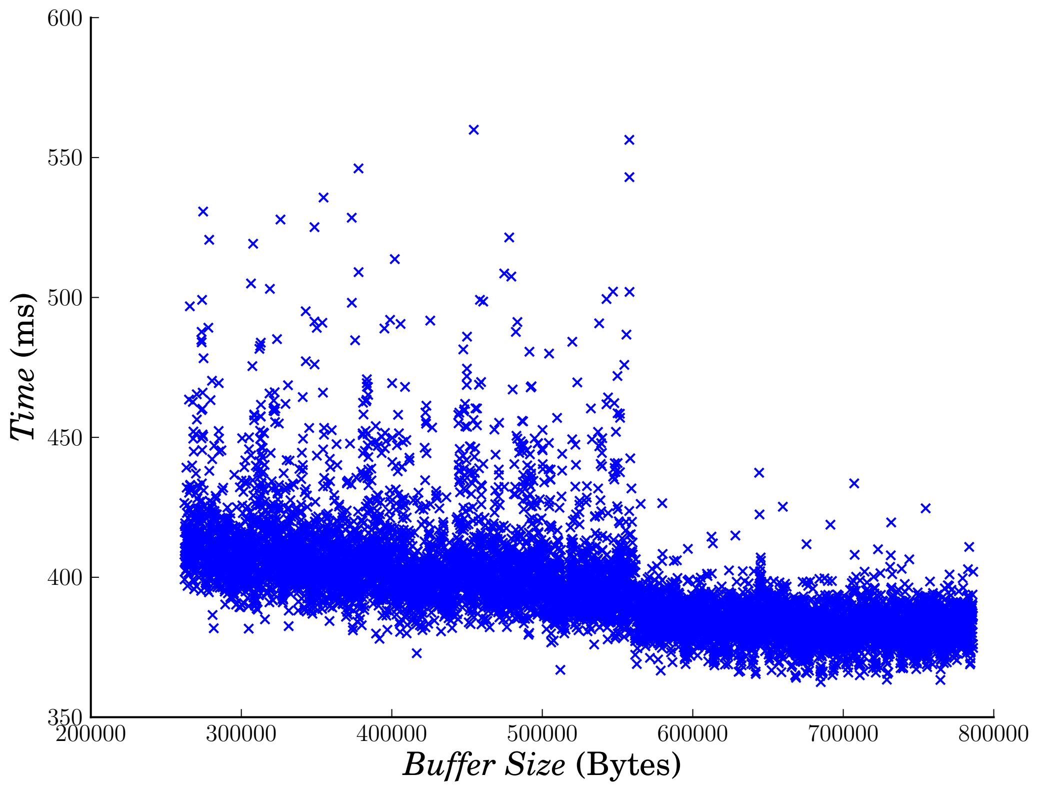


Figure 4.3: A graph showing the decode times of a direct `ByteBuffer` as the buffer size is varied.

Data Structures

While the efficient reading of the trace is important, the consideration of how the trace is stored is potentially more important.

There were two implementations, as mentioned in Section 3.1 for the way the trace was stored in memory.

- Each packet, as described in Figure 3.3, was encapsulated in an object. This encapsulation allowed the each packet to be thought of as a single item, aiding maintainability. Each packet object could then be place in a suitable data structure.
- For each of the five packet variables an array is constructed where each value from the packet is store in its corresponding array.

The first way may initially seem a better way of storing the packets, especially from a software engineering point of view. Although, as demonstrated in Figure 4.4, the time taken to decode a trace file and store it in memory is at most 37 times slower than the second method and on average 24 time slower. To decode the whole trace file using Java objects took 53.2 seconds compared to 1.5 seconds when just storing the values in arrays. This is mainly due to object creation. While on its own creating a Java object is relatively quick, creating millions of objects is a slow operation.

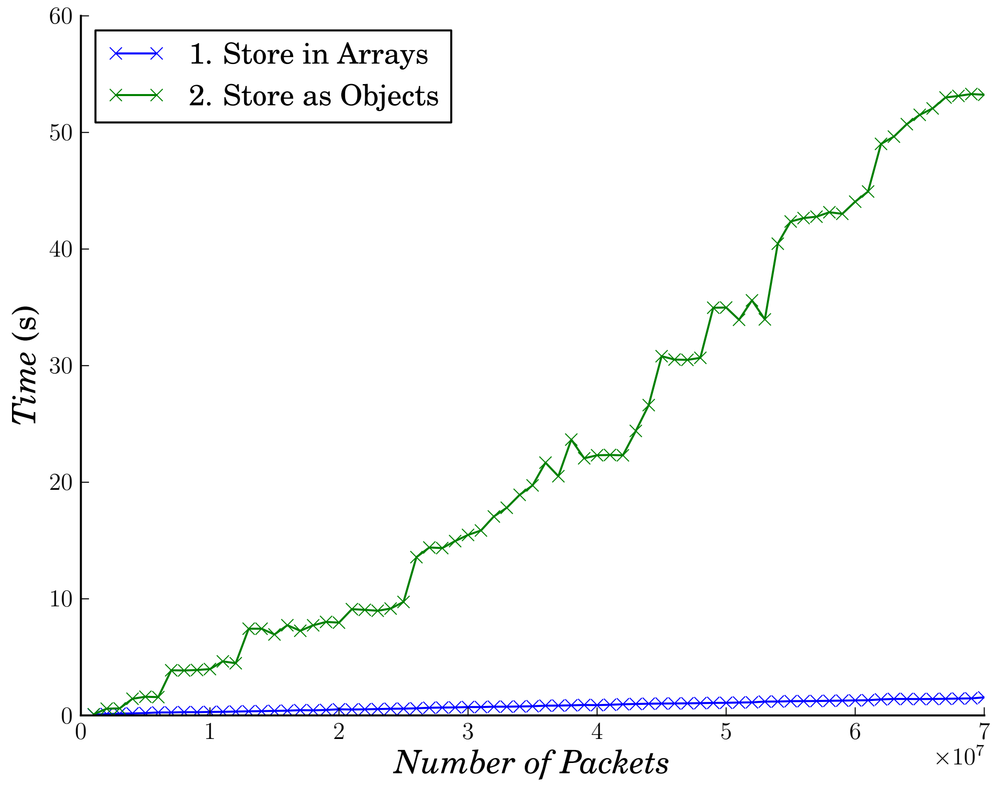


Figure 4.4: The average time taken to read a packet trace and store it in a suitable data structure.

Another advantage of the second implementation decision is the reduced memory footprint. Figure 4.5 shows how the memory footprint of the object encapsulation approach is larger than the array approach. This is mainly caused by the overhead of a Java object. Each packet object has a header of 12 bytes, two Java *long* primitives of 8 bytes each, an *int* of 4 bytes and finally two *byte* primitives of one bye in size. This a packet 34 bytes but, Java objects are always aligned by 8 bytes, meaning the total size of a packet object is 40 bytes. This does not include the reference to the object which add another 4 bytes per object to the memory footprint. In comparison an array has a memory footprint of its length multiplied by the array's primitive size plus an overhead of 12 bytes rounded to 8 bytes. So each packet is approximately 22 bytes in size.

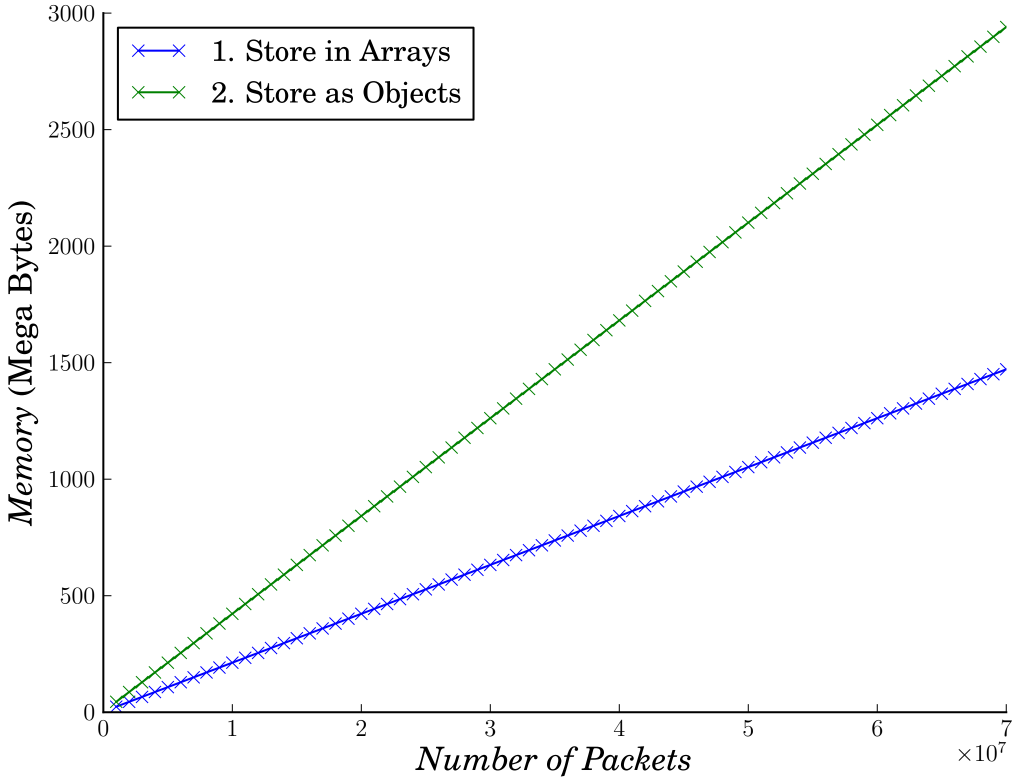


Figure 4.5: The memory requirements as the number of packets increases.

4.3.2 Packet Timeline

This section looks at the packet timeline and the time it takes to plot a packet trace using different optimizations as described in Section 3.2 and 3.4.

The graph in Figure 4.6 shows the time each optimization takes to plot the seventy million packet trace file, after it has been decoded. It can clearly be seen that the naive way of plotting the trace is extremely slow, having a complexity of $\mathcal{O}(n)$ where n is the number of packets. This linear relationship means that naively plotting every point is certainly unsuitable for such a large trace and the problem only gets worse when more traces are added to the packet timeline.

Using the algorithms described in Section 3.4.2, 3.4.3, 3.4.1 and 3.4.4 the plotting times are significantly reduced to almost a constant time compare to the naively plotting the graph. Removing the slow naive plotting time from the graph allows us to see the remaining optimisations in much greater detail, Figure 4.7

Not surprisingly both the selective sampling and bitmap algorithms take roughly the same time, with a maximum difference of approximately 0.03 seconds between the algorithms, for different numbers of packets to plot. This is mainly due to both algorithms performing in a similar way, they iterate through each packet in the trace and decide whether to plot that packet or not.

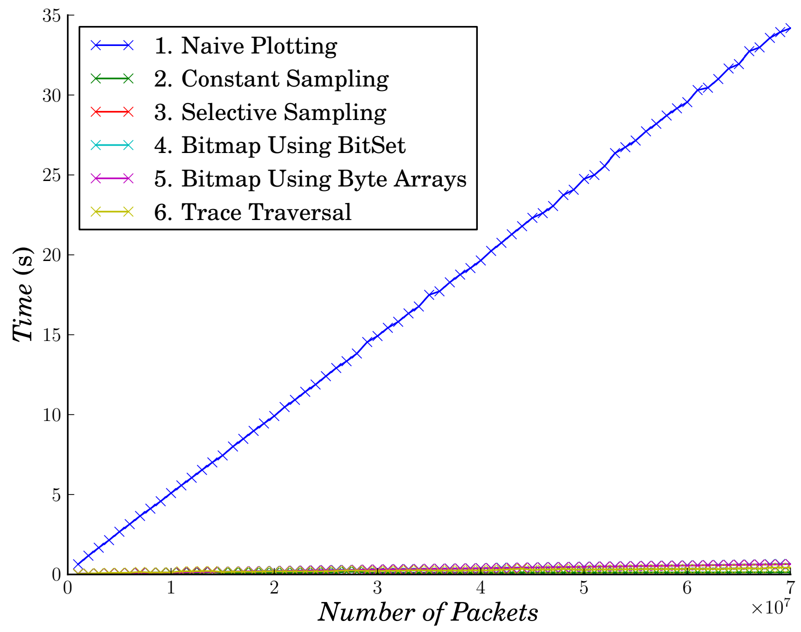


Figure 4.6: The time taken to plot one trace under different optimizations.

On the other hand the trace traversal algorithm has a log shape, as the number of points plotted increases. This is also as expected. The traversal algorithm searches for the next point to plot, and as such only relies on the complexity of that search rather

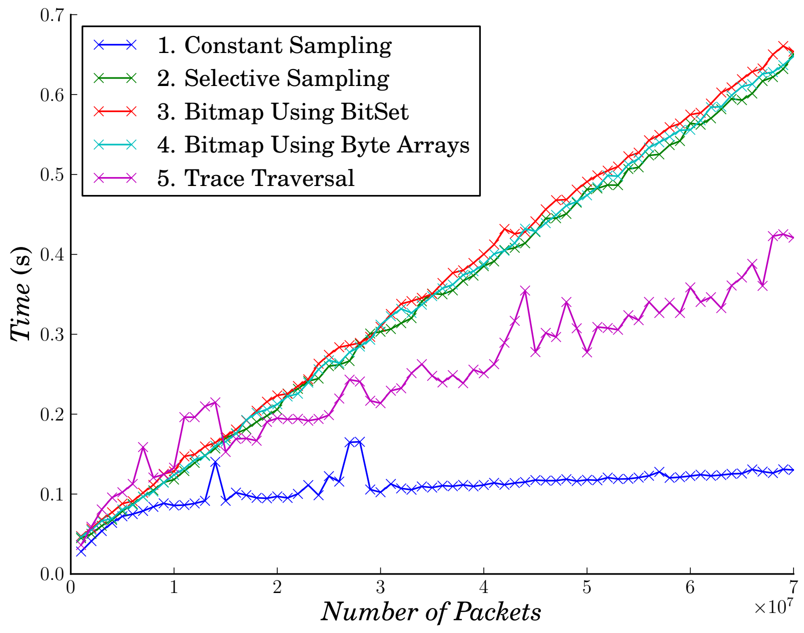


Figure 4.7: The time taken to plot one trace using the constant sampling, selective sampling, bitmap and trace traversal approaches.

than the time to traverse the packet trace.

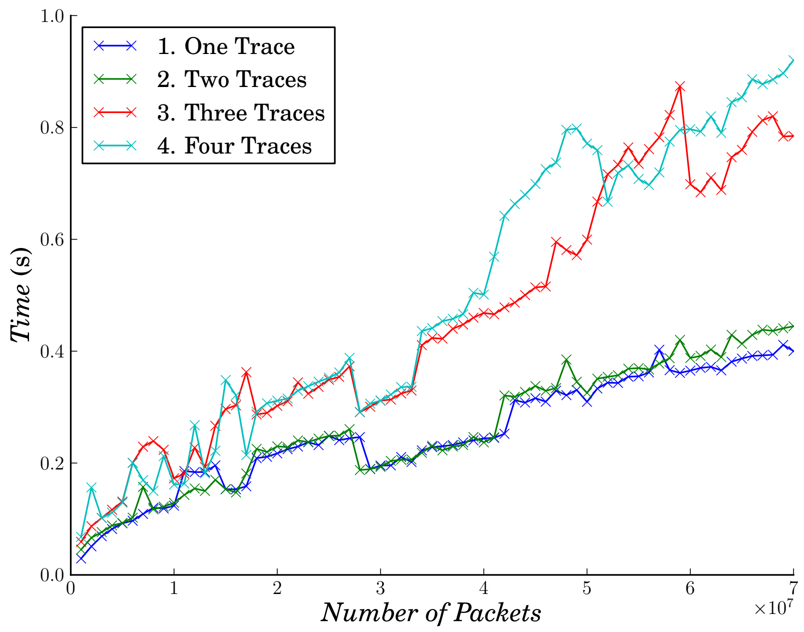


Figure 4.8: The time to plot multiple traces using the trace traversal algorithm. X–axis number of points per trace.

The time to plot the latency graph for multiple traces was measured as the visualisation tool requires multiple traces to be plotted. Figure 4.8 shows the times taken to plot multiple traces, from a single trace to four traces being plotted on the same graph, as the number of points in each trace increase. In each instance the plotting took less than one second, even when four complete traces where being plotted – a total of 280 million packets.

4.3.3 Latency Graph

The latency graph had the same problems that the packet timeline had to overcome, in terms of plotting millions of points, but it also had the additional problem of calculating the latency between each common packet in two traces.

Calculating Latencies

Calculating the latency information involved searching through two unordered list for common elements. This was not a trivial task but was completed in roughly the same time as decoding the packet trace, 0.5s to read a packet trace and 0.6s to for the latency calculations, even though reading the packet trace required I/O. As shown in Figure 4.9 the calculations scale well, with the time taken linear to the number of packets to compare.

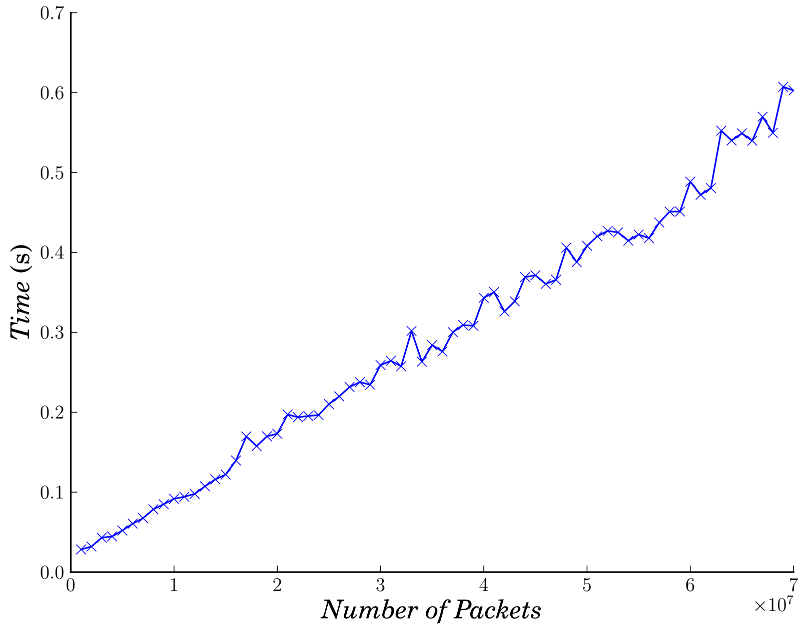


Figure 4.9: The time taken calculate the latency between each common packet in two different traces as the number of total packets compared in each graph increased.

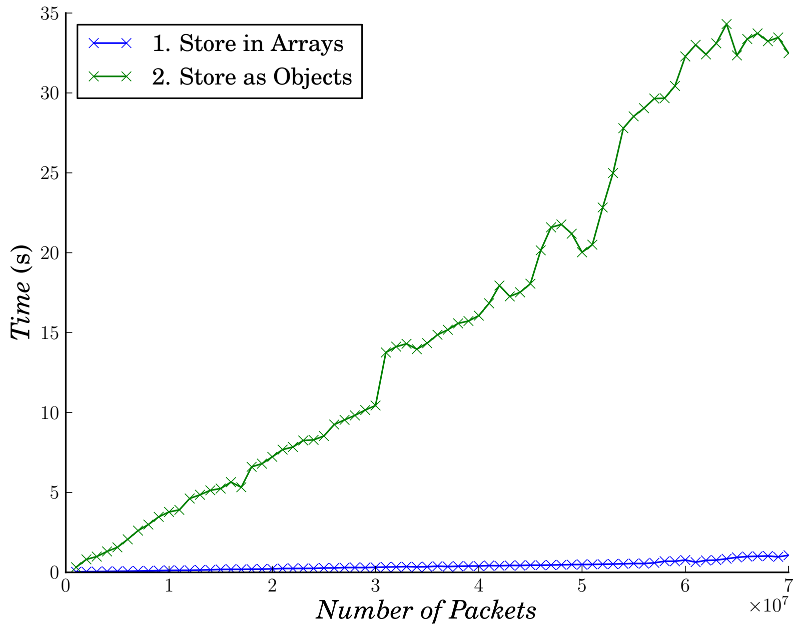


Figure 4.10: The time taken calculate and store the latency between each common packet in a suitable data structure as the number of packets increase.

In a similar way to decoding the packet trace, calculating the latency iterates through both traces and and output the information that is needed to plot the latency graph. Thus in a similar way to trace decoding calculating the latency requires either millions of new object to be created, to encapsulate the data (position and timing information), or to hold the data in sets of arrays.

Figure 4.10 shows the time taken to calculate the latency information for two traces. As expected, creating millions objects significantly slows down the initial creation of the latency graph. While placing each value in a set of arrays is significantly quicker. Comparing two different traces of seventy million packets, which resulted in about twenty three million common packets, takes about 33 seconds when objects are created for each common hash or 1 second when placed in arrays.

Latency Graph Plotting

As described in Section 4.1 point 3, the graphs have to be drawn with sub–second delays to allow for interactive analysis of the traces. Figure 4.11 shows the times taken to plot the graphs for both ways of plotting the latency graph, naively plotting all points on the graph and using a bitmap to avoid drawing points at co–ordinates that have already been plotted.

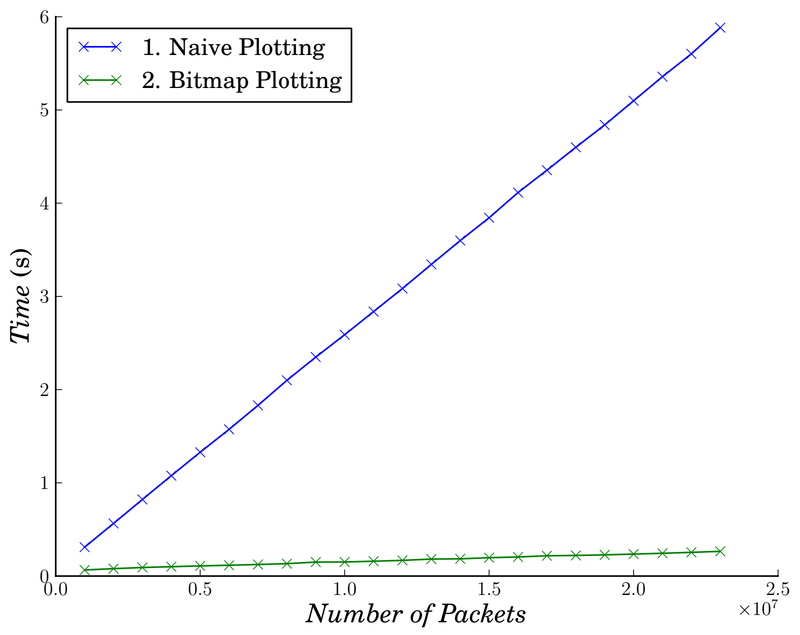


Figure 4.11: The time taken to plot a latency graph, against the number of common packets in the two traces. In this example the two seventy million packet traces have approximately twenty three million packets common to both traces.

Both algorithms show the same characteristics that we have already seen in the time required to plot the packet timeline. The naive plotting algorithm takes significantly longer than the bitmap approach. To plot the whole graph of twenty three million packets the naive algorithms takes 5.8 seconds compared to the 0.26 seconds required to plot using the bitmap approach. This is over a 22 times improvement for the bitmap approach compared to the naive way, and the bitmap approach is far under the sub second delay needed from the program.

4.4 Existing Plotting Tools

This section evaluates the performance of alternative plotting tools. As mention in Section 4.4 none of the other main plotting tools are able to give the performance required for plotting the number of packets that are in a typical trace.

4.4.1 gnuplot

gnuplot is unable to plot the trace file as it is in its current form. While gnuplot is able to plot custom binary files it has limited support for it and was unable to correctly plot the trace file, due to the complexity of the file. Thus before plotting the trace the file had to be translated into a format more suitable for gnuplot. This was done using the set of classes describe in Section 3.1 outputting the values to text file. As the trace file does not contain the transmit time of each packet it was calculated, so the final file that gnuplot was given included the start time and the time to transmit each packet, $length * 0.8$.

gnuplot also does not support the plotting of timeline data as standard, thus a non standard way had to be taken to plot the data. Figure 4.12 shows the command needed to plot the timeline. It plots using the vector command removing the vector's arrow head, thus leaving a line between two co–ordinates. The first column is the start time. The second column is the y co–ordinate, a constant. The penultimate and ultimate columns are just the deltas for x and y respectively, i.e the time to transmit each packet and 0. Thus the plotted line draws a vector from (x, y) to $(x + \Delta x, y + \Delta y)$.

```
plot ``packetTrace'' using 1:2:3:4 with vectors nohead
```

Figure 4.12: The gnuplot command that plotted each packet, where the input file has a format: x, y, Δx, Δy.

The packet timeline was plotted for varying numbers of packets. Figure 4.13 shows how the time to plot increases linearly with the number of packets that are to be plotted. With it gnuplot taking on average 267 seconds, over four minutes, to plot the seventy million packet trace. This is compared to 0.4 seconds for my implementation, thus gnuplot is over 600 times slower. Even when only plotting one million packets gnuplot takes 3.5 second compared to 0.04 seconds for my implementation. gnuplot is more comparable in performance to the naive plotting algorithm which takes 34 seconds to plot seventy million packets. A part of the time difference between the naive plotting algorithm and gnuplot is the need to load into memory the trace each time it is plotted, there is no way of measuring just the plotting time, although adding

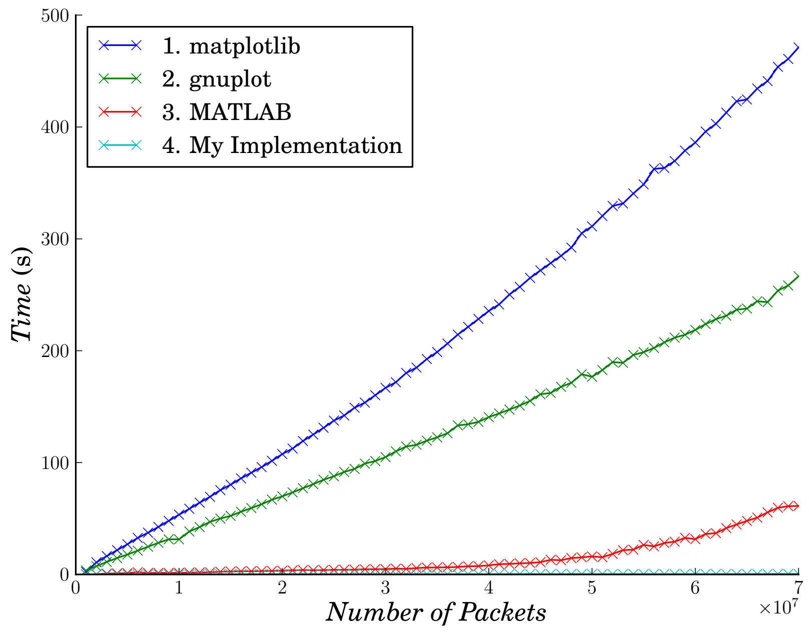


Figure 4.13: The time taken to for gnuplot, matplotlib, MATLAB and my implementation to plot the packet timeline against the number of packet being plotted.

the time taken to for my implementation to load the data and then plot it doesn't significantly close the gap between the two methods.

4.4.2 matplotlib

Unlike gnuplot, matplotlib is far more versatile due to the python integration. This versatility could of potentially been used to convert the binary trace file into a format that was usable by the matplotlib library, but as that would be essentially re–implementing the functionality described in Section 3.1 this route was not explored. Rather than measuring the time needed to decode the trace file, only the time required to plot the packet trace graph was measured, as with gnuplot.

Several options were available when plotting the graph. This included naively using the libraries plot function, as plotting one million points using this method took over an hour this was very swiftly discarded. Two other options were also explored, one using line collections, again this was far too slow with one million points taking 37 seconds to plot. The final method creates a single list of points with each set of coordinates delimited by `None` and then this is plotted. Figure 4.13 shows the time taken for the final plotting method.

Having a single list of points each separated by `None` is far faster than using line collections and the standard plot command, with one million points taking 2 seconds to plot. Although even this is very slow taking 7 minutes 51 seconds to plot the whole trace file, nearly double the time taken for gnuplot. Given that gnuplot was unsuitable matplotlib was also far below the performance required.

4.4.3 MATLAB

MATLAB is very much like matplotlib in its ability to manipulate data, and for the same reasons as matplotlib I did not explore the time require for MATLAB to manipulate data into the correct format needed for it to plot but rather only the time required for the graph to be drawn.

As with matplotlib I first looked at naively using the standard plot function. This predictably was very slow, but significantly better than matplotlib's standard plot. To plot eleven million point took close to an hour compared to matplotlib only being able to plot one million points in roughly the same time. No more testing was completed after plotting the eleven million points, instead a more optimised version[1] was tested. The results are shown in Figure 4.13.

MATLAB is significantly faster taking 0.1 seconds to plot a million points. This is very good but still slower than 0.04 seconds needed for my implementation. As with the other two plotting tools MATLAB does not scale well, plotting all seventy million packets takes 61 seconds. This is better than gnuplot and matplotlib but still does not fall under the "interactive" timescale that are expected.

4.5 Summary

In this chapter I have evaluated the performance of my implementation of the visualisations. I have the measured the time required to decode the packet trace files using different methods that Java has available to read files. I have looked at how the decoded files were stored justifying the design decisions made in Chapter 3. The time and memory requirement need to plot the graphs were measured and compared under different plotting strategies. Finally this was compared to the time that other well established plotting tool took to draw the packet timeline. All other tools failed to match the performance of my plotting tool and were all significantly worse. Thus each of the projects goals were successfully completed.

[1]The optimisations suggested in the documentation for MATLAB were implemented: `http://www.mathworks.co.uk/help/matlab/creating_plots/optimizing-graphics-performance.html`

Chapter 5

Conclusion

This dissertation has so far described the preparation, implementation and evaluation that was required to successfully create a visualiser for packet traces. This chapter will conclude my dissertation summarising what has been accomplished, the results and suggests possibilities for future work.

5.1 Achivements

This project was a success and I was able to implement a packet trace visualiser. I was able to decode the packet traces into memory and plot both a packet timeline, for one or more traces, and a latency graph, for two traces. The graphs could be manipulated with less than sub second delays, even when many traces were being plotted. Thus all initial goals were successfully fulfilled and in addition a number of optional extensions were also implemented.

The visualiser is able to plot the desired graphs and, to the best of my knowledge, far out classes all other existing plotting tools in both plotting speed and the ability to interactively manipulate the graphs.

5.2 Lessons Learnt

This project has taught me far more than I would ever be able to learn from just conventional lectures and supervisions. Most notably was the amount of work that was needed to complete a project of this size and complexity was far too easy to underestimate. This was demonstrated when trying to use the unfamiliar plotting tools: gnuplot, matplotlib and MATLAB. From reading the documentation of the three tools to trying to use them in the most efficient way to plot data in one dimension, something wasn't supported as standard.

While the plan for my project included buffer zones in case of delay they were very easy to use up as supervision work built up. Thus this project made me improve my time management skills, especially over a larger timescale. It also made me gave me experience with of some of the softer skills in software engineering, for example the need to throw away code that was less than optimal or that was fundamentally flawed and restart using a better method.

Finally as this project dealt with large amounts of data it was very hard to test and verify the correct functions of the programs. This was especially prevalent in the early stages of the development. The development cycle relied on an incremental and iterative approach thus naive versions of each function were written before being improved and optimised. The initial versions often took a significant amount of time to execute, over ten minutes in some cases. This problem was mitigated by sub sampling the data and limiting the amount of test data that was used before eventually testing on the full set.

5.3 Future Work

The visualiser is a fully functional system however there is scope for improvement. A few possible extensions are given below.

5.3.1 User Interface

As it stands the user interface, while functional, can be improved. The layout can be changed and optimised making the graphing environment more intuitive to use. At the moment only one graph is shown at a time, this can be changed to allow multiple windows to plotted each with their own distinct graphing options. Other simple UI changes can be made, for example click to zoom or the ability the change the graphing colours, that will help with usability. There is also a case for greater OS integration over the current Swing based style.

5.3.2 Memory usage

There is also the original extension that was listed in the project proposal. That was to create an implementation that would work efficiently in systems with low memory availability. Although it should be noted that the main defence against having to use a scheme like this was the efficient use of data structures and primitives to minimize memory usage. While the current system is able to function when memory usage exceeds the amount of system memory often the OS is using swap space and performance degrades. By researching different algorithms there may be a potential performance increase when this situation is encountered.

5.3.3 Optimisations

Finally there is a possibility that other optimisations could be applied to that graphing algorithms that may improve performance, especially in the latency plot as this is not as fast as the packet timeline. This could potentially be in conjunction with the UI improvements, for example pre–computing the graphs at different scales such that when the zoom is used the replot time is reduced.

The optimisations presented in this dissertation have been language agnostic, relying on clever algorithms and exploiting patterns in the data. Another potential approach might be to implement the program using hardware acceleration, for example taking advantage of the ubiquity of GPUs by exploring openGL and related APIs.

Appendices

Appendix A

Existing Plotting Tools

The code used to plot the graphs for each of the three main plotting tools is given here.

A.1 gnuplot

```
set yrange [0.99:1.01]
plot ``packetTrace'' using 1:2:3:4 with vectors nohead
```

Figure A.1: The gnuplot command that plotted each packet, where the input file has a format: x, y, Δx, Δy.

A.2 matplotlib

```
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
import time

xStart  = []
xEnd    = []
linecol = []
xValue  = []
yValue  = []
with open('trace.txt', 'r') as filename:
    for line in filename.readlines():
        elements = line.strip().split('␣')

        # Setup arrays for naive plotting
        xStart.append(int(elements[0]))
        xEnd.append(int(elements[1]))

        # Setup for line collection
        linecol.append(((int(elements[0]),1),
                       (int(elements[1]),1)))

        # Setup for single list of points
        xValue.append(int(elements[0]))
        xValue.append(int(elements[1]))
        xValue.append(None)
        yValue.append(1)
        yValue.append(1)
        yValue.append(None)

# Plot using hlines
startTime = time.time()
plt.hlines(0.5, xstart, xstop)
elapsedTime = time.time() - startTime

# Plot using line collection
startTime = time.time()
lines = LineCollection(linecol)
axes  = plt.gca()
axes.add_collection(lines)
axes.autoscale_view()
elapsedTime = time.time() - startTime

# Plot using single list of points
startTime = time.time()
plt.plot(xValue,yValue,'b-', alpha=0.1)
elapsedTime = time.time() - startTime
```

Figure A.2: The python program that plotted the packet timeline graph. All three versions are shown, naively using the plot command, using line collections and finally creating a single list of points for the x and y axes with each set of co-ordinates delimited by `None`

A.3 MATLAB

```
fileID = fopen('dag_0.txt');
c = textscan(fileID, '%u64 %u64');
fclose(fileID);

tic;
plot([c{1}, c{2}],[1,1]);
elapsedTime = toc;
```

Figure A.3: The naive MATLAB program to plot the packet trace.

```
function plotTimes

    function ManAxMode(h)
        pn = {'ALimMode',...
            'CameraPositionMode','CameraTargetMode',...
            'CameraUpVectorMode','CLimMode',...
            'TickDirMode','XLimMode',...
            'YLimMode','ZLimMode',...
            'XTickMode','YTickMode',...
            'ZTickMode','XTickLabelMode',...
            'YTickLabelMode','ZTickLabelMode'};
        for k = 1:15
            pv(k) = {'manual'};
        end
        set(h,pn,pv)
    end

    x = zeros(210000000,1);
    y = zeros(210000000,1);

    fileID = fopen('trace.txt');
    for i=1:3:210000000
        c = textscan(fileID,'%u64 %u64',1);
        x(i) = c{1}(1);
        x(i+1) = c{2}(1);
        x(i+2) = nan;
        y(i) = 1;
        y(i+1) = 1;
        y(i+2) = nan;
    end
    fclose(fileID);
    largestValue = x(209999999);

    % Initial plot to get the correct settings and
    % then toggle the setting to manual
    figure('Renderer','painters');
    h = axes('XLim',[0,largestValue],'YLim',[0,2],...
        'Position',[.145,.25,.775,.675],...
        'Drawmode','fast');
    line('XData',[0,largestValue],'YData',[1,1]);
    ManAxMode(h);

    % Plotting the graph
    tic;
    line('XData',x(1:end),'YData',y(1:end))
    elapsedTime = toc;

end
```

Figure A.4: The optimised MATLAB program to plot the packet trace.

Appendix B

Project Proposal

Christopher Wheelhouse
St. John's College
crw47

Part II Computer Science Project Proposal

Network Packet Trace Visualisation at Scale

Project Originator: Malte Schwarzkopf and Matthew Grosvenor

Resources Required: See attached Project Resource Form

Project Supervisor: Malte Schwarzkopf and Matthew Grosvenor

Director of Studies: Malte Schwarzkopf

Overseers: Dr Andrew Rice and Dr Timothy Griffin

Introduction and Description of the Work

In recent years modern, data centres have become more and more important, as large volumes of data have become readily available data centres are able to help analyse and process this data. One key concern in a modern data centre is the need to quickly transfer data from one node to another. As a data centre inherently relies on distributed applications, it must generate a significant amount of network traffic between nodes, especially as it is often quicker to retrieve data from another node's main memory rather than accessing one's own hard drive.[1]

The Computer Laboratory's Systems Research Group is currently developing a bufferless network architecture called R2D2. R2D2 aims to create a resilient realtime network through reducing latency, this is done by moving buffers to the end point of the network as well as sacrificing bandwidth. Hence any packets are able to traverse the network quickly without being delayed at a switch's queues.[2]

As part of the research into R2D2 and high speed networks it is vital that network packet traces can be analysed effectively and efficiently. Such analysis will look into how the packets behave in the network. It will look at the characteristics of different traffic patterns generated by applications, specifically looking at the spacing between packets and if "packet trains"[3] are generated. Packet trains are a series of packets in very short succession. They can potentially cause a latency spike for an application if two packet trains are merged into the same outgoing port at a switch.

The aim of this project is to create a high-speed interactive network trace analyser. The tool should be able to process and visualise traces on an interactive time scale; after loading into memory and without using too many system resources. Due to the size of the trace files – several gigabytes for a few seconds – current plotting tools for visualising the data such as gnuplot or matplotlib are not suitable. The current tools plot each point individually, even when zoomed out, thus they replot and recalculate everything at each zoom event. Hence sutible data structures and algorthims will have to be designed and implemented such that the systems is both effieient in its resource usage and is quick to recalculate and replot the graph when needed.

Resources Required

The project will be developed with my personal machine and regular backups will happen to the PWF's, SRCF's and Dropbox's servers.

I plan to use Java, Qt and Java 2D for manipulating the data, UI design and the visualisation drawing respectively.

Stating Point

To complete the project, I intend to build on the following resources:

1. **Previous Programming Experience**
   I am familiar with the Java and Qt. I will also be building on top of the programming experience I gained during my internship over the long vacation.

2. **Part 1B Courses**
   I will also be drawing on the knowledge I learnt from the *Computer Networking* and *Further Java* courses in Lent.

Substance and Structure of the Project

The main objective of this project is to design and implement a high-speed interactive network trace analyser that is able to quickly render visualisations of one or more traces without needing an extreme amount of memory.

The packet trace is supplied in a binary-encoded file and thus it must first be decoded and then placed into memory. The representation that the data is placed in should be carefully considered such that it does not waste too much memory but is flexible enough to allow quick access to the data when the trace needs to be redrawn. This is especially important as a trace file could contain between 10 and 100 million packets.

Each packet sample is 24 bytes and contains a timestamp, the length of the packet, a count of the number of packets lost by the capture card before the current packet, the packet type and also a hash of the packet contents. This means that for a 50 million packet trace file, the input data will occupy approximately 1.1GB of memory. This excludes any operational overheads, for example the space required for the in-memory representation of the trace.

The primary visualisation I intend to support is a packet timeline. This shows how the link was utilized by each packet transmission, thus allowing for easy identification of packet trains. It should also be capable of displaying more than one trace, so the user can easily compare different packet traces – after synchronising their start times.

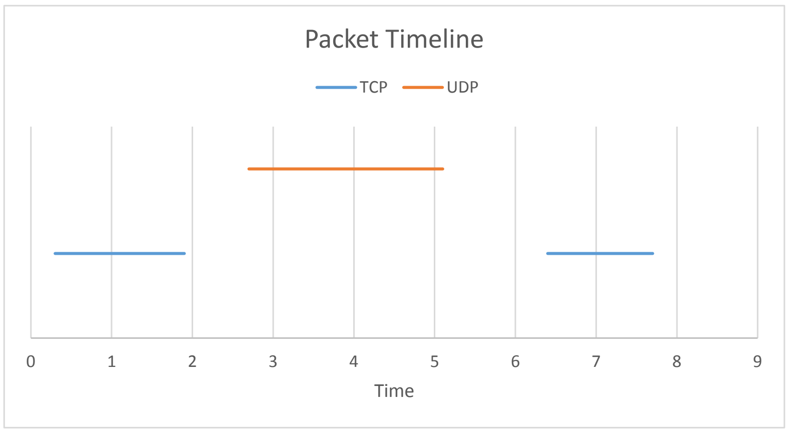


Figure 1: Packet Timeline

Figure 1 shows how a TCP packet uses the link from just after time 0 until just before time 2. The link is then idle until just before time 3 when a UDP packet is sent along the link until just after time 5.

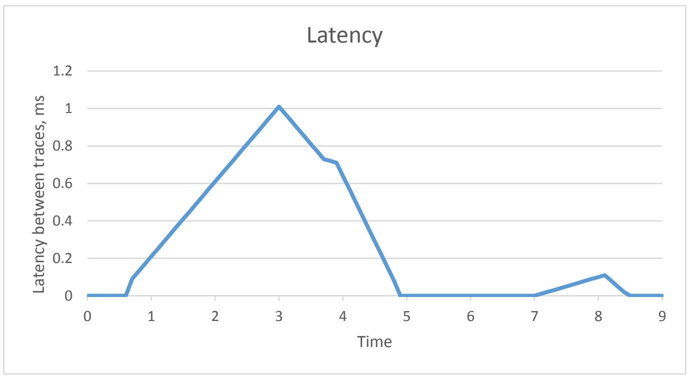


Figure 2: Latency between two traces

While Figure 1 represents the primary graph that is plotted, other auxiliary graphs will also be created to help with analysis. For example, Figure 2 shows the latency between two packet traces based on their hashes and difference in timestamps. The graph shows how the first there is no difference between the two traces and then at time 0.7, trace 2 starts to have a timestamp that is relatively after the corresponding packets of trace 1, i.e. trace 2 packets arrive after the corresponding packets in the first trace did. At time 3 the latency starts to decrease.

A key challenge in designing in the tool will be choosing the algorithms and data structures that are able to traverse the data effectively and then plot the traces. The user should be able to:

- Specify the time frame to look at and the zoom level to consider.
- Scroll up and down the network trace.
- Select the type of packets to be shown, e.g. only TCP.
- Specify packets of certain lengths, e.g. packets of length between 64 and 128 bytes.

Consideration of the Big $\mathcal{O}$ complexity bounds of the data structures and algorithms is essential so that the trace can be interactively redrawn.

The final aspect of the project would be to evaluate the performance of the tool. This will be done by:

- Measuring the plotting delay as a function of the number of data points or trace size.
- Measure the memory consumption using a system profiler.
- Compare the previous benchmarks to the results of existing plotting tools.

Optional Extensions

1. User interface: To make it easier for a user to interact with the program, see the network trace and manipulate the visualisation; a user interface could be designed.
2. The situation when the trace does not fit into main memory: It is possible that either the machine does not have enough memory for normal use, or the trace happens to be extremely large. In either case it is very likely that programs performance will be significantly degraded. Thus, the tool could apply a different way of representing the data in memory and a different algorithm to exploit the different representation, potentially sacrificing some efficiency for reduced memory usage.
3. Additional statistics: In addition to the main visualisation, additional statistics could be computed that help with analysis or can notify the user of malformed data in the trace.

Success Criteria

For the project to be considered a success the following criteria must be achieved:

1. The network packet trace file must be decoded.
2. A data representation to store the decoded file must be researched and designed.
3. Algorithms must be designed such that the packets can be visualised and analysed at iterative time scales, i.e. the trace must be re-drawn with approximatly less than sub-second delays.

4. Tests must be carried out to determine the speed and efficiency of the tool and how it compares to the standard plotting tools.
5. The dissertation should be planned and written.

In addition to the above points the following extensions will increase the success of the project but are not essential.

1. Add support for when the packet trace does not fit into memory.
2. Implement a user interface.
3. Add support for additional statistics that can help with analysing the trace.

Timetable and Milestones

0. 10th October – 25th October

- Investigate libraries and test them.
- Research existing plotting tools.

Milestone: Submit Project Proposal.

1. 26th October – 15th November

- Investigate data structures that could be used to store the decoded trace.
- Start implementing the ability to decode the trace files.

2. 16th November – 6th December

- Start implementing algorithms to plot the trace.

Milestone: Finish implementing the ability to decode the file and have it stored in a suitable representation in memory.

3. 7th December – 27th December

- Continue implementing algorithms to plot the trace.

Milestone: Finish implementing algorithms to plot the trace.

4. 28th December – 17th January

- Implement the ability to display the trace on a canvas.

Milestone: Be able to display the trace on a canvas.

5. 18th January – 30th January

- Finish implementation of the main program.
- Start extensions.

Milestone: Progress report submitted and presentation prepared.

6. 31th January – 20th February

- Continue with extensions.
- Start benchmarking.

7. 21st February – 13th March

- Start writing dissertation.

8. 14th March – 3rd April

- Continue dissertation.

Milestone: Finish a draft of the dissertation and submit to my supervisors and DoS.

9. 4th April – 24th April

- Complete dissertation.

10. 25th April – 16th May

- Revise for exams.

Milestone: Submit dissertation.

References

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In HotOS, 2011

[2] M. Grosvenor, M. Schwarzkopf, R. Watson, and A. Moore. R2D2: Bufferless, Switchless Data Center Networks Using Commodity Ethernet Hardware. In SIGCOMM, 2013

[3] R. Kapoor, A. Snoeren, G. Voelker, and G. Porter. Bullet Trains: A study of NIC burst behavior at microsecond timescales. In CoNEXT, 2013