# The Playback Network Simulator: Overlay Performance Simulations With Captured Data

Emily Wagner

A project report submitted to the Johns Hopkins University in conformity with the requirements for the degree of Master of Science in Engineering

December 2016

Advisors: Dr. Yair Amir, Amy Babay

## Acknowledgments

I would like to thank Amy Babay for her contributions to this work, along with her support, advice, and answers to many questions throughout the duration of this project, from the initial design of the tool, through the writing of the project report. She is an exceptionally thoughtful, patient, and hard-working person, and these traits, combined with her tremendous talent for writing both publications and code, make her a formidable force in anything she pursues.

I would also like to thank Dr. Yair Amir for his mentorship, which started when I took his Intermediate Programming course on C/C++ and continues through today. He is a visionary person with a rare talent for selecting meaningful problems and inspiring others to contribute their best efforts to them. I look forward to continuing to work with him on a vision for next-generation Internet services when I join LTN Global Communications in January.

I would like to thank LTN Global Communications for allowing Amy and me to run our data collection software on their globe-spanning commercial overlay network over several months. Their generosity enabled us to discover some fascinating results and design new routing algorithms based on real network data.

Finally, I would like to thank the other members of the Distributed Systems and Networks lab at Johns Hopkins who helped me with with my Masters project and distributed systems courses: Thomas Tantillo, Daniel Obenshain, and Jeffrey DallaTezza. It was a privilege to work with these talented scientists and engineers.

This work was supported in part by NSF grant 1535887 and by DARPA grant N660001-1-2-4014. Its contents are solely the responsibility of the authors and do not represent the official view of the NSF or DARPA.

## Contents

## 1 Introduction

The Playback Network Simulator models flows from any source to any destination on an actual overlay network, using fine-grained data it records on the network to closely match the behavior that the flow would have experienced at the specific time the data was recorded.

To enable modeling any source-destination flow using many different routing protocols that may not be known in advance, the data recording component of the simulator collects fine-grained data for each link in the overlay topology. The simulator then uses this per-link data to model the performance of a complete flow through the network: based on the time a packet is sent at a node, the simulator can calculate what behavior it would experience on that node's outgoing links at that time from the recorded data (i.e. whether it would arrive at the other side of the link, and if so, what its latency would be).

If the rate of the flow being modeled is the same as the rate of the recorded data, the behavior of each simulated packet can be determined directly by simply using the behavior of the recorded packet with the closest send time for each overlay link the simulated packet should be propagated over. This approach provides a nearly exact playback of the recorded data, but adapted to model a complete flow, rather than each individual link in the topology.

In general, however, a flow can have any sending rate, and may employ recovery protocols or redundant sending on each link, requiring more complex modeling. Therefore, the simulator uses a sliding window to model the behavior of a link at a specific time. For each link a simulated packet traverses, Playback considers all the packets within a sliding window centered at the time the simulated packet is sent on that link, calculating the loss rate and average latency over that window. To determine whether the packet is successfully transmitted, the simulator randomizes based on the loss rate at that time: if the packet is successfully transmitted, the average one way link latency for the window is used as the packet's latency.

Using this approach, the Playback Network Simulator can model flows of any sending rate between any source and destination. The simulator can model flows using a single path through the overlay from the source to the destination or using any arbitrary graph consisting of links in the overlay. The simulator includes built-in support for several different source-based routing protocols, including single-path, multiple disjoint paths, and a dissemination-graph-based protocol we developed to address problems around a flow's source or destination via targeted redundancy. Playback can be extended to support additional protocols. In addition, the simulator can model recoveries using a particular family of real-time recovery protocols.

Playback is a valuable tool for designing and tuning routing schemes because many different routing schemes and recovery parameters can be compared for the same data at a given time. Playback was used to design and provide simulation results for the targeted-redundancy dissemination graphs approach described in [6].

## 2 Related Work

Network simulators use mathematical models or captured data to simulate the behavior of a network. There are many network simulators in use today.

The ns-2 ([8]) and ns-3 ([2]) simulators are among the most popular. Both ns-2 and ns-3 accept explicit definition of the nodes, links, and packet queues, along with the loss rates and latencies. They includes standard internet protocols such as TCP, UDP, BGP and OSPF, along with common internet applications, such as HTTP. Additional protocols can be added manually. Both ns-2 and ns-3 are discrete-event network simulators, meaning that they simulate each packet. They also have an emulation mode, in which simulator instances can interact with and capture real network packets. This can be used to verify simulation results in emulation, as a simulated experiment can be easily adapted to an emulated experiment.

OMNeT++ ([12]) is another popular open-source simulation framework. In its core design, it is similar to ns-2/ns-3 in that it allows specification of a network topology and then performs discrete-event simulations.

In our overlay simulations, we can ignore the exact behavior of the underlying network because it is encapsulated by the captured packets. In this way, overlay and peer-to-peer simulators, such as [7], [11], and [10] fulfill a more comparable role. OverSim ([7]) is a peer-to-peer and overlay simulation framework that uses the OMNeT++ simulation framework. It simulates peer-to-peer and overlay protocols such as Chord, Kademila, Pastry, and GIA. PeerSim ([11]) is designed to simulate extremely large, high-churn networks. PlanetSim ([10]) provides an overlay simulation framework with a wrapper than allows the same simulation code to be deployed on network testbeds such as PlanetLab. It supports the creation and testing of overlay algorithms (e.g. Chord and Pastry) and overlay services (e.g. DHT) on top of existing overlays.

Network emulators create and send packets over a virtual network or test network that introduces latency and loss artificially. Examples are [13], [1], and [3]. These fulfill a different role, as they generate real network traffic on an emulated network, whereas Playback captures real network traffic on a real network, and then uses this to perform high-fidelity simulations.

Our goal with this work is to simulate arbitrary flows at a given time that use per-link data collected on a real network at that time, and to our knowledge, there is no existing software that achieves this goal.

## 3 The Playback Network Simulator

The Playback Network Simulator has two main components: data collection and simulation using the captured data.

The data collection software can be deployed on any overlay network. On a specified interval, every node sends a packet on all of its outgoing links. Nodes log all packets they receive. The simulator then uses the packets captured on each overlay link to simulate end-to-end performance for messages sent from a source to a destination anywhere in the network. The loss and latency characteristics of a given simulated link at a particular time are determined by captured packets sent on the real overlay link near that same time.

Simulated packets leave the source at a given sending rate (which can differ from the data collection sending rate), and then propagate through the simulated overlay network based on the specified routing protocol, given as a subset of overlay links (i.e. a graph) in the topology. Reroutes are specified as list of graphs and times to use each graph. For several families of routing protocols, including single path, $k$ node-disjoint paths, and targeted redundancy dissemination graphs, Playback includes programs to calculate necessary reroutes and their corresponding times.

Since Playback was designed to simulate performance for applications requiring always-timely, almost-always reliable service, the recovery protocol is selected from a family of real-time recovery protocols, including those of [4] and [5]. In these recovery protocols, $M$ retransmission requests are sent in a row, separated by $\varepsilon$ ms. If any one of these $M$ requests is received, $N$ retransmissions are sent, separated by $\gamma$ ms. Any additional requests received for that same packet are ignored. In addition, the requesting node will also wait $\Delta$ ms before sending its burst of requests, since packets may be re-ordered rather than lost. In support of always-timely, almost-always reliable applications, a *time deadline* can be specified: if messages take longer than the deadline to propagate from source to destination, they are considered *late.*

Finally, redundant per-link sending can be simulated using Playback. Multiple duplicates of a message can be sent from the source and propagated through the network on all available links, leaving the source separated by $\beta$ ms. If redundant per-link sending is used, the message is considered on-time if any of the duplicates arrives at the destination within the deadline.

Given the captured data, source, destination, sending rate, routing protocol, recovery protocol, deadline, and redundant per-link sending parameters, the Playback Network Simulator provides many metrics, including both per-message performance metrics and per-run summary metrics.

For each simulated message propagated from the source to the destination, Playback reports whether it arrived at the destination, and if so, what nodes were used on the fastest path and the corresponding latency. If the message did not reach the destination on time, Playback classifies the problem into one of four categories: source problem, destination problem, source and destination problem, and other (if there were problems in the middle of the network).

Playback also gives several summary statistics for each examined time period, including timely reliability overall, eventual reliability (if the deadline is ignored), average latency, and total messages attempted, late, dropped, or lost in disconnection. Normally, lost messages are considered *dropped*, but if recoveries are used and it takes too long for the loss to be detected (because there is an extended period of 100% loss rate), it is considered *lost in disconnection.* Playback also reports the sum of all problem classifications for messages that were dropped, late, or lost in disconnection. It provides a list of all the fastest paths used for messages throughout the run, along with their respective message counts.

Two cost metrics are measured: the average number of overlay links used in the routing protocol; and average number of packets sent on overlay links per message that reached the destination on time. The average number of overlay links measures the average size of the graph used (which varies if there are reroutes), while the average number of packets counts the ‘true cost’, from an ISP perspective, of propagating the message from source to destination. ISPs charge per packet sent on each overlay link, so this cost measure includes recoveries and redundant sending on each link. It also counts late packets and packets dropped in the middle of network as overhead for messages that were successfully delivered.

## 3.1 Data Collection

Playback's data collection stage provides the simulation with *raw logs* containing fine-grained data used to calculate performance for simulated packets. Simulated packets sent at a given time will base their latency and loss rates on captured packets sent near that same time.

The data collection program, `collect_cloud_data.c`, is deployed on any overlay, with an instance at each node. Every node sends a packet on all of its outgoing links at an interval determined by the data collection sending rate, which should be around at least 1/10th as frequent as the sending rate of the simulated flow. For example, if a simulated flow of 1ms is desired, 10ms would be a reasonable sending rate (but a more frequent sending rate would give a more accurate simulation). When a node receives a packet, it logs the packet sequence number, the source ID and IP address, the destination ID and IP address, the time the packet was received (on the receiver's clock), the time the packet was sent (on the sender's clock), and several metrics that aid in calculating the round-trip time for the packet. Every overlay node starts sending with sequence number 1 when the program begins; the sequence number is incremented before the node sends a packet to all of its neighbors.

The round-trip time for each packet is calculated using information about the last packet the sender received from the receiver. Each node saves a full second of sent packet records for each neighbor that contain the times it sent the packets and their respective sequence numbers. Packets sent to a given neighbor include the sequence number, $seq_{LR}$, and receive time, $t_{received_{LR}}$, of the last packet the sending node received from that neighbor. This can be used to calculate $\Delta_{elapsed}$: the difference between the time the sending node sent this packet and the time it received the last packet from the receiving node. The receiving node finds $seq_{LR}$ among the records it has saved in memory and finds the time it sent the last received packet, $t_{sent_{LR}}$. Finally, the round-trip time for the current packet can be calculated as follows:

$$\text{round-trip time} = t_{received} - t_{sent_{LR}} - \Delta_{elapsed} \qquad (1)$$

where $t_{receiver}$ is the time the current packet was received. This parallels a ping protocol, but these calculations eliminate the need to send responses to every received packet, as both sides are regularly sending messages. In addition, with this scheme, the round-trip time can still be calculated for every received packet even if some packets are lost.[1] A major complicating factor in this calculation is the fact that the sender and receiver clocks are not perfectly in sync, as will be discussed in Section 3.2.1.

[1]If there are re-orderings of more than a second, a different round-trip time estimation is needed (similar to the one-way latency discussed in Section 3.2.1).

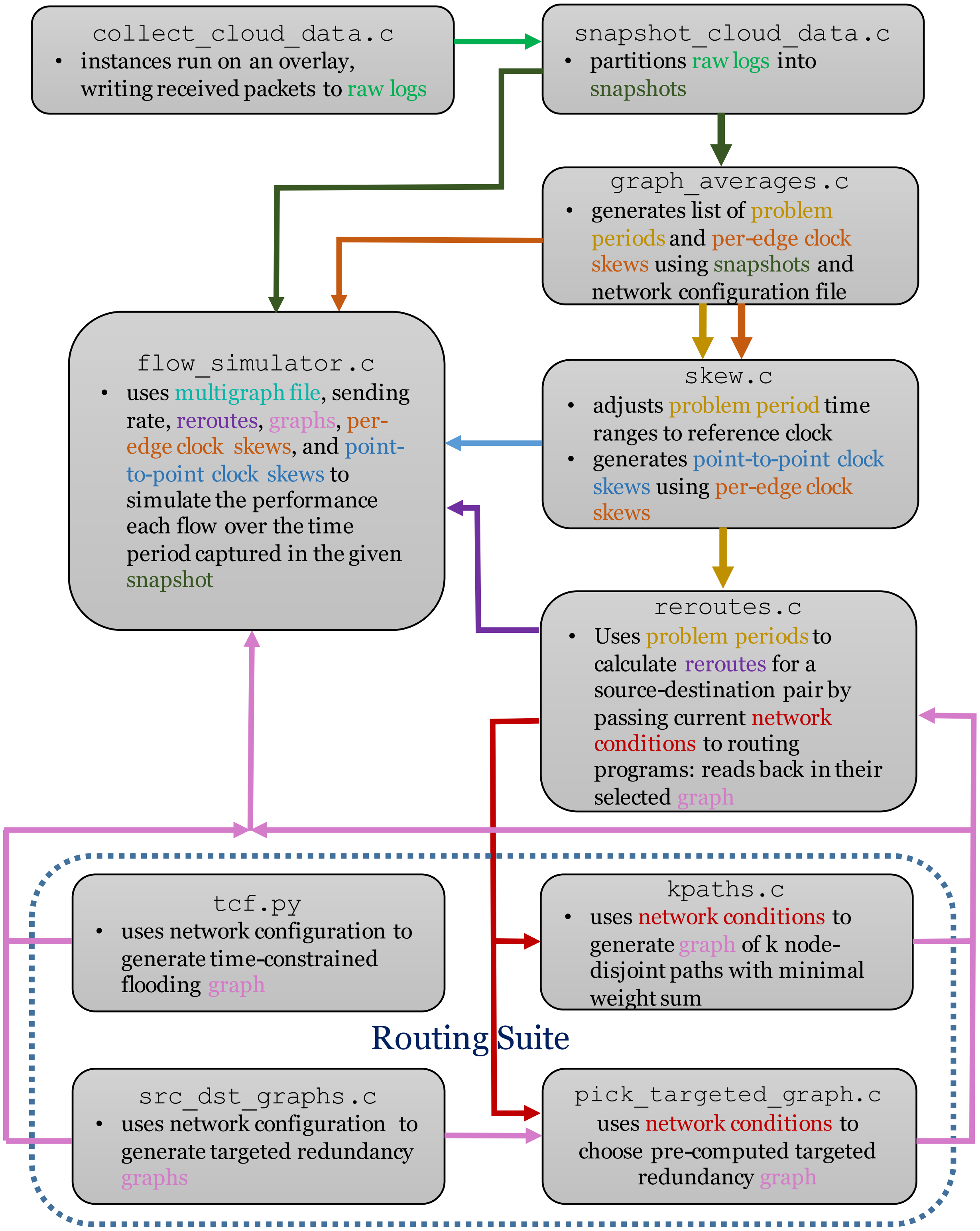
collect_cloud_data.c
• instances run on an overlay, writing received packets to raw logs
snapshot_cloud_data.c
• partitions raw logs into snapshots
graph_averages.c
• generates list of problem periods and per-edge clock skews using snapshots and network configuration file
flow_simulator.c
• uses multigraph file, sending rate, reroutes, graphs, per-edge clock skews, and point-to-point clock skews to simulate the performance each flow over the time period captured in the given snapshot
skew.c
• adjusts problem period time ranges to reference clock
• generates point-to-point clock skews using per-edge clock skews
reroutes.c
• Uses problem periods to calculate reroutes for a source-destination pair by passing current network conditions to routing programs: reads back in their selected graph
tcf.py
• uses network configuration to generate time-constrained flooding graph
kpaths.c
• uses network conditions to generate graph of k node-disjoint paths with minimal weight sum
Routing Suite
src_dst_graphs.c
• uses network configuration to generate targeted redundancy graphs
pick_targeted_graph.c
uses network conditions to choose pre-computed targeted redundancy graph

Figure 1: A flow chart containing the general purpose of each Playback program and the relations between programs. Arrows denote input to and output from each program; the arrow's color matches the text color used for the file or information being passed from program to program.

## 3.2 Simulation

### 3.2.1 Initial Processing: Detecting Network Problems and Calculating Clock-Skew

Initial processing has two main components: detecting loss and latency network problems and calculating the relative clock skews between all nodes. These will be used in reroute calculations and in the simulation's one-way link latency calculation.

First, depending on the file size of the captured data, it may first need to be split into sections, called *snapshots*, small enough to be stored in memory (using `snapshot_cloud_data.c`).

Then, `graph_averages.c` detects network problems in the snapshots. In a real system (like the Spines overlay messaging framework), overlay nodes may flood updates when they detect changes in loss or latency on their incoming links. Such updates may cause flows to be rerouted around problems. To approximate this behavior, Playback uses *problem periods*: `graph_averages.c` maintains a sliding *problem detection window* and traverses over the captured data. When the loss rate or average latency in the current problem detection window crosses a given loss or average latency threshold, a problem period begins; it ends when the loss rate and average latency are below their respective thresholds for a full problem detection window. Each problem period start and end time is written to file, along with the maximum loss rate and maximum average latency experienced over all problem detection windows in the period. This is somewhat inaccurate, as in a real system, nodes would likely send additional updates if the problem significantly worsened or became less severe. Note that the loss and latency thresholds, along with the length of the problem detection window, are parameters that can be specified by the user.

In addition to detecting network problems, initial processing also addresses issues stemming from clock skew between machines. Internal clocks on machines drift over time, and at varying rates. Thus, clocks between machines may not match exactly, so it is not correct to use the timestamps on the packets captured during data collection on overlay nodes that did not create the timestamp.

This issue arises in two contexts: calculating one-way latencies and calculating time ranges of problem periods and reroutes. Because we find that network problems that result in increased latency on an overlay link often occur in only one direction, using the half the round-trip latency is inaccurate. Instead, Playback uses the *one-way latency*:

$$\text{one-way latency} = t_{\text{received}} - t_{\text{sent}} + \mathit{skews}[\text{receiver}, \text{sender}] \tag{2}$$

Here, $t_{\text{received}}$ is the time the packet was received on the link receiver's clock, $t_{\text{sent}}$ is the time the packet was sent on the link sender's clock, and $\mathit{skews}[\text{receiver}, \text{sender}]$ is the time difference between the receiver's clock and sender's clock on a given overlay link.

The `graph_averages.c` program calculates these per-link clock skews by assuming that the latencies are symmetric (i.e. the one-way latency equals half the round-trip latency) when the link round trip latency is within a 'normal' range; then, Equation 2 can be manipulated to obtain the clock skew for that link. A normal range is considered to be within 10% of the round-trip latency given in a *network configuration file* containing all links in the overlay topology. [2]

[2]It is possible that there are no captured packets in the normal range for a given link in the entire snapshot. In

The second context in which clock-skew issues arise is in the calculation of problem periods and reroutes: from the captured data, we can only determine directly when the period occurred on the problematic link's receiver's clock. However, in the flow simulator, the list of reroutes and their corresponding times is interpreted on the flow source's clock. If the time to reroute to a new graph has arrived on the source's clock, then the message will be propagated to the destination using this new graph. If we want to correctly interpret the problem period start time for any source in the network, we will need a way to approximate the clock skews between any two nodes in the network.

Therefore, `skew.c` is used to calculate latencies between arbitrary nodes. It runs a breadth-first-search from a *reference clock* overlay node. The distance calculated between nodes in this BFS is the sum of the per-link clock skews on the path between each node and the reference clock node. This results in the clock skew between each node and the reference clock node. The start and end times in the loss and latency problem file generated by `graph_averages.c` are then modified to use this reference clock. Finally, the flow simulator will adjust the reroute times from the reference clock to the source clock. Using a reference clock eliminates the need to calculate and record point-to-point clock skews for every possible node combination in the network.

### 3.2.2 Routing

In Playback, each message's routing is specified as a graph containing all links that it traverses to reach the destination. Thus, any routing algorithm can be added as long as it can be specified as a list of graphs that each message uses and the times for which the graphs are used. All routing algorithms currently implemented in Playback are source-based dissemination graph routing algorithms, in which each message is stamped with the complete set of links it will traverse when it leaves the source. While dissemination graph-based routing is naturally transferred to Playback's routing framework, link-state routing can also be added: all reroutes are pre-calculated before being passed to the flow simulator.

Playback's `reroutes.c` calculates reroutes as a list of times to switch between graphs. It uses the list of loss and latency problems generated by `graph_averages.c` to calculate reroutes for a given source and destination over the duration of the snapshot. Every time a loss or latency problem starts on the current graph, it will call a separate program to calculate the graph to use given the current network conditions with the addition of this new problem. If the graph changes, the prior graph and the time for which it was used is written to file. Every time a problem ends anywhere in the network, the process is repeated. Note that the times for reroutes written to file use the reference clock discussed in Section 3.2.1.

The Playback Network Simulator includes routing protocols for $k$-node-disjoint paths routing (which includes single-path routing, i.e. $k = 1$), along with time-constrained flooding and targeted-redundancy dissemination graph routing. Today, `reroutes.c` calculates reroutes for either $k$-disjoint paths routing (by calling `kpaths.c`) or targeted redundancy dissemination graph routing (by calling `choose_dissemination_graph.c`

these cases, a second pass is required: a python script, `fill_in_blanks_and_rerun.py`, finds snapshots before and after this period that do have clock skews for this link, and uses these clock skews to fill in the missing link clock skew in the current snapshot.

When called, `kpaths.c` will read in the current link weights. These weights are the links' *expected latencies* from [4]:

$$u(1-p)\cdot T + (p - 2p^2 + p^3)\cdot(3T+\Delta) + (2p^2 - p^3)\cdot T_{max} \tag{3}$$

Here, $p$ is the current loss rate of the link, $T$ is the current latency of the link, $3T+\Delta$ is the time to recover a lost packet (a constant $\Delta$ to detect the loss, plus three propagation delays for the original send, request, and retransmission), and $T_{max}$ is the maximum allowed latency, used as a penalty for packets that are lost and cannot be successfully recovered by the deadline. The $k$-paths routing algorithm finds $k$ node-disjoint paths for which the sum of the weights is minimized.

When `choose_dissemination_graph.c` is called, it reads in a list of current problem periods and routes based on the targeted-redundancy dissemination graph routing scheme. This routing scheme generally uses 2 node-disjoint paths, except if a problem is detected at the source or destination; in these cases, graphs called *source-dissemination* and *destination-dissemination* graphs are used instead. These graphs maximize the number of routes out of the source and into the destination, respectively. An additional targeted redundancy graph called the *robust source-destination problem graph* combines the source and destination problem graphs, with an additional check to make sure that two node-disjoint paths are still available.

Switching to these problem graphs is determined by two different thresholds: an *on-path link problem threshold* and a *general link problem threshold.* The general link threshold determines how many links connected to the source or destination must have current problem periods before switching to the corresponding targeted redundancy graph. The on-path link threshold determines how many links connected to the source or destination *used by 2-node disjoint paths* must have current problem periods. Both thresholds are considered independently for the source and destination (i.e. to switch to the source graph, the on-path link problem threshold or general problem threshold must be exceeded only by links connected to the source). If the source or destination-dissemination graph is selected based on current network conditions, and additional problem periods affect links on this graph *not* at the source or destination, respectively, then the robust source-destination dissemination graph is used.

This approach was shown in [6] to be very close to the optimal performance possible for a 65ms time deadline.

Playback also includes `tcf.py`, which is an implementation of the time-constrained flooding algorithm. Overlay flooding sends on every link in the entire topology. It is very expensive, but clearly optimal for the network conditions. Time-constrained flooding improves on the cost of overlay flooding by not sending packets to nodes from which they cannot reach their destination within the time allowed. It also prunes nodes whose only paths to the source and destination overlap (i.e., they are part of node loops). Thus, it still maintains the optimal reliability given by overlay flooding for a given time deadline, but at a reduced cost.

### 3.2.3 Flow Simulator

The flow simulator, `flow_analysis.c` uses the snapshots, routing, resend parameters, source-destination pair, sending rate, and redundant per-link sending to give a performance profile for a given flow, including per-message behavior (loss, latency, path used from source to destination, problematic links), along with summary statistics (aggregating per-message behavior). Example output can be seen in Section 4. The flow simulator can perform many different simulations for a given sending rate and set of snapshots using a *multigraph file*, as will be discussed in Section 5 (i.e. many flows can be simulated during the same run, as long as the sending rate and snapshots/time range are the same).

The simulator must stitch together evolving fine-grained per-link behavior to simulate flows between any 2 nodes in the network. It must also be flexible enough to support diverse dissemination methods and recovery methods. The simulator should also compare different methods and parameters in a consistent way.

Dijkstra's algorithm, augmented with time-evolving link weights as well as probabilistic packet loss and recovery, is the basic framework used to solve this problem. Messages leave the source at an interval defined by the sending rate; link latencies and loss rates are continuously evaluated as each message propagates through the network. When a packet traverses a link in the graph, the latency and loss rate of the link at that time are calculated using a sliding *modeling window* centered at the time the packet leaves the link's start node. The loss rate used for this simulated packet is the number of lost captured packets divided by the total number of captured packets in the window. The latency is the average one-way latency (as described in Section 3.2.1) among all delivered captured packets in the window. To determine whether the packet successfully traverses to the link's end node, a random number is generated: if it is less than or equal to the loss rate, the simulated packet is lost. Otherwise, the packet is propagated to the link's end node with the average one-way latency for the window.

Now, if the receiving node is not the destination, it must forward this packet along its outgoing links. In order to calculate the sliding window loss rates and latencies for the outgoing links, we need know the time the simulated packet arrived at the receiving node: we will use the arrival time of the packet that took the fastest path to this node as the sliding modeling window center for its outgoing links [3]. The arrival time for a packet is calculated as the time the packet was sent from the link start (on the link start's clock) plus the average difference between receive times on the receiver's clock and send times on the sender's clock for captured packets in the window. Note that this would be equal to the one-way latency plus the clock skew if the clock skew were perfectly accurate at all times.

The graph over which Dijkstra's algorithm will perform its shortest-path search is determined by the routing algorithm. Reroutes, as discussed in Section 3.2.2, are given as a list of graphs and times to switch to these graphs. The flow simulator adds a constant *update propagation time* (given as a user parameter) for the knowledge of the problem to propagate through the network. Each reroute start time is adjusted from the reference clock to the source clock and increased by the update propagation time and the length of the problem detection window. Then, when this modified reroute start time passes in the simulation, the next message propagated from the source switches to use the new graph. If there are gaps in time in the list of reroutes, `flow_simulator.c` will switch to the *default graph*, which, by convention, is the first graph given in the user-provided list of graphs. Switching back to the default graph has a separate *switch-back* time, which may be useful if a routing algorithm delays switching back to a previously problematic graph until it has

[3]Note a node will only forward a given message once, ignoring all duplicate copies.

been problem-free for an extended period.

Recoveries are in the family discussed in Section 3. When a packet is dropped on a link, the flow simulator propagates packets to detect losses: another packet must successfully traverse this link in order for the first loss to be detected. The simulator will only attempt to detect the loss on the current link until a full deadline has passed. For example, with a deadline of 65ms and sending rate of once every millisecond, `flow_simulator.c` will attempt to detect the loss 65 times (the loss detection packet send times will increment by 1ms each time as well) before giving up. If the simulator gives up, this packet is said to be *lost in disconnection*. Note that consecutive packets that are lost must be independently detected: in a real system, receiving a single packet with a much higher sequence number might allow you to detect several losses at once, but in the simulation, they must all separately receive a subsequent packet to be detected. Once the loss has been detected, the receiving node will wait $\Delta$ ms, and then send $m$ retransmission requests separated by $\varepsilon$ ms. The sending node will send $n$ retransmissions separated by $\gamma$ ms. The window center for the message average continues to shift throughout the recovery process, so the $\varepsilon$ and $\gamma$ parameters can potentially move the window into a less lossy period.

The use of randomization for packet loss can cause certain anomalies: for example, routing schemes with recoveries may occasionally perform slightly worse than the same scheme without recoveries over the same time period, especially among small sample sizes or short time frames. When Playback is run with the *minimal* randomization mode, which uses a single random generator for all packet traversals across all links in the network, these effects can be observed. To ease comparison between routing and recovery schemes, Playback also includes a *consistent* randomization mode which guarantees identical behavior for a packet sent at a given time across a given link for a run starting at a given time. This mode, however, is significantly more computationally expensive, as many more randomizations are performed than in the minimal randomization mode..

In consistent randomization, there is one set of random number generators (consisting of a random generator for each link) for each redundant send, one set for each retransmission request, one set for each resend attempt, and one set for loss detection. This way, if two routing algorithm , such as single path and two node-disjoint paths, use the same link, and the initial attempt for a packet is dropped on that link at a certain time in the single path scheme, the same loss will occur for 2-node disjoint paths if the link is used at the same (or a sufficiently close) time. Consistent randomization also unifies across redundant sends and recoveries. For example, if you are comparing different numbers of retransmission requests on a given link at a given time (for example, $m = 2$ and $m = 3$, with identical timing parameters $\varepsilon$ and $\gamma$), the first 2 attempts will be identical, as each retransmission attempt gets its own random number generator. Note, however, if the two schemes arrive at the link start node at different times (because their paths to the start node differ), then while the random numbers used will be identical, the modeling window may differ because the window center differs.

Consistent randomization incurs significant overhead. This is because regardless of whether or not a link is used, and regardless of whether or not recovery is needed, the maximum possible number of randomizations will need to be performed on all links for each message propagated from the source. This ensures that all random generators will be synchronized. Despite the overhead, consistent recovery is very valuable for comparing the results of different routing approaches, recovery schemes, and redundant sending schemes.

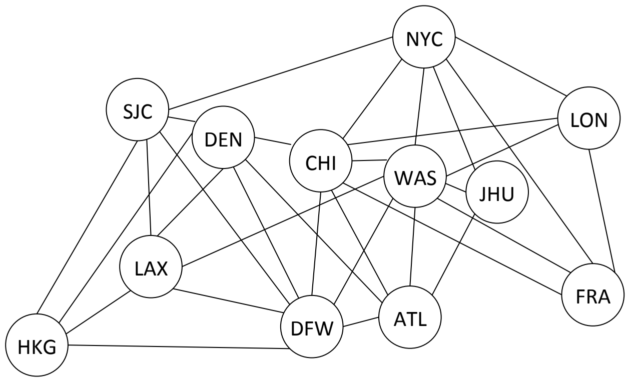

Figure 2: Overlay network topology on which experimental data was captured and later used for simulations.

The flow simulator includes several verbose modes to help understand the performance of a given flow in varying degrees of detail. The `-v` mode prints out the end-to-end results of every message sent from the source, along with the latency and path used. It can be used to generate plots like those seen in Figures 3 through 8. The `-vv` mode prints the results of every link, as well as the end-to-end results of every message sent from the source. The `-vn` prints only the end-to-end results of messages that were late, dropped, or lost in disconnection.

Finally, the `-vvn` mode prints the end-to-end results of messages that were late, dropped, or lost in disconnection, along with all of links that were late, dropped, or lost in disconnection for those problematic messages. Using knowledge about these links, the `-vvn` mode classifies each problematic message as a source problem, destination problem, source and destination problem, or other problem (if it was not a source or destination problem). This classification scheme has a few subtleties: when $k$ node-disjoint paths routing is used, the simulator can determine whether packets lost at the source or destination cut the network. If this is the case, any late packets in the middle of the network do not count towards the problem classification, as source and destination losses dominated the end-to-end behavior. The same capability (of ignoring interior problems if source and destination losses cut the network) is not yet available for more general graphs as there is currently no ability to detect whether the source and destination losses cut the network.

## 4 Case Study: Targeted Redundancy Dissemination Graphs Evaluation

Playback was developed to aid in designing protocols for remote applications with extremely strict timeliness constraints, such as remote manipulation and remote surgery. For the work in [6], we evaluated six different dissemination graph approaches with a 65ms deadline for 16 transcontinental flows between 4 cities on the east coast of the US (NYC, JHU, WAS, ATL) and 2 cities on the west coast (SJC, LAX). The overlay network topology, consisting of 12 nodes in the US, Europe, and East Asia, can be seen in Figure 2. This network was adapted from LTN Global Communications' overlay topology [9].

At 5:06am on October 17, 2016, a high-loss period lasting 60 seconds affected all of Los Angeles' (LAX) incoming links. We examine how this problem affected on-time delivery of messages from New York City (NYC) to LAX. We consider 6 dissemination graph routing approaches with $m = 1$, $n = 1$ recoveries.

Single-path routing with the ability to reroute, or *dynamic* single paths, is the least expensive dissemination approach considered. In this simulation, the path is calculated using `kpaths.c` for $k = 1$. The message profile, which depicts the latency of each message sent from NYC to LAX, can be seen in Figure 3. Blue dots indicate the message was delivered on time. Red dots with latency 0 indicate the message was dropped or lost in disconnection, and all other red dots indicate indicate the message was late. The blue line depicts messages that arrived without using recovery on this path, the high red line depicts recoveries on this path (all of which were late), and the low red line depicts messages that failed to be recovered. Table 1 contains the summary metrics for this approach and flow, including the raw message counts, fastest paths taken along with their respective counts, and the two cost metrics measured. The problem classification is not included in the tables as all late/lost messages were classified as destination problems for all six approaches.

Dynamic single-path routing selected the `NYC` → `WAS` → `DFW` → `LAX` route for nearly the entire duration of the problem, as its expected latency was lowest. However, it was not possible to recover on this path within the deadline, so single-path routing effectively suffered the same low delivery rate as the underlying link (about 75.7%). This shows a potential shortcoming in the expected latency metric as it is used in this simulation: $T_{max}$ (as seen in equation 3) is set to 300ms, to accommodate the longest links in the overlay topology. While this setting does penalize longer recovery times more than short ones, in most cases, it penalizes links more if recovery is not likely to be successful than if recovery is successful but the packet arrives late. Another choice might have been to penalize dropped packets that were not likely to be recovered within the deadline as much as if they were lost (by setting $T_{max}$ to 65ms and capping both $T$ and $3T + \Delta$ at this $T_{max}$). However, this makes many paths with a large difference in latency look identical.

An example of rerouting can be seen towards the end of the period. The problem ends slightly earlier on the `DEN` → `LAX` link, so the single path switches to `NYC` → `CHI` → `DEN` → `LAX`. The latency on this path is around 38ms, compared to the original path's 35ms. This corresponds to the small section of higher latency blue dots at 05:05:14. Then, the `DFW` → `LAX` link has a brief period of good performance, causing the path to return to the `NYC` → `WAS` → `DFW` → `LAX` path. Unfortunately, by the time the link update has propagated back to the source, the problem has recurred, and a few more packets are delivered late: this can be seen in the small disconnected section of red dots at the end of the period.

In Figure 4 and Table 2, the performance of dynamic single path routing with redundant per-link sending is evaluated. Two copies of every message are sent on each link, separated by $\beta = 0.5$ms. The routing is identical to dynamic single path routing, but the reliability is much higher (91.4%) due to the redundant per-link messages. The red line for late packets is much less thick. This is because redundant sending on each link also allows more chances for the loss to be detected in a timely fashion, as the effective flow rate of packets on a link is twice as fast. Note that this simulation is a very optimistic view of the performance of redundant per-link sending: losses are independent of one another in this simulation, but on the real internet, they are bursty (if one packet is lost, the likelihood that the follow packet will be lost is much higher).

Figure 5 and Table 3 show the performance of two node-disjoint paths without reroutes, or *static* two node-disjoint paths. Recoveries do not help on either path here because the two paths are calculated from the network configuration when there is no loss, so the expected latency does not incorporate the recovery time. Still, this approach achieves 93.2% reliability. It is interesting to note that the two disjoint paths have nearly identical latencies when no recoveries are used, but when when a recovery is necessary, the latencies differ by around 35ms because links into the destination (`DFW` → `LAX` and `WAS` → `LAX`), where the packets were dropped, have different propagation delays.

Figure 6 and Table 4 show the performance of dynamic two node-disjoint paths routing. This is the first example where recoveries allow additional messages to be delivered, as the last hop of the `NYC` → `SJC` → `LAX` path can be recovered within the deadline. The reliability achieved is 97.4%. Note that the packet cost per on-time packet of this approach is actually cheaper than single path routing with per-link redundant sending and two static node-disjoint paths routing because the performance is significantly better; thus the overhead from late and lost packets is reduced. In addition, for single path routing with redundant sending, the packets are being sent twice on a path that touches 4 nodes, whereas the two-paths approaches each use one path that touches 4 nodes and one path that touches 3 nodes.

Figure 7 and Table 5 show the performance of targeted-redundancy dissemination graph routing. Recall that normally, this dissemination scheme uses 2 static node-disjoint paths. When there is a problem at the source or destination, it uses a targeted redundancy graph, which was, in this case, a destination dissemination graph. Even with all of available links into the destination, it was still only possible to recover on the `NYC` → `SJC` → `LAX` path. However, the additional destination links allowed more opportunities to reach the destination when recovery was not needed, as can be seen by the lower 3 blue lines. This approach reached 99.6% reliability.

Figure 8 and Table 6 show the performance of time-constrained flooding. This looks very similar to targeted redundancy dissemination graph routing (it also achieved 99.6% reliability), but the cost is much higher. In addition, in this message profile, there are only 2 blue lines that do not use recovery. The third blue line is collapsed into the existing 2, as when the destination dissemination graph must use the `NYC` → `WAS` → `DFW` → `DEN` → `LAX` path, time-constrained flooding can use a faster path, `NYC` → `CHI` → `DEN` → `LAX`. Note the targeted redundancy graph takes a longer path to `DEN` because it results in a graph with fewer links.

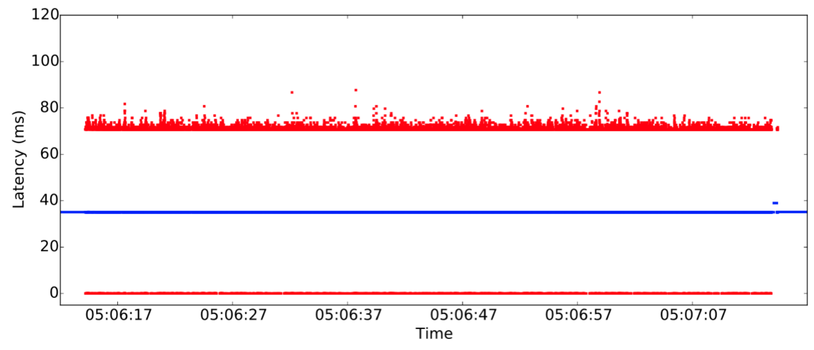

Figure 3: Single-path routing message profile for 60-second problem on NYC → LAX flow.

| Performance (# Messages) | |
|---|---|
| Delivered | 49,203 |
| Late | 10,809 |
| Dropped | 4,979 |
| Disconnected | 0 |
| **Fastest Paths Taken (# Messages)** | |
| NYC → WAS → DFW → LAX | 59,372 |
| NYC → WAS → LAX | 378 |
| NYC → CHI → DEN → LAX | 262 |
| **Cost Metrics** | |
| Edges Per Delivered Message | 3.00 |
| Packets Per On-Time Message | 4.28 |

Table 1: Single-path routing simulator summary metrics for 60-second problem on NYC → LAX flow.

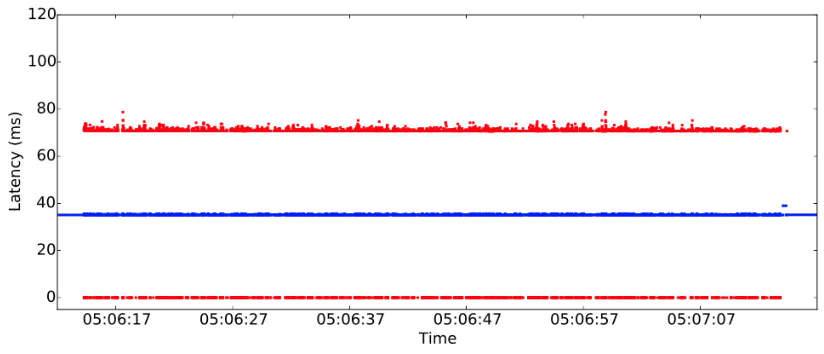

Figure 4: Single-path routing with redundant sending message profile for 60-second problem on NYC → LAX flow.

| Performance (# Packets) | |
|---|---|
| Delivered | 59,454 |
| Late | 3,593 |
| Dropped | 1,944 |
| Disconnected | 0 |
| **Fastest Paths Taken** | |
| NYC → WAS → DFW → LAX | 62,407 |
| NYC → WAS → LAX | 378 |
| NYC → CHI → DEN → LAX | 262 |
| **Average Cost Metrics** | |
| Edges Per Delivered Message | 3.00 |
| Packets Per On-Time Message | 6.64 |

Table 2: Single-path routing with redundant sending simulator summary metrics for 60-second problem on NYC → LAX flow.

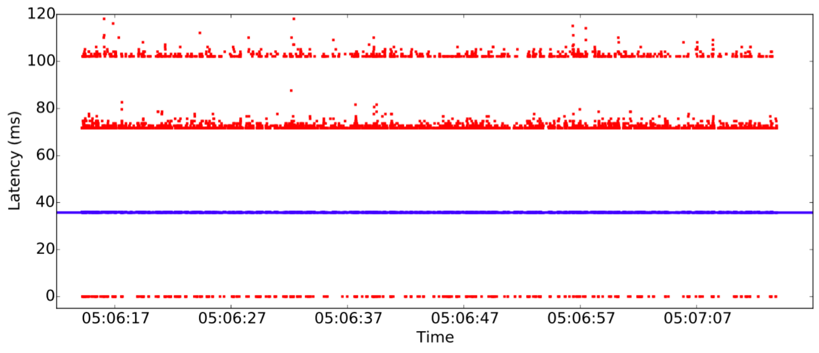

Figure 5: Static two-path routing message profile for 60-second problem on NYC → LAX flow.

| Performance (# Packets) | |
|---|---|
| Delivered | 60,575 |
| Late | 3,978 |
| Dropped | 432 |
| Disconnected | 6 |
| **Fastest Paths Taken** | |
| NYC → CHI → DFW → LAX | 15274 |
| NYC → WAS → LAX | 49279 |
| **Average Cost Metrics** | |
| Edges Per Delivered Message | 5.00 |
| Packets Per On-Time Message | 5.90 |

Table 3: Static two-path routing simulator summary metrics for 60-second problem on NYC → LAX flow.

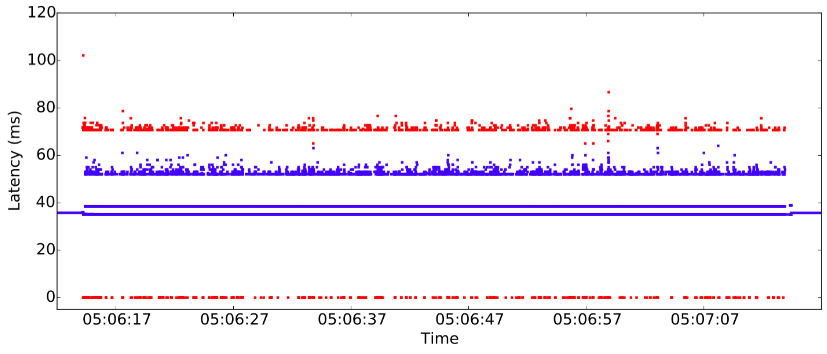

Figure 6: Dynamic two-paths routing message profile for 60-second problem on NYC → LAX flow.

| Performance (# Messages) | |
|---|---|
| Delivered | 63,328 |
| Late | 1,169 |
| Dropped | 494 |
| Disconnected | 0 |
| **Fastest Paths Taken (# Messages)** | |
| `NYC → WAS → DFW → LAX` | 45596 |
| `NYC → SJC → HKG → LAX` | 25 |
| `NYC → CHI → DFW → LAX` | 35 |
| `NYC → WAS → LAX` | 4714 |
| `NYC → SJC → LAX` | 14117 |
| `NYC → CHI → DEN → LAX` | 10 |
| **Cost Metrics** | |
| Edges Per Delivered Message | 5.01 |
| Packets Per On-Time Message | 5.64 |

Table 4: Dynamic twp-paths routing simulator summary metrics for 60-second problem on NYC → LAX flow.

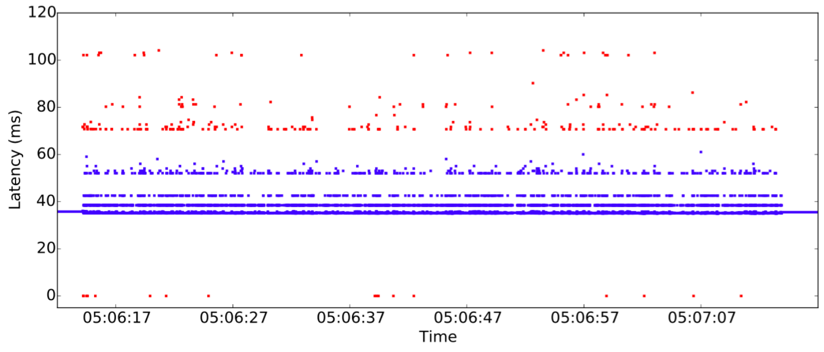

Figure 7: Targeted-redundancy dissemination graph routing message profile for 60-second problem on NYC → LAX flow.

| Performance (# Packets) | |
|---|---|
| Delivered | 64,699 |
| Late | 275 |
| Dropped | 16 |
| Disconnected | 1 |
| **Fastest Paths Taken** | |
| `NYC → WAS → DFW → LAX` | 45,039 |
| `NYC → WAS → DFW → DEN → LAX` | 558 |
| `NYC → CHI → DFW → LAX` | 35 |
| `NYC → WAS → LAX` | 15,727 |
| `NYC → SJC → LAX` | 3,615 |
| **Average Cost Metrics** | |
| Edges Per Delivered Message | 7.80 |
| Packets Per On-Time Message | 9.08 |

Table 5: Targeted-redundancy dissemination graph routing simulator summary metrics for 60-second problem on NYC → LAX flow.

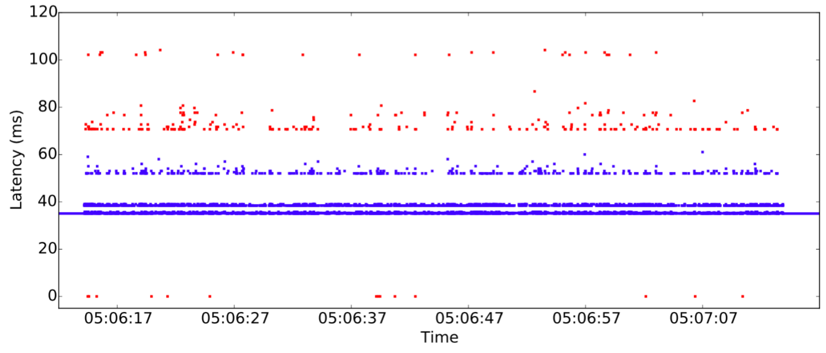

Figure 8: Time-constrained flooding message profile for 60-second problem on NYC → LAX flow.

| Performance (# Packets) | |
|---|---|
| Delivered | 64,705 |
| Late | 273 |
| Dropped | 13 |
| Disconnected | 0 |
| **Fastest Paths Taken** | |
| `NYC → WAS → DFW → LAX` | 49,402 |
| `NYC → WAS → LAX` | 11,397 |
| `NYC → SJC → LAX` | 3,619 |
| `NYC → CHI → DEN → LAX` | 559 |
| **Average Cost Metrics** | |
| Edges Per Delivered Message | 33.00 |
| Packets Per On-Time Message | 34.40 |

Table 6: Time-constrained flooding simulator summary metrics for 60-second problem on NYC → LAX flow.

## 5 Running the Playback Network Simulator

As depicted in Figure 1, there are many programs that comprise Playback. A library API may eventually be included to call these programs, but for now, an example bash script that calls the simulation programs in a valid order can be found in `example_run_pipeline.sh`. First, raw logs are collected using `collect_cloud_data.c`. This script then splits the raw logs in into 10-minute snapshots for the requested range and runs a simulation of 6 approaches, as seen in Figures 3 through 8 and described in Section 4. It also runs two sets of recovery parameters on all approaches: no recoveries, and $m = 1$, $n = 1$. The single-path routing algorithms additionally run recovery with $m = 3$, $n = 3$. All of these approaches are run on 16 source-destination pairs. A user might change the source-destination pairs, routing protocols, or recovery protocols to be run.

Additional materials that are needed for Playback include a *network configuration* file containing the 'normal' round-trip latencies for every link in the overlay network. These normal latencies are learned by user measurement. A *multigraph file* is used to specify many flows (including their source, destination, routing, redundant per-link sending, and recovery protocols) to run on a given set of snapshots, although a (more limited) command-line specification of the routing and flows can also be used. The `example_run_pipeline.sh` script generates this multigraph file for the six approaches discussed in Section 4.

In the simulations we conducted for our work in [6], we collected data over four weeks across four months using a sending rate of 10ms for a network of 12 overlay nodes (with 64 total directed links) in North America, Europe, and East Asia. In this case, each week's logs took up about 600G on disk. We deployed 6 instances on each of 8 six-core machines and ran different portions of the week in parallel. The week was split into 1008 10-minute snapshots, and each 10-minute snapshot contained 224 simulations repeated over the 10 minutes: six approaches were run for 16 source-destination pairs, where each approaches was run with two different sets of recovery parameters, and two approaches were run with an additional set of recovery parameters. This is the set of flows run by `example_run_pipeline.sh`. To run all 224 simulations for every 10-minute chunk in the week with this configuration took approximately 36 hours.

Some scripts useful for presenting results are additionally included. The `compare_6runs.py` script generates a csv containing a column for each dissemination graph method and a row for every snapshot, presenting the summary metrics for each flow in the corresponding entry. The `plot_data.py` script generates plots of the type seen in Figures 3 through 8 from the `flow_simulator.c` program output when it is run with the `-v` option. Finally, `process_flow.py` generates a list of network problems (for example, the case study in Section 4 is considered one network problem) and compares them across all approaches, splitting the lost packets into unavailable time (if the loss rate was over 50% over a period of at least 1 second) and reliability over the time the flow was available. These scripts can be modified to fit the needs of the user.

## 6 Conclusion

We have introduced the Playback Network Simulator, which collects fine-grained per link data on any overlay network and simulates a variety of routing protocols for arbitrary sending rates and source-destination flows in the network using the captured packets, in order to closely match the behavior the flow would have experience at the specific time the data was recorded.

## References

[1] Comparison of core network emulation platforms. In *Proceedings of IEEE MILCOM Conference*, pages 864–869, 2010.

[2] The ns-3 network simulator. `http://nsnam.org`, 2011–2015.

[3] The line network emulator. `http://wiki.epfl.ch/line/documents/line.html`, 2014.

[4] Y. Amir, C. Danilov, S. Goose, D. Hedqvist, and A. Terzis. An overlay architecture for high-quality VoIP streams. *IEEE Transactions on Multimedia*, 8(6):1250–1262, Dec 2006.

[5] Y. Amir, J. Stanton, J.W. Lane, and J.L. Schultz. System and method for recovery of packets in overlay networks, May 2013.

[6] Amy Babay, Emily Wagner, Michael Dinitz, and Yair Amir. Timely, Reliable, and Cost-effective Internet Transport Service using Dissemination Graphs. Technical report, Johns Hopkins University, 2017.

[7] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[8] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[9] LTN Global Communications. LTN Global Communications. `http://www.ltnglobal.com`. Retrieved April 7, 2015.

[10] Pedro Garca Lpez, Carles Pairot Gavald, Rubn Mondjar Andreu, and Jordi Pujol Ahull. Planetsim. `https://github.com/jpahullo/planetsim`, 2005.

[11] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[12] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[13] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pages 255–270, Boston, MA, December 2002. USENIXASSOC.