REALISTIC TRAFFIC SHAPING IN DUMMYNET LINK EMULATOR

by

Aisha Syed

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computing

School of Computing

The University of Utah

August 2014

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Aisha Syed**

has been approved by the following supervisory committee members:

Robert Ricci	, Chair	**05/02/2014** Date Approved
Sneha Kasera	, Member	**05/02/2014** Date Approved
Jacobus van der Merwe	, Member	**05/02/2014** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Dummynet is a link emulator that can be used by itself, as well as integrated within testbeds such as Emulab. Despite its popularity in the research community, Dummynet still lacks the ability to precisely emulate certain real network effects. In particular, it has no support for packet reordering. Since reordering is a common and prevalent network phenomenon just like packet loss or delay, it cannot be ignored when implementing emulators if we want to provide realistic emulation.

It has been observed that networks suffer from reordering caused by packet striping, retransmissions, load balancing, multipath forwarding, etc. This has significant negative effects on the performance of both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). With the increase in prevalence of real-time streaming UDP applications such as video conferencing and Internet Protocol Television (IPTV), it has become important to focus on this problem which affects the performance of all these applications. Research into models and tools to diagnose and understand reordering requires that a sophisticated metric be used to describe it.

So, in this thesis, I make two contributions: improving the realism of traffic shaping in Dummynet emulator by adding support for emulation of reordering, and an algorithm, a max-flow solver, that generates reordered sequences to be used by Dummynet, from a sophisticated reordering metric called Reorder Density (RD). My implementation enables the user to specify the desired amount of reordering in a metric, such as RD (or even others), and have Dummynet generate traffic that is reordered according to the input metric's value. This is accomplished within Dummynet by the use of a newly implemented scheduler.

I conclude my thesis with an evaluation using real and software generated network traces to show that the algorithm is scalable and the implementation works correctly. Also, a datapath evaluation to show that my modifications to Dummynet do not result in any unnecessary increase in emulation running time is included.

For my family and friends.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First off, I would like to thank Robert Ricci for patiently guiding me for the past two years. He has been an exceptionally helpful and knowledgeable advisor and the completion of this thesis would not have been possible without him.

I am grateful to Sneha Kasera for his words of encouragement, and his advice to get into research through independent studies during my very first semester, one of which finally resulted in the form of this thesis. My thanks also go to Kobus Van Der Merwe for his helpful advice about conducting proper research and for serving on my committee and providing useful feedback.

I would also like to thank Suresh Venkatasubramanian for taking time out to answer my questions and suggesting the application of max-flow to solve the sequence regeneration problem discussed in this thesis.

Last but not the least, my thanks go to all my friends and family for their love, support, and encouragement. And to my dearest friend, Bilal, for being the wonderful source of happiness that he is.

CHAPTER 1

INTRODUCTION

Packet reordering is a network phenomenon that is just as common and prevalent within the Internet as packet loss or delay [18] [15] [9]. It is caused by various factors such as packet striping at layer 2 and 3, retransmissions due to loss, duplication, load balancing or priority scheduling within routers, and route fluttering or multipath forwarding among many others [20] [19].

Despite studies that show its prevalence, and the fact that it affects both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) and can significantly degrade application performance [16], reordering has somewhat been ignored by researchers and not enough attention has been paid to understanding its nature. Bennett, et al. [9] found in their famous study that reordering is not just a pathological behavior as researchers, such as Paxon [18] had posited, much of it occurs as a natural result of increasing parallelism within the Internet. Reordering can make it hard for TCP to grow its congestion window, cause it to make incorrect estimations of round-trip times, result in unnecessary retransmissions, and thus degrade application performance overall. A study [12] conducted to quantify the effect of reordering in a backbone link found that even a small amount of reordering coupled with some packet loss can cause significant degradation in link utilization and thus application throughput, especially for long-lived flows.

Reordering also has an adverse effect on delay-based realtime UDP applications such as video conferencing [20] [13]. It has become especially important to focus on this problem now as streaming media, Voice over Internet Protocol (VoIP) and Internet Protocol Television (IPTV) are becoming increasingly prevalent on the Internet.

As Piratla, et al. [15] argue, measuring and characterizing reordering and devising models for understanding reordering can help us deal with it in scalable ways. For doing all of this, researchers need a realistic platform for experimentation and testing. This is provided by means of either simulation, emulation, or live network testing. These three are all various ways to evaluate network and distributed systems research. While simulation and

live network testing occur at opposite extremes, emulation takes an intermediate approach. It introduces the simulator into a live network, thus combining the benefits of simulation and live network testing and providing researchers with a controllable, repeatable, transparent environment like simulation, while not sacrificing the realism by having real applications as traffic generators. For this reason, the focus of this thesis is on support for reordering using emulation.

Even if researchers are not performing experiments specifically related to the phenomenon of reordering, due to the arguments presented earlier, ignoring reordering during traffic shaping in the emulator will result in experimental traffic that is not representative of real network traffic. So, if we want to provide researchers with emulated traffic that is realistic then we surely need to include support for reordering in our emulators along with the usual support for other important network characteristics such as packet loss or delay. I chose Dummynet [10] network emulator for this implementation. It is popular in the research community [1] due to its good feature set, low learning curve, wide availability in various platforms, and the fact that a Dummynet-enabled bridge can be inserted in an existing network without changing the configuration or disrupting any existing software installations. Dummynet has also been integrated in testbeds such as Emulab [7] and PlanetLab [6].

Once the choice of emulator has been made, I needed to decide what metric the emulator is going to take as input. I believe a good metric to quantify reordering is important for fine-grained emulation. Basic metrics that are limited in their usefulness, such as percentage of reordering or n-reordering [3] have been used to describe packet reordering in networks. Then a comprehensive and useful metric called Reorder Density (RD) [20] [5] was devised somewhat recently to measure reordering. RD can help researchers evaluate their protocols and implementations with respect to their impact on reordering. It can further help them devise models for reordering and thus gain useful insights about the nature of reordering, its causes and impacts, and amount of buffers needed for recovery. This made me decide to use RD as the default metric for emulation support in Dummynet.

There can be two kinds of algorithms for any given metric: a calculation algorithm that given a packet sequence calculates the metric from it, and a sequence regeneration algorithm that does the opposite of this and given the calculated metric, regenerates the original or an equivalent (in the amount of reordering) packet sequence from it. What I needed for the emulator was a sequence regeneration algorithm, since the researcher or user

of the emulator is going to give the emulator the amount of reordering expressed in RD metric and the emulator then uses this input metric and generates a packet stream from it. Unfortunately, an RD sequence regeneration algorithm does not exist as indicated by a literature search and so devising such an algorithm is another contribution I make in this thesis.

The fact that researchers will preferably want to use basic metrics, such as percentages or n-reordering for doing course-grained or quick emulation, requires that our emulator be able to support them too. This problem can be solved if Dummynet is made to take as input a generic pattern of reordering and does traffic shaping based on it. This pattern can be generated offline through RD or any other reordering metric. To enable this, I divided my implementation into two components, one inside the Dummynet for supporting reordering, and one outside that is packaged with Dummynet as a command line tool and basically implements the RD sequence regeneration algorithm. This tool can then run offline on the client machine, it takes RD as a metric and generates a reordered packet pattern or sequence of numbers. This can then be fed into Dummynet with appropriate configuration options, which result in Dummynet triggering my reordering implementation within for traffic shaping.

So, to sum it up, my project aimed at introducing support for controlled and fine-grained emulation of reordering in Dummynet and chose RD, due to its usefulness and other important attributes, as the default metric to be packaged as command line tool along with the Dummynet reordering implementation. Also, the implementation was made flexible enough to take any kind of metric as input and be not just limited to RD so users can skip the use of the provided tool and use any other metric they choose.

1.1 Thesis Statement

It is possible to make Dummynet emulate more realistic network conditions by making it support emulation of reordering.

1.2 My Contributions

To restate, my contributions include the following.

- A sequence regeneration algorithm to generate a reordered packet sequence from a given RD.

- Support for packet reordering in Dummynet using reordering metrics such as RD.

1.3 Organization

The remainder of this thesis is structured as follows.

- Chapter 2 describes the background and related work and gives the reader enough information about RD and Dummynet to help them understand my implementation, which is described later.
- Chapter 3 then goes on to describe my contributions: the RD sequence regeneration algorithm, its integration into Dummynet, and the entire experimenter workflow.
- Chapter 4 provides an evaluation for my implementation using real and software-generated network traces.
- Chapter 5 finally concludes this document by restating the thesis, how I proved it to be true, and possibilities for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this section, I introduce RD, the default metric I chose for the network phenomenon I am trying to emulate, namely, reordering, and also provide a brief background of the Dummynet architecture to help the reader understand how reordering support will be incorporated. Specifically, Section 2.1 gives an introduction to the reordering metric RD, its usefulness and various attributes, and its calculation algorithm. Section 2.2 then describes the Dummynet architecture.

2.1 Reordering Metric

Percentage of reordered packets, n-reordering, and reordering extent, all standardized by Internet Engineering Task Force (IETF) [4], are some basic metrics that have been used to specify the amount of reordering in a packet stream. However, as Piratla and Jayasumana [17] argue, RD is more sophisticated and comprehensive, and also possesses attributes deemed to be important for a reordering metric [17] [9]. These include the following.

- Capture reordering: The metric should be able to capture the amount of reordering in a stream. While some metrics, such as n-reordering, are lateness-based metrics in that they only consider late packets to be re-ordered, others are earliness-based metrics and consider only early arriving packets to be reordered. RD, on the other hand, provides a complete picture by capturing information about both early and late packets.
- Orthogonality: It should be independent of, or have low sensitivity to other network phenomena, such as loss and duplication. RD is not affected by duplicates and declares packets as lost if they do not arrive within a certain threshold.

- Usefulness: In addition to just capturing the amount of reordering, a good metric should be useful to the application or resource management schemes. For example, RD can help in TCP flow control, provide estimates for buffer size that would be required to recover from reordering, and can also help in network diagnosis by hinting about the possible causes of reordering as demonstrated by Piratla [11]. RD also has the very useful property in that given the RDs for two individual subnets, the collective RD for the end-to-end connection formed by cascading the two networks can be predicted [14]. This is especially helpful for measuring the end-to-end RD of a complex network. Finally, parameters, such as 90th percentile, median, and average, can also be derived from RD, if needed.
- Low evaluation complexities: To allow for fast on-the-fly computation, the metric should have low space and time complexities. RD has constant size buffer requirements and the time complexity is O(N). Other metrics such as n-reordering or reordering extent have spatial and time complexities of O(N) and O(N^2), respectively.
- Robustness: The metric should be robust against different errors and network phenomena. These may include rogue packets, a packet with a very large sequence number due to some error or sequence number wraparound, burst of losses, etc. The use of a threshold allows RD to minimize the effect and recover quickly from many such errors or peculiarities. This is unlike other metrics mentioned above which are often unable to counter these events and result in having a disproportional effect on the reordering measurements.

2.1.1 Reorder Density (RD)

RD is a metric that captures the amount of reordering by measuring the displacements of packets from their original positions [20] [5]. The following is an example of how RD is calculated from a given sequence.

Let us say the sender sends the sequence of N = 6 packets:

Sequence sent:

1	2	3	4	5	6

The network causes reordering and the following is what gets received by the receiving side:

Sequence received:	4	1	5	2	3	6

The receiver assigns a receive index (RI) to each of the packets it receives according to the order of its arrival, such as packet 4 arrives first so it gets the RI = 1, packet 1 comes next so it gets RI = 2, and so on. Then, displacement D of the received packets is calculated. The displacement of a packet is defined as the difference between RI and the received sequence number of the packet, i.e., the displacement of packet i is RI[i] - i. Thus, a negative displacement indicates the earliness of a packet and a positive displacement the lateness, while a displacement of zero indicates that the packet has arrived in order. Also defined is a displacement threshold (DT) on the displacement of packets. It allows the RD metric to classify a packet as lost or as a duplicate such that a packet is considered lost if it does not arrive within a certain defined displacement threshold DT, and similarly, a packet is considered duplicate and discarded if another packet with the same sequence number has already been received within the DT window. The DT value is selected by the user based on the TCP send/receive windows or the nature of the application and the network, such as, for VoIP applications it can be selected based on the maximum time duration the application waits for a packets arrival before considering it lost. Finally, displacement frequency FD[k] is defined as the number of received packets having a displacement of k, where k takes values from -DT to DT.

Expected sequence:	1	2	3	4	5	6

Received sequence:	4	1	5	2	3	6
Receive Index (RI):	1	2	3	4	5	6
Displacement (D):	-3	1	-2	2	2	0

Now the reorder density RD of a sequence is defined as the distribution of the displacement frequencies FD[k] normalized with respect to N', where N' is the length of the received sequence after ignoring lost and duplicate packets. N' is equal to the $\sum_{k=-DT}^{DT} FD[k]$, (i.e., for k in [-DT, DT]). For the above received sequence, RD is calculated as shown in Table 2.1.

Table 2.1: RD Generated for sequence (4 1 5 2 3 6); N = 6. Percentages also shown.

k	FD[k]	RD[k] = FD[k] / N'	Percentages = RD[k] * 100
-3	1	0.1667	16.67%
-2	1	0.1667	16.67%
0	1	0.1667	16.67%
1	1	0.1667	16.67%
2	2	0.3333	33.33%

As described earlier, in an emulator, to provide reordering support, we need the ability to regenerate the original packet sequence given its RD distribution by the user. The algorithm for regeneration is my contribution and will be described in later sections.

Table 2.2 shows an RD I generated from part of a real TCP trace [2] which collected 145 hours of packet data on 6 web sites from the host lamar.colostate.edu, CO, USA.

2.2 Dummynet Architecture

Currently, Dummynet does not include built-in support for introducing reordering in the traffic. However, a trick can be used to introduce reordering in the packet stream [11]. It involves configuring multiple pipes (pipes being Dummynet's internal structures that represent physical links) with different bandwidths or delays. Then the pipes are given different probabilities and the incoming packets are sent through them based on their probabilities. Thus, reordering is achieved by emulation of multipath. However, this approach only provides course-grained and uncontrolled reordering and proves to be insufficient if we need repeatability and precision.

Dummynet processing pipeline emulates physical links using structures called pipes which are themselves composed of individual delay and bandwidth queues. Packets are enqueued when they enter Dummynet, processing occurs, then they are dequeued and sent out in the network. To cause delay, the delay queue is used. At the time of dequeuing, it is checked whether the user-specified delay time has been satisfied, if not, then Dummynet waits until delay amount of time to perform the dequeue. Then, the bandwidth queue is used that dequeues packets at the rate of the user-specified bandwidth. Similarly, more functions such as loss and reordering can be added. Figure 2.1 shows the architecture of Dummynet.

The modules for handling packet processing (such as, enqueue and dequeue operations on packets) within Dummynet are called schedulers. Dummynet uses a timer and the schedulers get invoked at every tick.

Table 2.2: RD generated from a packet trace (N = 997).

k	FD[k]	RD[k] = FD[k] / N'
-7	1	0.001
-4	7	0.007
-2	110	0.110
-1	10	0.010
0	790	0.792
1	12	0.012
2	12	0.012
3	16	0.016
4	26	0.026
5	6	0.006
6	4	0.004
7	2	0.002
9	1	0.001

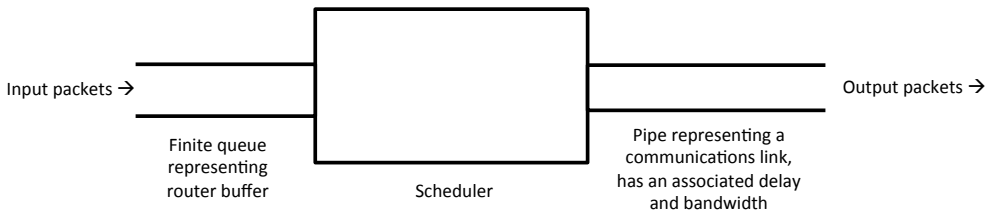

Figure 2.1: Binding between queues, scheduler and the corresponding pipe (which is made up of the delay and bandwidth queues in current implementation; more can be added)

Dummynet implements schedulers in a modular way so that if a different kind of processing (such as support for reordering or a different loss model) is needed then a new scheduler can be implemented with any required internal data structures, not just queues, and the scheduler name registered with Dummynet. The user can then specify the scheduler name that they want their packets to be processed by.

CHAPTER 3

ALGORITHM AND IMPLEMENTATION

This chapter describes my contributions. Specifically, Section 3.1 discusses the RD sequence regeneration algorithm, Section 3.2 briefly explains the implementation in Dummynet, and Section 3.3 describes the experimenter workflow.

3.1 RD Sequence Regeneration

For use in a network emulator, we need to be able to regenerate a sequence from a given RD. In practice, the user might only provide percentages of displacements where there might be rounding errors, that is, instead of something precise like Table 2.1, we may only get Table 3.1 as input.

My implementation provides users with a helper script to first convert these percentages to a suitable N' and FD[k] that fit the given percentages, the script can let the user specify an acceptable error threshold value and the minimum number of total packets (N) that should be present in the solution. The output from the script is an RD similar to the one shown in Table 2.1. The main sequence regeneration algorithm takes this RD as input and generates a sequence of numbers that fit that RD. The pseudocode is shown in Figures 3.1, 3.2, and 3.3.

As seen in the pseudocode, first a main graph consisting of a super-source, a super-sink, subsource(s), subsink(s), and bipartite graph(s) is constructed, this is modeled after max-flow. The super-source is connected to the subsource(s), the subsource(s) connects to the

Table 3.1: RD given for an unknown sequence.

k	Percentages = RD[k] * 100
-3	17%
-2	17%
0	17%
1	17%
2	32%

```
Input: A array R of RDs, such that R[i].displacement and R[i].count are the displacement and number of packets of the i'th RD
Output: Array of packet positions with reordering applied, and array of bipartite sub-graphs used in the next step

function ConstructGraph(R) {
    K = R.length               # total number of displacements
    N = sum(R[i=1 to k].count) # total number of packets

    G = Empty Graph
    bipartite = array of K bipartite graphs of length N each
    # assume no connections in bipartite graphs, we'll add those to G later

    Source = a vertex representing the super source
    Sink = a vertex representing the super sink

    s = array of K vertices, acting as sub-source
    t = array of N vertices, acting as sub-sinks

    # connect super source to sub-sources, one for each bipartite graph
    for each vertex s[i] in s
        G.add_edge(Source, s[i], capacity=R[i].count)

    # connect sub-source[i] to all left vertices of bipartite[i]
    for i=1 to K
        for j=1 to N
            G.add_edge(s[i], bipartite[i].left_vertex[j], capacity=1)

    # connect suitable left vertices of bipartites to suitable right vertices of bipartites
    # for each bipartite graph
    for i=1 to K
        displacement = R[i].displacement
        # for each left side vertex
        for j=1 to N
            k = j + displacement
            if k >= 1 and k <= N
                G.add_edge(bipartite[i].left_vertex[j], bipartite[i].right_vertex[k], capacity=1)

    # connect all right vertices of all bipartite graphs to the corresponding sub-sink
    for i=1 to N
        for j=1 to K
            G.add_edge(bipartite[j].right_vertex[i], t[i], capacity=1)

    # connect all sub-sinks to super sink
    for each vertex t[i] in t
        G.add_edge(t[i], Sink, capacity=1)

    return G, bipartite
}
```

Figure 3.1: Pseudocode for sequence regeneration from given RD: ConstructGraph().

left side of bipartite graph(s), the left side of the bipartite graph(s) connects to the right side of the bipartite graphs, the right side of the bipartite graph(s) is then connected to the subsink(s), and finally the subsink(s) connects to the super-sink. The capacities for all of the edges is one except for those that connect the super-source to the subsources, the capacities of these edges are equal to the number of packets. Figure 3.4 depicts how this graph generated by the algorithm for an example input RD (also shown in the figure) might look. Note that in the figure only one subsource is shown as being connected to the corresponding right side of the bipartite graph (graph 4); the remaining three subsources connect to the corresponding right side of the three remaining bipartite graphs (graphs 1, 2, 3) in exactly the same way, this was not shown to avoid cluttering the figure. While a subsource connects to all the vertices in only its corresponding bipartite graph's left side, a subsink, in contrast, connects to only one corresponding vertex in each of the bipartite graphs' right side. Only one subsink is shown as connected in the figure, the other three

```
# this is a depth-first search of the problem space
function SolveStep(solution, step) {
    # we reuse some variables defined in other functions here
    if step > N
        return true # problem solved!

    # for each step we loop over connected vertices of the current step's bipartite graph
    for i=1 to K # each bipartite graph
        if remaining_packets[i] == 0 # can we select more from this one?
            continue

        vartex = bipartite[i].left_vertices[step]
        if vertex.has_right_vertex
            right_vertex_index = vertex.right_vertex_index

            # we skip vertices whose target is already part of the solution from some previous step
            if solution[right_vertex_index] != null
                continue

            solution[right_vertex_index] = vertex
            remaining_packets[step]-- # decrement for this vertex
            if solve(solution, step+1)
                return true # solved!

            # this vertex didn't work out, reset values
            remaining_packets[step]++
            solution[right_vertex_index] = null

    return false # no solution found in this call
}
```

Figure 3.2: Pseudocode for sequence regeneration from given RD: SolveStep().

```
# input is G constructed by ConstructGraph()
function Solve(G)
{
    solution = array of size N
    # solution[1]=5 means packet at position 1 should get the packet from original position 5

    remaining_packets = array of size K, initialized such that remaining_packets[i]=R[i].count
    # we'll use this to keep track of the packets we are done with

    if SolveStep(solution, 1)
        return solution
    else
        return null # no solution!
}
```

Figure 3.3: Pseudocode for sequence regeneration from given RD: Solve().

remaining subsinks connect to the remaining vertices of all the bipartite graphs' right sides in exactly the same way.

Once this main graph to represent all possible permutations of displacements from the input RD table has been constructed, I then use graph search, a greedy search with backtracking, to get to the output packet sequence that satisfies the given RD. This sophisticated algorithm was needed because we have a constraint problem and the naive approach of just randomly picking displacement values from the input RD and using them to generate the output sequence would not work. The first iteration of my algorithm was a max-flow solver (an implementation of the Fork-Fulkerson algorithm). However, the problem turned out to have additional constraints not easily expressible in a simple max-flow graph. The graph

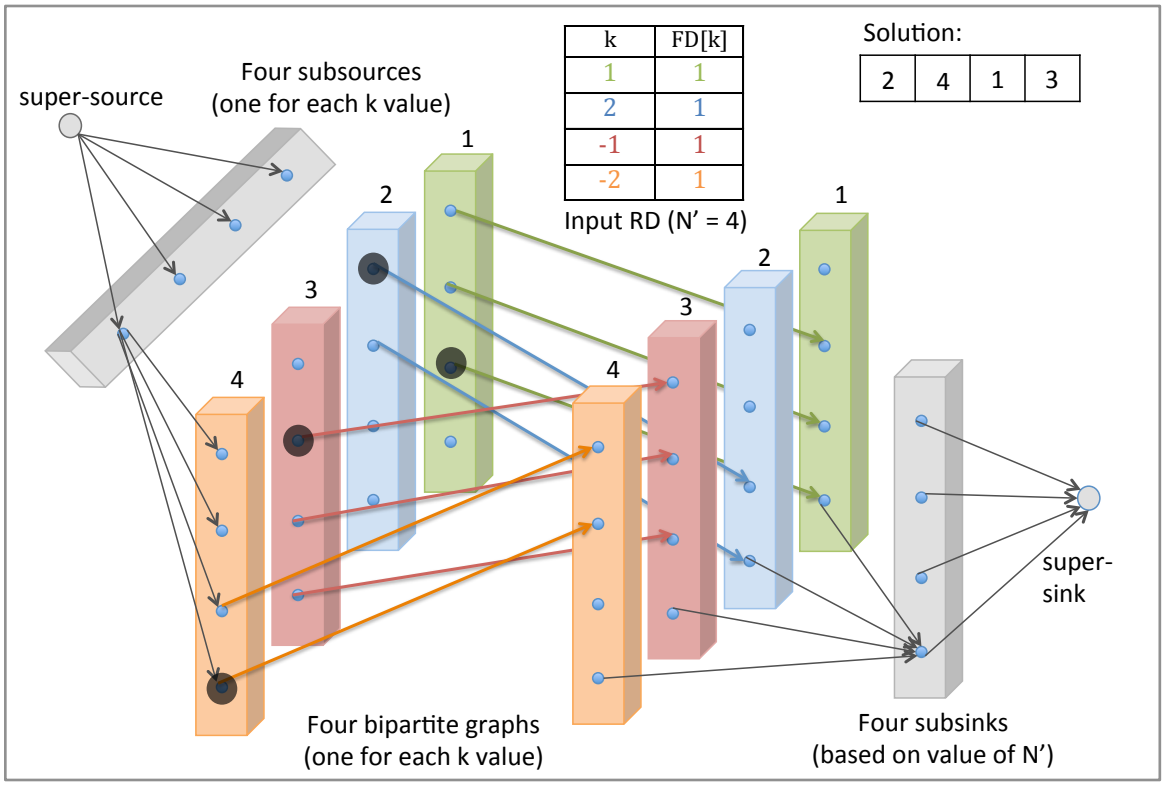

Figure 3.4: Graph generated by the RD sequence regeneration algorithm. The filled black circles are drawn over vertices that were selected for the solution.

is two-layered (the left and right sides of the bipartite graphs), and while our graph can ensure that only unique packets are selected on the right side, the left side has no such constraint, so you can end up with a solution where two packets could be reordered to the same position. While a constraint like this could theoretically be expressed in a graph, the combinatorial complexity explosion makes it impractical. So the final solver was written with max-flow as a model, but has this additional constraint and has been specialized for this specific problem.

Our algorithm is reduced to a depth-first selection in the bipartite graphs with these constraints:

- No two selected vertices may have the same source position (our new constraint).
- No two selected vertices may have the same destination position (the sub-sink capacity).

- Number of packets selected from a given bipartite graph must exactly match the packets input for that RD (the subsource capacity).

The new constraint actually helps prune the search tree and can allow a large reduction in the search space, leading to faster results for typical inputs. I have also inlined the sources and sinks parts of the graph in the calculations in the code.

Going back to our example graph in the figure, some points to note for the graph construction:

- The number of subsources created is determined by the number of k values in the input RD (in other words, the number of lines in the input RD table), which is 4 in this example, and so the number of subsources is also 4.
- The number of subsinks created is determined by the number of N' (the number of packets in the input RD). Since N' is 4 in this example, the number of subsinks is also 4.
- The number of bipartite graphs created is determined by the number of k values in the input RD. Since the input RD in this example has 4 k values, so the number of bipartite graphs is also 4. So, basically, graph 1 in the figure is representing k=1, graph 2 is representing k=2, graph 3 is representing k=-1, and finally graph 4 is representing k=-2.
- The height of the left or right side of the bipartite graph is determined by the value of N'. Since N' for the example input RD is 4, so the height of each side of the bipartite graph is also 4.

Since N' is equal to 4, so the output solution array will also be of size equal to 4; currently the array is empty and looks like this [, , ,]. Now, to get to the solution for this example, I look at the left sides of the bipartite graphs and work in horizontal layers. So, I look at the topmost layer of vertices then the second layer then the third and finally the fourth horizontal layer of vertices in the left sides of the bipartite graphs. What I am really looking for here are the connected vertices (connected means the vertex is connected using an edge to the right side of the bipartite graphs). All of these connected ones are possible solutions from which I have to choose the right ones, if later on it turns out the solution was not the correct choice then I backtrack and choose another solution. Also, as mentioned

earlier, each bipartite graph represents a k value from the input RD, so I can only select a vertex as a possible solution from a bipartite graph if it is a connected vertex and if the corresponding FD[k] value for the bipartite graph which contains that connected vertex is nonzero.

This is how I will do the selection for the given RD: I start with the first horizontal layer of vertices (the top-most layer) of the left side of the bipartite graphs. I first look at the orange graph (graph 4), I see that its vertex is not connected which means that this vertex is not a solution so I move over to the red graph (graph 3), its vertex is also not connected, I move over to the blue graph (graph 2), now its vertex is connected and the corresponding FD[k] value for the blue graph, which FD[2] is also nonzero, so this means that this vertex can be selected as a solution. The connection of this top vertex in the left-side of the blue bipartite graph is to the third vertex in the right side of the bipartite graph so this connection is telling me that packet number 1 should be placed in position number 3 in the output sequence. So, I put packet 1 in position 3 in the output array. The output array now looks like this [, , 1 ,]. Now, because I have used the blue graph once, I decrement the FD[2] value since, as mentioned earlier, the blue graph represents k=2. So, now FD[2] = 0, this value of zero will mean that I will no longer be able to select a vertex from the blue graph even if the vertex is connected. This is a constraint in the algorithm.

Now, I move on to the second horizontal layer of vertices, in a similar way as used above I select a connected vertex from the red graph (graph 3), decrement FD[-1], and update the output array: [2 , , 1 ,].

Next, I move on to the third horizontal layer of vertices, and select the connected vertex from the green graph (graph 1), decrement FD[1], and update output array as follows: [2 , , 1 , 3]

Finally, I move to the fourth and last layer of horizontal vertices, select the connected vertex from the orange graph (graph 4), decrement FD[-2], and update the solution array: [2 , 4 , 1 , 3].

As can be seen the final solution array contains a sequence of packet numbers that have exactly the same RD as the input RD. Thus, the sequence regeneration algorithm correctly worked.

To sum it up, in the algorithm, once the main graph is constructed and connected, I try to find suitable flows from the super-source to the super-sink. This means that I take one input packet (at the super-source), choose a displacement for it (using suitable connected vertices from the left side to the right-side of bipartite graphs; this is a greedy search with

backtracking) which lets me find which position it should be placed in, in the final output sequence.

The algorithm is a strict solver. It will find a solution if at all possible, and will fail if it is not possible, it does not output approximate solutions. It uses various optimizations to quickly get a result, but will exhaustively scan the problem space. However, this occurs only when the reordering is extremely high, there are very little packets that are in order, and the reorderings in the sequence are heavily overlapping. Reordering events in a packet sequence can either occur independently or they can interact with each other, overlapping with or embedding within other events [15]. The higher this amount of interaction, the more complex it becomes for the sequence regeneration algorithm to search the right permutation that would fit the input RD. This is rare in real traffic and we expect to see only moderate amounts of reordering. Also, as the algorithm is run offline and the users have the option of using it repeatedly to regenerate large sequences, the worst-case running time would not occur in practice.

The space complexity of the problem is O(N*K). The time complexity of the problem is fairly hard to pin down. The complexity varies wildly depending on the RD. While you could treat this as a permutation problem (O(N!/(N-K)!)), or as a max-flow problem ($O(VE^2)$, etc.), that does not take into account the complexity reduction our constraints allow.

3.2 Integration into Dummynet

My RD sequence regeneration algorithm is made part of the userland, it generates output files containing the reordered sequence which can then be input to my enhanced version of Dummynet along with the other usual parameters, such as loss and delay. This also implies that if the user wishes to use some other reordering metric such as n-reordering, or some other custom metric then the output from that metric can also be input in place of RD, as long as the output is in the form of a reordered packet sequence.

At the kernel side, I implemented and registered with Dummynet my new scheduler with reordering functionality. So, if the user has specified the reordering scheduler as part of the input, then instead of invoking the default scheduler to process incoming packets, Dummynet instead uses this new scheduler that sends the incoming packets out in a reordered fashion based on the reordered sequence that came as input to Dummynet. The scheduler basically uses a buffer to hold onto incoming packets until as many packets as are needed to accomplish the desired amount of reordering have arrived. So, for example,

if a packet is supposed to be reordered as 2 places late, I store it in the buffer and when 2 other packets have arrived that can go before it, I then send the stored packet out. The highest number of packets that will be buffered by the scheduler is limited by the highest displacement value in the input RD. The highest displacement in a given RD is in turn limited by the displacement threshold (DT) of the RD calculation algorithm, as mentioned earlier. It is usually small, such as, somewhere between 5 to 25.

3.3 Experimenter Workflow

To show where the algorithm fits, the following list of steps shows what the workflow looks like. The output from each step is used as input in the next.

1. Either take a packet trace and calculate its RD, or take an RD from the literature.
2. Repeat the previous step zero or more times as desired to generate a set containing one or more RDs.
3. Take the output set from step 2 and run the RD sequence regeneration algorithm over each RD in the set.
4. Take the output sequence(s) from step 3 and input to Dummynet implementation along with any other required traffic shaping parameters (such as delay or loss values).
5. Run the Dummynet emulator to finally generate the experimental traffic shaped according to the parameters provided in the previous step.

The workflow is shown in Figure 3.5.

Note that, as the above workflow shows, the RD algorithms are run offline and also the number of packets used to generate RDs in step 1 is independent of the number of packets in the experimental traffic that Dummynet is made to generate in step 5. So, if Dummynet is made to run experimental traffic containing a million packets, the user only needs to run the regeneration algorithm in step 3 once over, say, a 1000 packets and then in step 4 Dummynet can use that 1000-packet output from step 3 repeatedly to generate the million packets. The following example illustrates the workflow; the numbers used for RD are unrealistic but have been chosen to illustrate the point. In a real experiment, an RD would be expected to have been calculated over 500-1000 packets.

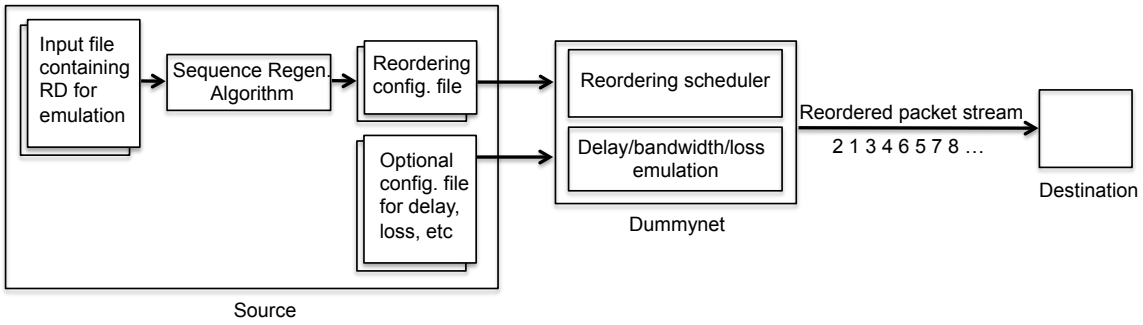

Figure 3.5: Experimenter workflow.

Let us say we need to generate experimental traffic in Dummynet containing a million packets. A possible workflow is as follows.

1. We take a small packet trace containing 4 packets and calculate its RD. The output is shown in Table 3.2.
2. We decide not to repeat step 1 and so our set of RDs contains only one RD.
3. We run RD sequence regeneration algorithm over our set and the resulting sequence is, resulting sequence is,resulting sequence is,resulting sequence

Regenerated sequence:

2	4	1	3

4. We take this regenerated sequence and input it to Dummynet.
5. We run the Dummynet emulator to generate experimental traffic containing a million packets that are reordered according to the regenerated sequence from step 4. The regenerated sequence is basically a pattern that tells Dummynet that for every set of 4 experimental traffic packets (from the total million it has to generate), take packet number 2 and put it in position 1; take packet number 4 and put it in position 2; take packet number 1 and put in position 3; and finally, take packet number 3 and put it in position 4. So, this is how an RD calculated over only 4 packets can be used repeatedly by Dummynet to generate a million reordered packets.

This chapter provided the description of the algorithm and its implementation as well as the experimenter workflow, the evaluation of the algorithm's scalability, implementation's correctness and its effect on emulation running time is described in the next chapter.

Table 3.2: Example RD (N = 4).

k	FD[k]	RD[k] = FD[k] / N'
-1	1	0.25
-2	1	0.25
1	1	0.25
2	1	0.25

CHAPTER 4

EVALUATION

In this chapter I describe the evaluation of my algorithm and implementation and thus provide evidence to support the claim that I have proven my thesis true by having Dummynet support emulation of reordering. In the first two evaluations, I show that my algorithm is scalable and the implementation works correctly in regenerating a reordered packet sequence from the user input RD that is equivalent to the real traffic sequence over which reordering was originally calculated by the user. Finally, a datapath evaluation is done to show that my modifications to the original Dummynet do not cause any unnecessary increase in the emulation running time.

The chapter is organized as follows. Section 4.1 briefly explains the general evaluation plan I followed. Section 4.2 then describes my evaluation for correctness using real network traces in which I, side-by-side, compare the RDs for real network traces and my regenerated packet traces. In Section 4.3, I describe my second evaluation which further tests my algorithm for scalability using controlled simulation. Finally, in Section 4.4, I report the results from the datapath evaluation.

4.1 Evaluation Plan Followed

For showing correctness, the main evaluation plan I used was as follows:

1. Take real packet traces.
2. Calculate RD over them.
3. Feed those RDs into my implementation and generate packet sequences.
4. Calculate RD on the resulting packet sequences, and show how they compare to the RDs from step 2. If they are equal or very close to the ones calculated in Step 2, then my implementation is demonstrated to work correctly.

For scalability evaluation, my plan was as follows:

1. Generate traces using software. The factors to be varied include number of packets and amount of reordering. The properties of the traces that will be kept constant will be at realistic values (realistic values were set based on observations from real network traces).
2. Calculate RD over the traces and measure runtimes.
3. Plot the resulting runtimes to show the algorithm's scalability.

For the datapath evaluation, my plan was to conduct two kinds of experiments. In the first one:

1. Run traffic through original as well as modified Dummynet with reordering turned off and note the maximum interarrival time observed for both.
2. Conduct a statistical test to determine if any significant difference exists between the times observed.

In the second experiment:

1. Run traffic through modified Dummynet with reordering turned on, note the maximum interarrival time observed. The expected time is equal to H*M, where H is the highest displacement in the RD used as input for reordering and M is the maximum interarrival time observed for modified Dummynet when reordering was turned off in the first experiment.
2. Conduct a statistical test to determine if any significant difference exists between the expected and observed times.

4.2 Real Network Traces

I ran the RD calculation algorithm for network traces consisting of long-lived connections from a host in Colorado to multiple networks located in different continents and the results were collected hour-by-hour [2]. I fed these into the RD regeneration algorithm to regenerate the packet sequences. Then, the RD calculation algorithm was run over the regenerated packet sequences and the output set of RDs were compared with the original set of RDs for the traces. The regeneration algorithm had worked correctly in all cases as no difference was found among the two sets by the script that used diff for comparison. The output RDs from three sets of measurements along with the corresponding RDs calculated over the regenerated sequence are shown in Table 4.1, Table 4.2, and Table 4.3. Also mentioned

Table 4.1: RDs for server in India (N = 136768, percentage of reordering = 0.11%, sequence regeneration runtime = 0.126s).

Original RD		Regenerated Sequence RD	
k	FD[k]	k	FD[k]
-5	8	-5	8
-4	5	-4	5
-3	10	-3	10
-2	30	-2	30
-1	16	-1	16
0	136626	0	136626
1	32	1	32
2	16	2	16
3	9	3	9
4	5	4	5
5	11	5	11

Table 4.2: RD for server in Cape Town (N = 138275, percentage of reordering = 0.03%, sequence regeneration runtime = 0.057s).

Original RD		Regenerated Sequence RD	
k	FD[k]	k	FD[k]
-5	2	-5	2
-4	3	-4	3
-3	3	-3	3
-2	8	-2	8
-1	4	-1	4
0	138239	0	138239
1	4	1	4
2	5	2	5
3	3	3	3
4	2	4	2
5	4	5	4

is the time taken by the sequence regeneration algorithm. As seen from these three tables both RDs completely match. Only the frequency counts FD[k] of the displacements k are shown, RD[k] is omitted to save space.

4.3 Software-Generated Network Traces

The previous evaluation focused on showing correctness, this one focuses on scalability which is not easy to show using real traces, thus, software-generated controlled ones are required. For a moderate amount of reordering as is common in real networks, the evaluation tests discussed in the previous section showed that the sequence regeneration algorithm

Table 4.3: RD for server in Pakistan (N = 136107, percentage of reordering = 0.51%, sequence regeneration runtime = 122.730s).

Original RD		Regenerated Sequence RD	
k	FD[k]	k	FD[k]
-5	4	-5	4
-4	37	-4	37
-3	29	-3	29
-2	61	-2	61
-1	343	-1	343
0	135410	0	135410
1	26	1	26
2	52	2	52
3	47	3	47
4	41	4	41
5	57	5	57

regenerates packet sequences within reasonable time. However, the running time becomes high as the amount of reordering events and/or interaction between reordering events in the sequence increases. So, in this section I want to show the effect that increasing the number of packets and amount of reordering will have on the run time of the algorithm. I wrote a simple trace generator to test the limits of the algorithm. The generator used the Fisher-Yates shuffle algorithm [8] to randomly shuffle the input array containing the packet sequence. It allowed the user to specify the value of N (total number of packets to generate in the sequence), the percentage of packets that needed to be left in-order, and the maximum value of k (i.e., the maximum displacement a packet in the sequence could be displaced by). In all experiments, based on observations from real network traces, I set k = 5.

I conducted two sets of experiments. In the first set, I varied the amount of reordering while keeping the number of packets, N, constant to 1000 packets, which is a reasonable number to expect in a real workflow of common experiments. The output sequences from the generator were then fed to my implementation and running times were calculated. The results are shown in Figure 4.1.

In the second set of experiments, I varied N while keeping the amount of reordering constant at an RD sampled from the real network traces whose results were shown in the previous section. Again, the output sequences were used as input for sequence regeneration algorithm and running times were observed. The results of this set are shown in Figure 4.2.

To show that the algorithm works really well for realistic inputs, I also included in

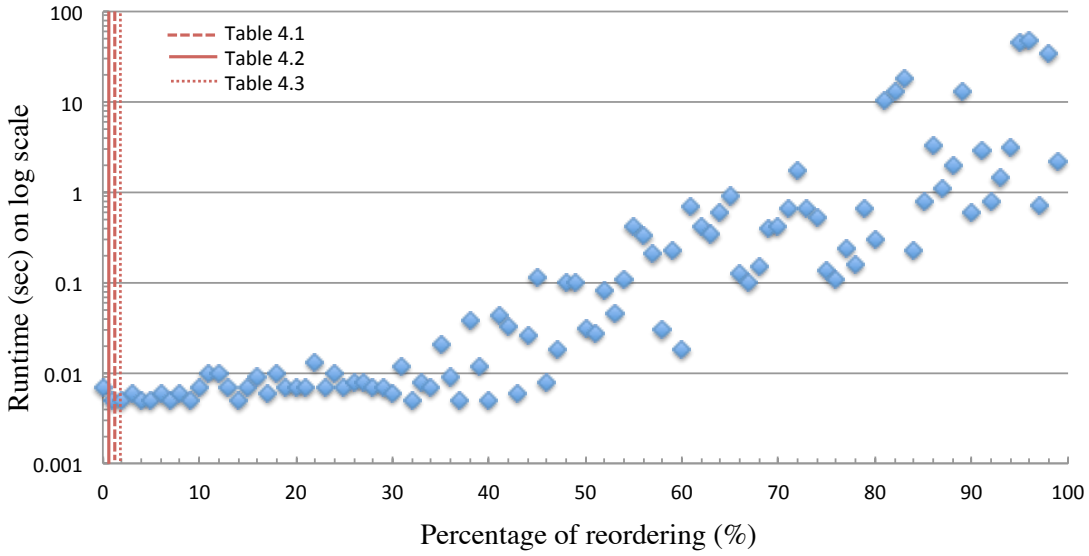

Figure 4.1: Effect of amount of reordering events on algorithm runtime. Number of packets kept constant. Vertical red lines highlight position of data points from real traces.

both graphs, as three data points (the three vertical lines in the graphs), results from the real traces presented in the previous section in Tables 4.1, 4.2, and 4.3. Of course, it is not unexpected to see experimenters trying out higher values of algorithm parameters as compared to what we see in real traces, for example, to test their protocols for some network that causes a relatively higher amount of packet reordering to occur. In such cases, the algorithm would still work pretty well (also, remember that it runs offline), as can be seen from the upper bound of running times in the two graphs. For both graphs, the runtimes are subsecond for the realistic cases, and always in the low tens of seconds.

4.4 Datapath Evaluation

This evaluation's aim was to show if the modifications I did in the original Dummynet in order to support reordering increased the running time of the emulation more than was necessary. To show this, I ran an experiment 20 times which consisted of using ping with the flood option turned on to send 500 packets through the original Dummynet as well as the modified Dummynet (with reordering turned off) and noting the maximum interarrival time observed in both cases. The number of runs for the experiment was determined using a statistical test to get 95% confidence level, the result from the test was rounded off to the nearest tenth to get the value to be used as the number of runs. The max interarrival times observed from all the runs of the experiment were averaged using mean and are

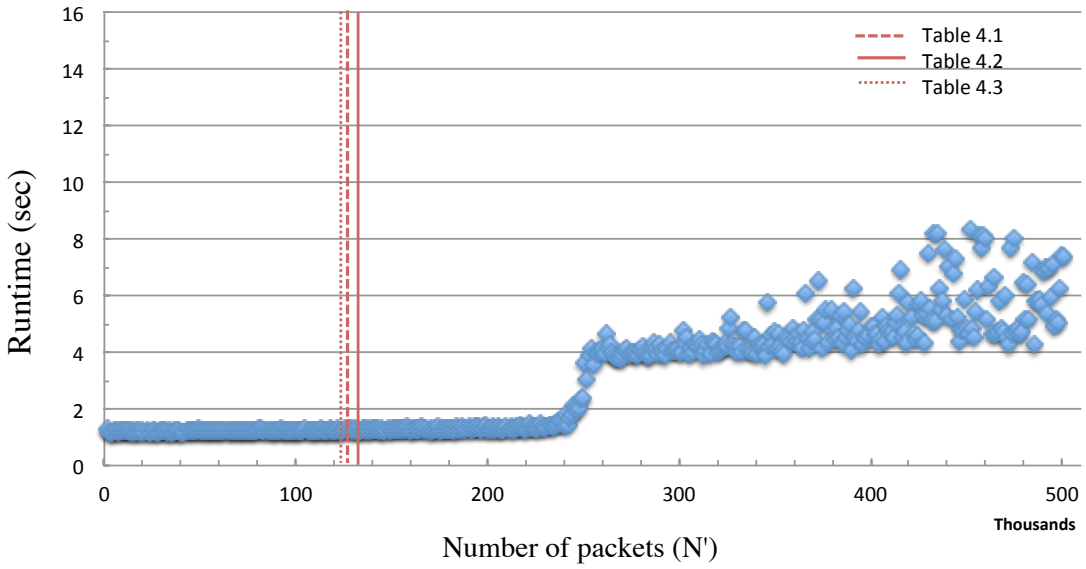

Figure 4.2: Effect of number of packets on algorithm runtime. Amount of reordering kept constant. Vertical red lines highlight position of data points from real traces.

reported in Table 4.4. The modified version of Dummynet is referred to as MyDummynet. Statistical tests showed with 95% confidence level no significant difference between the times from the two configurations, and so it was concluded that if reordering is turned off then MyDummynet does not add any unnecessary processing in the datapath that would result in an increase in the running time of the emulation.

I did another experiment with all configurations the same as earlier except that this time reordering was turned on in MyDummynet. This was to see if any unnecessary increase in running time was being caused by my modifications to the original Dummynet. A simple RD was used for reordering, it is shown in Table 4.5. The largest displacement in the RD is 19. This means that the max interarrival time we expected to see when traffic is run through MyDummynet should be very close to (19*2.51), which is 47.69 msec. The value 2.51 is the max interarrival time we observed when reordering was turned off in MyDummynet, it is reported in Table 4.4. So, the expected as well as the observed times for this experiment with reordering turned on are reported in Table 4.6. Statistical tests showed no significant difference with 95% confidence, hence it was concluded that the datapath for MyDummynet when reordering was turned on did not introduce any unnecessary processing that would have increased the running time for the emulation more than was necessary for applying the reordering.

Table 4.4: Maximum interarrival time observed on original Dummynet and on MyDummynet (Dummynet after my modifications).

Configuration	**Max interarrival time (msec)**
Original Dummynet	2.48
MyDummynet (Reordering off)	2.51

Table 4.5: RD used for datapath evaluation (N = 20).

k	FD[k]	RD[k] = FD[k] / N'
-19	1	0.05
0	18	0.90
19	1	0.05

Table 4.6: The expected and the observed maximum interarrival time on MyDummynet with reordering turned on.

Configuration	**Max interarrival time (msec)**	
MyDummynet (Reordering on)	Expected	47.69
	Observed	48.07

CHAPTER 5

CONCLUSION

The focus of my thesis was increasing the realism of network emulation. I argued and provided evidence that packet reordering is a prevalent network phenomenon that affects performance of both TCP and UDP applications and hence deserves attention, including research into models and tools to diagnose and understand it, just as is given to other phenomena, such as packet loss or delay. So, since it is a phenomenon that cannot be ignored and emulation would not be realistic if the emulator did not also have support for reordering, my thesis made this first contribution of implementing its support within Dummynet. The second contribution was an offline algorithm, a max-flow solver, for sequence regeneration from a sophisticated reordering metric called RD. The output from this algorithm was then used as input to my implementation within Dummynet. I used real and software-generated traces to show that the algorithm is scalable and the implementation works correctly. I also did a datapath evaluation to show that my modifications to Dummynet do not result in any unnecessary increase in emulation running time.

Currently, my sequence regeneration algorithm expects the input RD to be properly formed, with negative displacements balancing out positive ones completely. Thus, it is left to the users to manually adjust the RD based on their knowledge about the network they want to emulate. So, as future work, I plan to include support for this adjustment within my implementation.

REFERENCES

[1] Dummynet references from Citeseer. *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.2969*, 2013.

[2] Packet reordering trace. *http://www.cnrl.colostate.edu/Projects/PacketReordering/Trace/packet_reordering_trace.htm*, 2013.

[3] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser. Packet reordering metrics. *RFC 4737*, 2006.

[4] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser. Packet reordering metrics. *IETF Internet-standard: RFC4737*, 2006.

[5] A. P. Jayasumana, N. M. Piratla, T. Banka, A. A. Bare, R. Whitner. Improved packet reordering metrics. *RFC 5236*, 2008.

[6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 2003.

[7] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *Proc. OSDI Symposium*, 2002.

[8] P. E. Black. Fisher-Yates shuffle. *Dictionary of Algorithms and Data Structures [online], US National Institute of Standards and Technology*, 2005.

[9] J. C. R. Bennett, C. Patridge, and N. Shectman. Packet reordering in not pathological network behavior. *IEEE/ACM Trans. Netw.*, 1999.

[10] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving accuracy in end-to-end packet loss measurement. *ACM SIGCOMM Computer Communication Review*, 2005.

[11] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev. 40*, 2010.

[12] M. Laor and L. Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, 2002.

[13] M. Lelarge. Packet reordering in networks with heavy-tailed delays. *Mathematical Methods of Operations Research*, 2008.

[14] N. M. Piratla, A. P. Jayasumana and A. A. Bare. RD: A formal, comprehensive metric for packet reordering. *Proc. IFIP Networking Conference*, 2005.

[15] N. M. Piratla, A. P. Jayasumana and T. Banka. On reorder density and its application to characterization of packet reordering. *Proc. 30th IEEE Local Computer Networks (LCN) Conference*, 2005.

[16] N. M. Piratla and A. P. Jayasumana. Reordering of packets due to multipath forwarding – An analysis. *Proc. IEEE International Conference on Communications*, 2006.

[17] N. M. Piratla and A. P. Jayasumana. Metrics for packet reordering – A comparative analysis. *International Journal of Communication Systems*, 2008.

[18] V. Paxson. End-to-end Internet packet dynamics. *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1997.

[19] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Measurement and classification of out-of-sequence packets in a tier-1 IP backbone. *IEEE/ACM Transactions on Networking (ToN)*, 2007.

[20] T. Banka, A. A. Bare and A. P. Jayasumana. Metrics for degree of reordering in packet sequences. *Proc. 27th IEEE Conference on Local Computer Networks*, 2002.