Realistic Packet Reordering for Network Emulation and Simulation

Aisha Syed
University of Utah
aisha.syed@utah.edu

Robert Ricci
University of Utah
ricci@cs.utah.edu

ABSTRACT

We present an algorithm that takes measurements of the packet Reorder Density (RD) metric and generates reordering sequences. These sequences can be used by a simulator or emulator to precisely and repeatably reorder packets in a way that recreates the original RD. We show that our algorithm is efficient for a range of realistic reordering scenarios, and present an extension to the Dummynet emulator that uses makes use of it.

CCS Concepts

•**Networks** → **Network performance evaluation; Network performance modeling; Network experimentation;** *Network dynamics;* Network simulations;

Keywords

Emulation; simulation; packet reordering; RD metric; realistic traffic shaping; Dummynet

1. INTRODUCTION

Packet reordering is a phenomenon just as fundamental to Internet traffic as packet loss or delay [13, 11, 6, 10, 12, 14]. Despite studies demonstrating its prevalence and its ability to significantly degrade application performance [10], reordering is less well-studied than loss or latency. Bennett, et al. [6] found that reordering is not just pathological behavior: 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 reduce performance overall. Reordering also has an adverse effect on delay-based real-time UDP applications such as video conferencing [12, 9]. This effect has become especially important as streaming media, VoIP and IPTV are becoming increasingly prevalent on the Internet. A study [8] of reordering in a backbone link found that even a small amount of reordering coupled with packet loss can cause significant degradation in link utilization and thus application throughput.

We argue that it is critical to be able to incorporate precise, repeatable reordering into the emulations and simulations that we use to evaluate networked systems. As Piratla, et al. [11] argue, measuring and characterizing reordering and devising models for understanding it can help us deal with it in scalable ways. Even when reordering itself is not the primary subject of study, ignoring it during traffic shaping in an emulator or simulator results in experimental traffic that is not representative of real network traffic. As is the case with latency and available capacity, emulation is most realistic when it faithfully re-creates measurements taken from real networks.

There are two basic approaches to emulating (or simulating) network effects: emulating the underlying *cause* in the network, or emulating the *effects* directly. When re-creating behavior measured from real networks, it is often preferable to emulate *effects*, because they can be measured from the edge of the network. In contrast, most *causes* are not directly observable from the edge, and it can be difficult or impossible to "reverse engineer" from measured conditions a network that produces them precisely. So, while there are existing methods to emulate re-ordering *causes* (for example, the multipath feature in Dummynet [7]), our goal is to take end-to-end measurements of re-ordering and re-create the *effects* directly in a simulator or emulator. We also aim for the ability to accurately apply the same reordering pattern to repeated experiments, and to be able to make precise adjustments to the degree of reordering in order to perform sensitivity analyses or parameter-space explorations.

Reorder Density (RD) [1, 10, 12] is a comprehensive

CoNEXT'15 December 01 – 04, 2015, Heidelberg, Germany

 ISBN 978-1-4503-3412-9/15/12...$15.00

DOI: 10.1145/2716281.2836110

metric for quantifying reordering, and it has been well-studied. RD is defined in the forward direction: one can take a sequence or trace of packets and calculate its RD. What is missing to drive simulations or emulations is the reverse direction: given a measured RD value, creating a re-ordering sequence that produces the metric. This regenerated packet sequence can be used by a traffic shaper to reorder traffic within an experiment. Devising such an algorithm for RD is the primary contribution of this paper. Our algorithm is not specific to any particular emulator or simulator. As a second contribution, we have implemented an extension to the Dummynet emulator [7] that reorders traffic according to the sequences produced by our algorithm. Our extension to Dummynet is agnostic to the source of the reordering sequences that it uses; while we built it for use with our RD-based sequences, it can accept reorder sequences that come directly from traces, are derived from other reordering metrics [2], or are entirely artificial.

In Section 2, we cover related work and the RD metric. Section 3 describes our RD sequence regeneration algorithm, and Section 4 describes our Dummynet extension and how it uses the output from our algorithm to emulate reordering. Section 5 presents our results from the evaluations for both the algorithm and the Dummynet extension, and we conclude in Section 6.

2. BACKGROUND & RELATED WORK

There are many ways to measure reordering in a packet stream, including the percentage of reordered packets, n-reordering, and reordering extent, all of which are standardized by the IETF [2]). Reorder Density (RD) [12] is a particularly useful metric, because it provides information about both early and late reordered packets. Its calculation has low space and time complexities ($O(1)$ and $O(N)$, respectively) and is not affected by other network phenomena such as loss or duplication. Also, RD can provide useful information to applications. For example, it can help in TCP flow control by providing estimates of the buffer size necessary to recover from reordering, and can also help in network diagnosis by hinting about the possible causes of reordering, etc. [10]

RD Calculation. RD captures the amount of reordering by measuring the displacements of packets from their original positions [1]. Let us take an example (shown in Table 1) in which a sequence of $N = 6$ packets is sent in-order, and arrives at the receiver in this sequence: `[4 1 5 2 3 6]`.

The receiver assigns a receive index RI to each packet according to the order of its arrival. In our example, packet 4 arrives first and gets $RI = 1$, packet 1 comes next and gets $RI = 2$, and so on. Then displacement D of a received packet is defined as the difference between its RI and the order in which it was sent (i), i.e., the displacement of packet i is $RI[i] - i$. Thus, a negative displacement indicates an early packet and a positive displacement a late packet. A displacement of zero indicates the packet arrived in order.

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

Table 1: An example RD calculation.

k	$FD[k]$	RD[k] = $FD[k]/N'$	Pct. = $RD[k] \cdot 100$
-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%

Table 2: RD histogram for [4 1 5 2 3 6]; N'=6

The RD calculation algorithm also defines a *displacement threshold* DT such that a packet is considered lost if it does not arrive within the window defined by DT, and similarly, a packet is considered duplicate and discarded if another packet with the same sequence number has already been received within the current 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; for example, for VoIP applications, it can be selected based on the maximum duration the application waits for a packet's arrival before considering it lost.

Finally, a displacement frequency $FD[k]$ is calculated as the number of received packets having a displacement of k, for k in range $[-DT, DT]$. The reorder density RD is then 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 ($N' = \Sigma(FD[k])$). For our example received sequence, the RD is shown in Table 2.

3. RD SEQUENCE REGENERATION

To use RD in a network emulator or simulator, we need to be able to regenerate an ordering sequence from a given RD. The input to our algorithm is an RD table similar to the one shown in Table 2, consisting of the k and $FD[k]$ columns.[1] We run a preprocessing step that calculates the number of packets, $\Sigma(FD[k]) = N'$ and generates an *input sequence* (IN) consisting of consecutive numbers $1..N'$. This sequence is then permuted by the main algorithm. The output is a *reordering sequence* (OP), which is an array indexed by the packets' locations in the input array, with the content of each cell indicating the packet's sequence in the reordered

[1]The user can also provide percentages for $FD[k]$ which can be converted to precise packet counts by our preprocessor.

stream. The algorithm we present is modeled after the max-flow problem [5] but is specialized for our problem due to the presence of the following three constraints:

A. Each packet must be displaced (reordered) by exactly one displacement—that is, OP is a permutation of IN (which it itself the sequence $1..N'$).

B. Two packets cannot be displaced such that they end up in the same position in the output array. If we denote the displacement of packet i as D_i, then:

$$\forall i, j \in \{1..N'\} | i \neq j \quad i + D_i \neq j + D_j$$

C. The number of packets displaced by displacement k should be exactly equal to $FD[k]$. That is, for every k, there is a subset of IN called IN_k that consists of all packets displaced by k so that the following holds true:

$$|IN_k| = FD[k]$$

The complexity of these constraints precludes a simple method for picking a legal permutation. They do, however, help prune the space of legal permutations, enabling quick searches through it for typical inputs.

Algorithm. To find a reordering sequence OP, we construct a graph representing the solution space and conduct a search through it:

1. For each unique displacement k such that $FD[k] > 0$, construct a bipartite graph with N' vertices on each side. Connect vertex j on the left to vertex $j + k$ on the right, for $j <= N' - k$. The left side represents packets' positions in the input stream, the right side their order in the output stream after being displaced by k positions.
2. Construct a sub-source vertex s_k for each such displacement k, and connect it via a *capacity* $= 1$ edge to each left side vertex of the corresponding bipartite graph. The limited capacity of the edge helps to enforce constraint B *within* the set of packets displaced by k positions.
3. So far, for each k, we have one bipartite graph plus its corresponding sub-source s_k. Now construct a super-source vertex S and connect it to all s_k. Each edge has *capacity* $= FD[k]$ for the corresponding k, enforcing constraint C.
4. Construct a sub-sink t_j for each *j=1..N'*, representing packets' placement in the reordered stream. From each bipartite graph, connect the right side vertex j to t_j.
5. Connect all the t_j to a super-sink T. Each edge has *capacity* $= 1$, enforcing both constraint B. Combined with the capacities on super-source S, this also enforces constraint A.

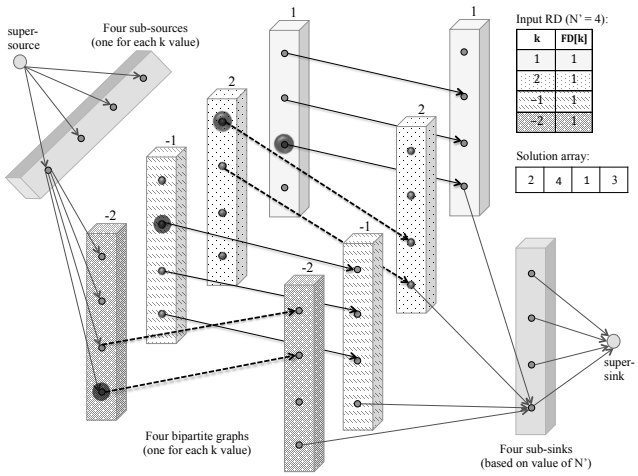

Figure 1: Graph generated by the RD sequence regeneration algorithm. Filled black circles show vertices selected for the solution.

6. Find a set of N' unit flows through the graph from S to T without violating capacities on any edges.

The graph constructed in steps 1–5 represents all possible permutations of displacements from the input RD table minus the permutations not possible due to the application of our constraints. In step 6, we use a greedy graph search with backtracking to get to a solution packet sequence that satisfies the given RD. This is accomplished by selecting suitable vertices from the left side of bipartite graphs (the source side) that connect to suitable vertices on the right side of bipartite graphs (the sink side). The position in the left side gives us the packet's position in the input stream, and the position on the right side gives us its position in the output stream. The construction of the graph ensures that all input and output positions are used exactly once, and that each displacement k is used $FD[k]$ times.

Complete pseudocode is in the Appendix and code can be found at [3]. Additional discussion can also be found in [15].

Example. Figure 1 depicts a graph constructed using this algorithm for the RD shown in the top right of the figure. To reduce clutter, only one sub-source s_k is shown as being connected to the corresponding left side of the bipartite graph; the remaining three s_k are not shown. Similarly, only one sub-sink t_j is shown as connected.

As dictated by the algorithm, the number of sub-sinks and the height of the bipartite graphs are both $N' = 4$. The number of sub-sources, and number of bipartite graphs is each equal to $|K| = 4$ where K is the set of all unique displacements k. The bipartite graph labeled "1" in the figure represents $k = 1$, the graph labeled "2" represents $k = 2$, and so on. The two sides of these

k	FD[k]
1	1
2	0
-1	1
-2	1

(a)

k	FD[k]
1	1
2	0
-1	0
-2	1

(b)

k	FD[k]
1	0
2	0
-1	0
-2	1

(c)

k	FD[k]
1	0
2	0
-1	0
-2	0

(d)

Table 3: RD table updates for our example

graphs are connected such that a left vertex at position j connects to the right vertex at position $j + k$. A bipartite graph representing displacement k=0, connects "straight across." For a bipartite graph representing $k = 2$, position 1 on the left connects to position 3 on the right, position 2 on the left connects to 4 on the right, etc. If a connection would under- or over-flow either the left or the right side, no edge is made.

Now, we walk through the graph search to show how the solution is found. In our example, $N' = 4$, so output solution array OP will also be of size 4. Initially, the array is empty: `[NIL, NIL, NIL, NIL]`.

Then vertex selection proceeds as follows: we start with the first horizontal layer of vertices (representing the first packet in the input stream) of left side of the bipartite graphs. We first look at the vertex from graph labeled "-2": since this vertex is not connected to the right side, it cannot be a solution and we move on to the graph labeled "-1". Its vertex is also not connected, so we move to the graph labeled "2", which is connected, and the corresponding $FD[k]$ value for $k = 2$ is nonzero, meaning that this vertex can be selected as part of the solution. The edge from this vertex connects to the third vertex in the right side of bipartite graph; this tells us that packet number 1 should be placed in position number 3 in the output sequence. We make this placement: `[NIL, NIL, 1, NIL]`.

Because we have used the graph labeled "2" once, we decrement the $FD[2]$ count (Table 3(a)), representing the capacity constraint between S and s_2. Now, $FD[2] = 0$—this zero value means that we can no longer select a vertex from the graph labeled "2" even if its vertex is connected, according to constraint C.

Now, we move on to the second horizontal layer of vertices. In a similar manner, we select a connected vertex from the graph labeled "-1", decrement $FD[-1]$ (Table 3(b)), and update the output array: `[2, NIL, 1, NIL]`. On the third layer, we select a vertex from graph "1" (Table 3(c)), giving us `[2, NIL, 1, 3]`. Note that we selected from graph "1" even though graph "-2" had a connected vertex: this is because the selection from "-2" was not possible due to constraint B. Finally, we move to the fourth and last layer of horizontal vertices, select the connected vertex from the graph labeled "-2", decrement $FD[-2]$ (Table 3(d)), and update the solution array: `[2, 4, 1, 3]`.

At this point, all $FD[k]$ values are zero and the solution array is completely filled. If, at any point, we had been left with no legal options, we would have simply backtracked "up" a level in the bipartite graphs and selected a different option. Note that, while our simple example did not have any in-order packets ($k = 0$), this is just an artifact of the example: in most instances of the problem, this will be the most common case.

Complexity. The space complexity of our algorithm is $O(N * |K|)$ where K is the set of all unique displacements. Because we modeled it after max-flow, using an optimized adjacency list representation for implementation results in $O(|Edges|*|Edges\_leaving\_source|)$ time complexity—in our construction, $|Edges|$ is $O(N' * |K|)$ while $|Edges\_leaving\_source|$ is $|K|$ giving complexity $O(N' * |K|^2)$. However, this upper bound does not take into account the fact that our three constraints substantially prune the search tree, and we find that the algorithm is quite efficient in practice, as demonstrated in Section 5. Additionally, $|K|$ is typically small because it is bounded by $(2 * DT)$ where DT itself is always less than the typical buffer size for the network application over whose traffic RD was calculated.

4. IMPLEMENTATION IN DUMMYNET

We modified Dummynet to support reordering. Our modified version takes regenerated reordering sequences as input along with its usual traffic shaping parameters. Dummynet has modules called "schedulers" that get invoked to regulate the processing and traffic shaping of incoming packets, and to control the release of emulated traffic towards its destination. We implemented our reordering functionality as a new scheduler. Our scheduler uses a buffer (the "reorder buffer") to hold packets that need to be delivered late. For example, if a packet is supposed to be reordered as 2 places late, we store it in the buffer until 2 other packets have arrived that can be sent before it. Our scheduler is agnostic to the specific sequence regeneration algorithm used, so users are free to choose metrics other than RD for emulation. Also, our scheduler allows users to supply more than one such reordering sequence as input. The scheduler then simply iterates over this list of sequences, picks one randomly, and applies the reordering dictated by it over the traffic it is emulating.

Our modified version of Dummynet is available on GitHub [3].

Experimenter Workflow. To create a full emulation, a user will typically:

1. Generate a set containing one or more RDs; these could be calculated over traces from a real network, taken from the literature, or derived a public measurement repository, etc.

2. Run the RD sequence regeneration algorithm over each RD in the set. This gives a set of reordering sequences.

3. Take above sequence(s) and input to Dummynet along with other traffic shaping parameters.

4. Run traffic from the system under test through Dummynet. Dummynet will reorder, delay, and drop packets according to the parameters provided in previous step.

The RD sequence regeneration algorithm is run offline, before the emulation or simulation begins, and the number of packets used in Step 1 is independent of the number of packets used in the experiment in Step 4. So, if Dummynet is made to emulate a million packets, the user only needs to run the RD regeneration algorithm in Step 2 once using, for example, 1,000 packets and Dummynet can use that 1,000-packet output in Step 4 repeatedly to shape the million packets.

5. EVALUATION

In this section, we use real and synthetic traces to show that our algorithm is scalable and works correctly in regenerating a reordered packet sequence from the input RD. We also evaluate our implementation in Dummynet to show that our modifications work correctly and do not introduce unnecessary overhead.

5.1 Real Network Traces

We ran the RD calculation algorithm for network traces taken from [4], consisting of long-lived connections from a host in Colorado to multiple destinations located on different continents, with the results collected hour-by-hour. We fed the RDs into our algorithm to regenerate the reordering sequences. We checked the correctness of the emulation by configuring our extended Dummynet using these sequences, sending traffic through it, and calculating RD' at the receiver. In all cases, our algorithm and Dummynet extension worked correctly, producing RD' = RD. The output RDs from two out of the five sets of measurements along with the corresponding RD's calculated over the regenerated sequence are shown in Table 4. The first is to a host in Cape Town, South Africa ($N' = 138275$, 0.3% reordering, sequence regeneration runtime 0.057s), the second to a host in India ($N' = 136768$, 0.11%, 0.126s). More results are reported in [15].

5.2 Synthetic Network Traces

This set of evaluations uses synthetic traces to evaluate scalability as both the number of packets and amount of reordering increase. Recall from Section 4 that this algorithm is run as part of the setup of a simulation or emulation, not as part of the experiment itself; thus, our goal is to keep times low enough that they do not become a major overhead in experiment preparation.

RD		*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

RD		*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: RD and RD' for destinations in Cape Town and India

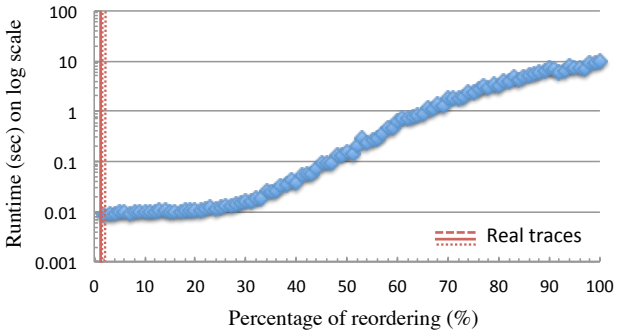

Figure 2: Effect of amount of reordering on runtime.

We conducted two sets of experiments. Based on observations from real network traces, we set $DT = 5$. For both sets of experiments, we collected preliminary results over 30 runs of each experiment. These results were used to decide the number of runs that would be needed for each data point in each experiment to get a 95% confidence interval with a sample error of ±5%. For comparison purposes, the parameters associated with the real traces are shown as vertical lines in the graphs.

In the first set, we generated packet traces in which we varied the amount of reordering while keeping number of packets, N', constant at 1,000. The RDs calculated over the generated traces were then fed to our implementation and the algorithm running times were calculated. Results are shown in Figure 2. The graph shows the mean value for each data point and the 95% confidence interval, which are narrow enough that they are not visible. The running time stays quite consistent for up to about 30% reordering and starts to increase after that; as the reordering becomes more commonplace, the algorithm runs into more "dead ends" and has to backtrack more. Up to about 60% reordering, the algorithm finished less than a second and even with 100% reordering, it finished within 10 seconds. As can be seen from the

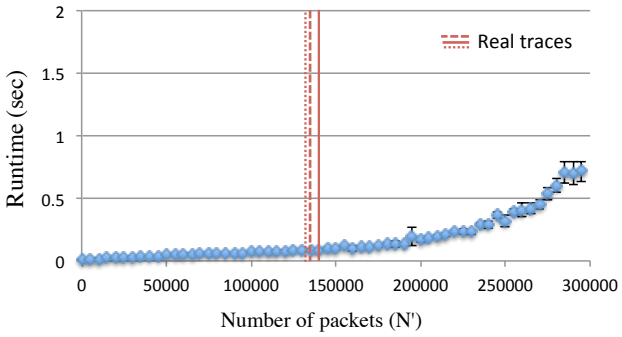

Figure 3: Effect of number of packets on runtime.

positions of the vertical lines, real Internet traces tend to be on the low end of this curve, with the algorithm always finishing in sub-second times.

In the second set of experiments, we generated traces in which we varied N' while keeping the amount of reordering constant. Again, the RDs calculated over the generated traces were used as input for our algorithm and running times were observed. The results are shown in Figure 3. The graph shows the mean value for each data point and the 95% confidence interval, which remains tight until the high end of the graph. The vertical lines again show the position of the real traces used earlier, at 130K to 140K packets. The running time is quite low for up to about 250K packets, staying below 2 seconds, but starts to increase past that point. The algorithm scales quite well to large traces, processing over a quarter million packets in less than two seconds.

5.3 Datapath Evaluation

Our final evaluation looks at whether the modifications we made to Dummynet increased the delay seen by individual packets during the emulation more than necessary. To show this, we ran an experiment 20 times that used flood ping to send 500 packets through the original Dummynet and M-Dummynet (our modified version), noting mean and maximum inter-arrival times. For this test, we used a reorder sequence with no reordering so that we could test the base overhead of our scheduler. The max inter-arrival times from all runs of the experiment were averaged using mean and are reported in Table 5. Statistical tests showed with 95% confidence level no statistically significant difference exists between the mean times from the two configurations, so we conclude that the base operation of our scheduler does not add unnecessary processing in the datapath.

We ran another experiment with the same configuration, this time using a reorder sequence that did induce reordering. The RD used for this experiment is shown in Table 6. The largest displacement in the RD is 19, which means that the max inter-arrival time we expect to see

Configuration	Mean time(ms)	Max time(ms)
Original Dummynet	2.34	2.48
M-Dummynet (Reordering off)	2.32	2.51

Table 5: Mean and max inter-arrival times on original Dummynet and on M-Dummynet.

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

Table 6: RD used in datapath evaluation, N'=20

Configuration	Max inter-arrival time (ms)	
M-Dummynet (Reordering on)	Expected	47.69
	Observed	48.07

Table 7: Expected and observed max inter-arrival times on M-Dummynet with reordering.

for this traffic is 19 times the base max inter-arrival time observed previously, or $(19 \cdot 2.51) = 47.69ms$; this is the amount of time the most-displaced packet should have to wait in the reorder buffer. The expected and observed times for this experiment are reported in Table 7, and a statistical test showed no significant difference between them with 95% confidence. Hence we conclude that the datapath in our scheduler does not introduce any additional latency beyond what is naturally caused by holding packets for reordering.

6. CONCLUSION

We argue that packet reordering is a prevalent network phenomenon that affects performance of both TCP and UDP, and cannot be ignored when conducting simulations and emulations. We presented a sequence regeneration algorithm that takes as input the RD metric and generates reordered sequences that can be used by any emulator or simulator to support fine-grained, controlled, and repeatable reordering. We built an extension for the Dummynet emulator to support precise reordering sequences. Using real and synthetic traces, we have shown that our algorithm is scalable and the implementation works correctly.

Acknowledgments

We thank Suresh Venkatasubramanian for his help in formulating the problem, and Kobus Van der Merwe, Shena Kasera, and our anonymous reviewers for their feedback on the work. This material is based upon work supported by the National Science Foundation under Grant No. 0709427.

7. REFERENCES

[1] A. Jayasumana, N. Piratla, T. Banka, A. Bare, and R. Whitner. Improved packet reordering metrics. *IETF RFC 5236*.

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

[3] A. Syed and R. Ricci. Code for the RD sequence regeneration algorithm and Dummynet extension. https://github.com/aishasyed/reordering, 2015.

[4] CNRL. Packet reordering trace. *http://www.cnrl.colostate.edu/Projects/PacketReordering/Trace/packet_reordering_trace.htm*.

[5] L. Ford and D. Fulkerson. Flows in networks. *Princeton University Press: Princeton*, 3, 1962.

[6] J. Bennett, C. Patridge, and N. Shectman. Packet reordering in not pathological network behavior. *IEEE/ACM Trans. Netw.*, 7:789–798, 1999.

[7] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, Apr. 2010.

[8] M. Laor and L. Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, 16(5):28–36, 2002.

[9] M. Lelarge. Packet reordering in networks with heavy-tailed delays. *Mathematical Methods of Operations Research*, 67(2):341–371, 2008.

[10] N. Piratla, A. Jayasumana, and A. Bare. Reorder density (RD): A formal, comprehensive metric for packet reordering. *Proc. NETWORKING*, LNCS 3462:78–79, May 2005.

[11] N. Piratla, A. Jayasumana, and T. Banka. On reorder density and its application to characterization of packet reordering. *Proc. IEEE LCN*, 1:401–414, Nov. 2005.

[12] N. Piratla and A. Jayasumana. Metrics for packet reordering – A comparative analysis. *Int. J. Commun. Syst.*, 21(1):99–113, Jan. 2008.

[13] V. Paxson. End-to-end Internet packet dynamics. In *ACM SIGCOMM Comput. Commun. Rev.*, 1997.

[14] 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*, 15(1):54–66, 2007.

[15] A. Syed. Realistic traffic shaping in the Dummynet link emulator. Master's thesis, University of Utah, 2014.

APPENDIX

Algorithm 1: ConstructGraph(R)

```
   // R is an RD table where R[i].displacement and
      R[i].count are the displacement and number of pkts.
 1 K = R.length // total number of displacements
 2 N = sum(R[i=1 to k].count) // total number of packets
 3 G = Empty Graph
 4 bipartite[] = K bipartite graphs, each length N
 5 Source = a vertex representing the super source
 6 Sink = a vertex representing the super sink
 7 s[] = array of K vertices, acting as sub-sources
 8 t[] = array of N vertices, acting as sub-sinks
 9 foreach vertex s[i] in s:
10     G.addEdge(Source, s[i], capacity=R[i].count)
11 for i=1 to K:
12     for j=1 to N:
13         G.addEdge(s[i], bipartite[i].leftVertex[j],
                     capacity=1)
14 for i=1 to K:
15     displacement = R[i].displacement
16     for j=1 to N:
17         k = j + displacement
18         if k >= 1 and k <= N:
19             G.addEdge(bipartite[i].leftVertex[j],
                   bipartite[i].rightVertex[k], capacity=1)
20 for i=1 to N:
21     for j=1 to K:
22         G.addEdge(bipartite[j].rightVertex[i], t[i],
                     capacity=1)
23 foreach vertex t[i] in t:
24     G.addEdge(t[i], Sink, capacity=1)
25 return G, bipartite
```

Algorithm 2: DFS(solution, step)

```
 1 if step > N: return true
 2 for i = 1 to K:
 3     if remainingPackets[i] == 0: continue
 4     vertex = bipartite[i].leftVertices[step]
 5     if vertex.hasRightVertex:
 6         rightVertexIndex = vertex.rightVertexIndex
 7         if solution[rightVertexIndex] != null:
 8             continue
 9         solution[rightVertexIndex] = vertex
10         remainingPackets[step]−−
11         if solve(solution, step+1): return true
12         remainingPackets[step]++
13         solution[rightVertexIndex] = null
14 return false
```

Algorithm 3: Solve(G)

```
 1 solution[] = array of size N
   // solution[1]=5 means input packet 5 should be
      placed in position 1 in the solution sequence
 2 remainingPackets[] = array of size K, initialized such
      that remainingPackets[i]=R[i].count
 3 if DFS(solution, 1): return solution
 4 else: return null // no solution!
```