



- FINAL -

**Architecture, Protocols, Design and Use Cases
for the
Generalized Network Assisted Transport (GNAT)
part of the DARPA Dispersed Computing Program**

Contract No.: HR0011-17-C-0050

Prepared By:
Raytheon BBN Technologies Corp.
10 Moulton St.
Cambridge MA 02138-1119

Prepared For:
SPAWAR Systems Center , Pacific
53560 Hull St. Code 56120
San Diego, CA 92152-5000

DISTRIBUTION STATEMENT A: Approved for Public Release, Distribution Unlimited.

This document does not contain technology or technical data controlled under either the U.S. International Traffic in Arms Regulations or the U.S. Export Administration Regulations.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-17-C-0050. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

REVISION HISTORY

Version	Summary of Changes	Date
-01	Draft release of GNAT Architecture Protocols, Design and Use Cases	5-January-2018
-02	Final release of GNAT Architecture Protocols, Design and Use Cases	1-March-2021
-03	Update markings for public release	24-March-2021

APPROVALS

Prepared By: _____ *On file* _____

Steve Zabele
Principal Investigator

Date

Concurrence: _____ *On file* _____

Scott Loos
Program Manager

Date

Concurrence: _____ *On file* _____

Maurice Foley
QA/CM

Date

CONTENTS

1	Project Overview	7
1.1	Problem Description	7
1.2	Approach and Innovations	7
2	Architecture, Protocols, and Design	10
2.1	Introduction	10
2.2	Backpressure Forwarders (BPFs)	11
2.3	Capacity Adaptive Tunnels (CATs)	13
2.4	Protocol Proxies	15
2.4.1	UDP Proxy (including best effort, semi-reliable multicast, and reliable multicast)	16
2.4.2	TCP Proxy	16
2.5	Admission Planner (AMP)	17
3	Backpressure Forwarder	18
3.1	Summary of Module Goals and Objectives	18
3.2	High Level Design	18
3.2.1	Adjacent Modules and Components	20
3.2.2	Input Sources and Information Needed	20
3.2.3	Output Destinations and Information Provided	21
3.2.4	Events	22
3.2.5	Configurable Parameters	23
3.2.6	Performance Monitoring Statistics	24
3.3	Detailed Design	24
3.3.1	Bin Identifiers (Bin IDs)	24
3.3.2	Bin Maps	25
3.3.3	Bin Indexes	25
3.3.4	Destination Vectors	26
3.3.5	Multicasting within GNAT	27
3.3.6	Queues and Queue Lengths	28
3.3.7	Queue Length Advertisement Message (QLAM) Packet Format	33
3.3.8	End-to-end Latency Estimation, Time-to-Go Tracking, and Impeding Packet Circulation	35
3.3.9	Backpressure Forwarding Methods	38
3.3.10	Hierarchical Forwarding	50
3.3.11	Known Limitation: Partial Starvation of Traffic Generated by a Source of Latency-Heterogeneous Flows	
51		
3.3.12	Queue Depth Smoothing	51
3.3.13	ASAP: Adaptive Starvation Avoidance Protocol	53
3.3.14	No Packet Left Behind (NPLB)	55
3.3.15	Zombie Latency Reduction (ZLR)	57
4	UDP Proxy	63
4.1	Summary of Module Goals and Objectives	63
4.2	High Level Design	63
4.2.1	Adjacent Modules and Components	63
4.2.2	Input Sources and Information Needed	64
4.2.3	Output Destinations and Information Provided	64
4.2.4	Events	64
4.2.5	Use Cases	65
4.2.6	Performance Monitoring Statistics	66
4.3	Detailed Design	67
4.3.1	Sender Side Processing	67
4.3.2	Receiver Side Processing	81
4.3.3	Internal State Management	85
4.3.4	Interprocess Communications	86
4.3.5	UDP Proxy Configuration	86
5	TCP Proxy	87
5.1	Summary of Module Goals and Objectives	87
5.2	High Level Design	88
5.2.1	Adjacent Modules and Components	88
5.2.2	Input Sources and Information Needed	88
5.2.3	Output Destinations and Information Provided	89
5.2.4	Events	89
5.2.5	Performance Monitoring Statistics	91
5.3	Detailed Design	92

5.3.1	TCP Proxy Overview	92
5.3.2	Getting Application Packets Into the TCP Proxy.....	92
5.3.3	Error Control	93
5.3.4	Admission Control.....	94
5.3.5	Flow Termination State Transitions	96
5.3.6	Self-Adapting Buffer Sizes.....	100
5.3.7	Controlling the Advertised Window Size	100
6	Shared Memory IPC.....	102
6.1	Overview.....	102
6.1.1	Rationale	102
6.1.2	Architecture	102
6.2	Shared Objects	103
6.2.1	Queue Depths	103
6.2.2	Packets.....	103
6.3	Signaling Mechanism.....	104
6.4	Events.....	104
7	Capacity Adaptive Tunnels (CATs)	105
7.1	Summary of Module Goals and Objectives	105
7.2	High Level Design.....	105
7.2.1	Adjacent Modules and Components	105
7.2.2	Input Sources and Information Needed	106
7.2.3	Output Destinations and Information Provided	108
7.2.4	Events.....	109
7.2.5	Configurable Parameters	111
7.2.6	Performance Monitoring Statistics	111
7.3	Detailed Design	112
7.3.1	SLIQ Protocol	112
7.3.2	Congestion Control Algorithms	155
7.3.3	SLIQ CAT	205
7.3.4	Combining Congestion Control Algorithms	211
8	Admission Planner	217
8.1	Summary of Module Goals and Objectives	217
8.2	High Level Design.....	217
8.2.1	Adjacent Modules and Components	217
8.2.2	Use Cases	217
8.2.3	Input Sources and Information Needed	218
8.2.4	Output Destinations and Information Provided	219
8.2.5	Performance Monitoring Statistics	220
8.3	Detailed Design	220
9	Multicast Group Management Sniffer (MGMS)	225
9.1	Summary of Goals and Objectives	225
9.2	High Level Design.....	225
9.3	Detailed Design	226
9.3.1	Receiving Multicast Group Management Packets	226
9.3.2	Processing Received Multicast Group Management Packets	226
9.3.3	Multicast Group Membership Cache.....	226
10	Analysis Supporting GNAT Design Points.....	228
10.1	MatLab Model for Assessing Latency-constrained Error Control Trades.....	228
10.1.1	Problem Formulation	228
10.1.2	Coded ARQ	228
10.1.3	Analysis	228
10.2	Analytical Basis for GNAT's Latency-Constrained Adaptive Error Control (AEC) Algorithm.....	232
10.2.1	Overview.....	232
10.2.2	Basic Error Control Model.....	233
10.2.3	Mathematical Basis for AEC	234
10.2.4	Midgame and Endgame Run-time/Implementation Details.....	241
10.2.5	Simple Example Showing AEC Processing	241
10.2.6	Analytical Comparison of AEC Performance with the (near) Optimal Solution	243
11	Use Cases.....	244
11.1	A Day in the Life of a Packet	244
11.2	Response to a Link Failure	245
11.3	Distributed Dynamic Load Balancing of Multicast Traffic	246

12	Support Tools.....	248
12.1	LinkEm.....	248
12.1.1	LinkEm Basics	248
12.1.2	Controlling LinkEm Using LinkEmClient	249
12.1.3	Modeling Concurrent Parallel Networks.....	250
12.1.4	Modeling Multiple Independent Paths.....	250
12.1.5	Modeling Access Links	252
12.1.6	Configuring LinkEm	252
12.1.7	Starting LinkEm	256
12.1.8	Choosing LinkEm Buffer Sizes	257
12.2	NACK-Oriented Reliable Multicast (NORM)-based File Transfer Protocol (<i>nftp</i>).....	260
12.2.1	Summary of Goals and Objectives	260
12.2.2	High Level Design.....	260
12.2.3	Detailed Design	261
12.2.4	<i>nftp</i> Receiver Packet Filtering	263
12.2.5	NORM Enhancements.....	264
12.2.6	Configurable Parameters	266
12.3	GNAT-enabled Secure Copy (scp) Application.....	269
12.3.1	Assumptions:	269
12.3.2	Usage:	269
13	Creating and Running GNAT Experiments.....	270
13.1	Preface	270
13.2	Test Network Emulation.....	270
13.3	Testbeds.....	272
13.3.1	Configuring Testbed Nodes	273
13.4	Testbed Abstraction.....	278
13.4.1	Generic Terminology For Testbeds	278
13.4.2	Testbed Topology Files	279
13.5	Experiment Configuration Templates.....	282
13.6	Creating A New Experiment	283
13.6.1	Define Experiment	283
13.6.2	Generate Experiment Configuration Files.....	284
13.6.3	Optionally Modifying the GNAT Network "Plumbing"	287
13.6.4	Finalize Experiment Details	288
13.7	Preparing Experimenter's Run-Time Environment.....	291
13.7.1	Shell (login environment) Set-up	291
13.7.2	Experimenter ssh Configuration	292
13.8	Reserving Testbed Nodes	294
13.9	Experiment Execution.....	296
13.9.1	Experiment Overview.....	296
13.9.2	Running an Experiment	297
13.10	Common Debugging Techniques	302
13.10.1	Experiment Logging	302
13.10.2	Experiment Packet Captures	302
14	Installing and Configuring PIM-SM on a GNAT Testbed Host.....	304
14.1	Step-by-step guide.....	304
14.2	Configuration Note	305
14.3	Testing Installation.....	306
15	Setting Up and Running USC's Jupiter	307
15.1	Execution Environment.....	307
15.2	Clone Jupiter Repository	307
15.3	Installing Kubernetes and Required Components	307
15.3.1	Install Kubernetes	307
15.3.2	Install pip3	308
15.3.3	Install Required Python Packages	308
15.3.4	Install Kubernetes Tools	308
15.3.5	Install Minikube	308
15.3.6	Docker	309
15.4	Setting Up Kubernetes Cluster	310
15.4.1	Kubernetes Master Node	310
15.4.2	Kubernetes Worker Nodes	315
15.5	Setup Local Private Docker Registry	316

15.5.1 Registry Configuration and Execution.....	316
15.5.2 Examining Registry Contents	317
15.5.3 Registry Control	318
15.6 Configuring and Deploying Jupiter.....	318
15.6.1 Verifying Jupiter Deployment.....	319
15.7 Some Useful Debugging Techniques	321
15.7.1 kubelet logs	321
15.7.2 Kubernetes Dashboard.....	321
15.7.3 Viewing Pod Logs	324
15.8 Teardown.....	325
15.8.1 Jupiter Teardown	325
15.8.2 Kubernetes Cluster Teardown	325

Preface

The overall goal for GNAT (for *Generalized Network Assisted Transport*) is improving the performance of applications that must exchange information over wide-area networks (WANs), with a special emphasis on improving the performance of multicast applications that are latency sensitive. Our core approach leverages fully distributed computation both within the network and at the edges that collectively works to continually maximize Cumulative Network Utility (CNU). GNAT's CNU optimization approach starts with a robust, low-overhead, distributed optimization approach known as *Backpressure* that is known to be throughput optimal. GNAT extends Backpressure by taking into account per-packet end-to-end latency requirements, with the goal of creating a system that is throughput optimal subject to latency constraints. GNAT completes the picture by adding a host of latency reduction techniques including latency sensitive hop-by-hop error control, congestion control, and a set of queuing delay reduction techniques to make end-to-end delays across the system as small as possible.

Backpressure consists of a pair of complementary techniques: a packet-forwarding algorithm that leverages relative queue differentials between neighbors, and a suite of admission control algorithms that regulate application traffic access to the backpressure forwarded network. The combined techniques work to continuously maximize network utility (and hence CNU) without requiring global knowledge of traffic pattern, packet priorities or network topology. Each GNAT node performs a local optimization that, in cooperation with its peers, yields a global maximization of CNU.

1 Project Overview

1.1 Problem Description

Over 80% of military tactical communications is point-to-multipoint – i.e., *multicast*. Situation awareness, push-to-talk, training simulations, etc. all require low-latency communications between multiple entities. For example, discussions with the Naval Airborne Networking Community identified the need for a capability that could make multicast forwarding decisions at each network node in terms of which path(s) to use in forwarding any given packet based on dynamic load and dynamic link conditions, subject to delivery latency constraints. Moreover, since demand generally exceeds capacity in tactical networks, the overarching goal was delivering the most useful data possible.

While solutions providing distributed, in-network throughput optimization for reliable multicast have advanced considerably over the last 20 years, technologies that support distributed dynamic load balancing of multicast traffic *subject to latency constraints* have not. Per the above discussion, the Navy’s best approach involved using Multi-Protocol Label Switching (MPLS)-like mechanisms to pre-provision forwarding paths based on long term estimates of traffic load. An MPLS approach works if network loads are well below the available capacity such that there are no queueing delays; however, this is an unaffordable luxury in a tactical environment where capacity is very limited. Moreover, unpredictable loading can easily leave some communications resources unused with others oversubscribed. *The goal for GNAT is addressing this need.*

The training community has very similar needs, especially given the advent of multi-participant Live-Virtual-Constructive (LVC) simulation. In particular, the tight latency constraints and the need to support multiple concurrent participants in augmented reality applications are also currently accommodated by over provisioning. GNAT’s multi-source, multi-destination probabilistic-reliable, latency-sensitive utility-driven multicast transport capability *will improve throughput while meeting latency constraints, making LVC simulation scalable and more easily distributed.*

1.2 Approach and Innovations

Absent latency constraints, multicast networking is solved (in the academic literature) since maximizing throughput is a convex optimization problem. The presence of a latency constraint turns this relatively tractable problem into a highly non-convex optimization problem. Developing a *distributed* solution to this problem requires multiple innovations.

GNAT Innovations – The upper-left corner of Figure 1 illustrates the problems with existing reliable multicast protocols: when packets are lost due to congestion or bit errors, end-to-end error recovery takes too long for latency sensitive applications and can also be quite inefficient, especially if losses occur near the receiver. Several key GNAT innovations are summarized in the remaining portions of Figure 1:

- Hop-by-hop recovery (upper right) automatically optimizes response to loss – either reactively re-transmitting packets or proactively coding for channel loss – to provide a quick response and to reduce network load.

- In the presence of competing traffic, GNAT makes *forwarding decisions that maximize the value of received information* (lower left) based on estimates of utility at the receiver.
- GNAT uses *distributed dynamic load balancing* (lower right) to leverage all resources to provide the highest level of service possible.
- Other innovations include: the *use of a probabilistic reliability model* to explicitly capture the receiver's maximum acceptable loss rate; *selectively enabling GNAT on a subset of Network Computation Points (NCPs)* to reduce total compute load and reduce slow-path latency; *use of heterogeneous, temporally based utility functions* capturing each receiver's required update rate; and *use of multicast meshes to improve resilience and responsiveness to link availability and degradation*.

GNAT Provides a Generalized Solution – As proper subsets of multicast, GNAT supports broadcast and unicast. GNAT provides a wide range of services. For example, GNAT provides RTP-like service by setting the allowable loss to 10% and the latency bound to 150 milliseconds; it provides a TCP-like service by setting the allowable loss to 0% and the latency bound to a few seconds. GNAT subsumes a wide range of existing special-case services and so is a generalized solution with broad applicability.

GNAT Innovations Build upon Our Prior and Ongoing Work – GNAT leverages the significant government investments that has been performed by members of our team over the last twenty years. Leveraging this research and development allowed us to quickly and efficiently address the fundamental research challenges associated with optimizing network multicast performance *subject to latency constraints*. In particular, we leveraged the following experience and capabilities.

- The Active Error Recovery (AER) Nominee-based Congestion Avoidance (NCA) reliable multicast protocol created by members of our team for DARPA's Active Networks program, which demonstrated that significant gains in throughput and reduction in latency can be achieved using in-network hop-by-hop techniques.
- The use of dynamically updated utility functions to reduce forwarding of packets with no net utility to individual multicast receivers, which was demonstrated by members of our team under DARPA's Active Networks program.

and most importantly

- The use of hop-by-hop utility-based load balancing using backpressure techniques in support of latency-sensitive and latency-tolerant unicast, developed by us under DARPA's EdgeCT program.

In particular, the Intrinsically Resilient Overlay Network (IRON) capability developed by us under the EdgeCT program provides a baseline architecture, to which we have added a set of non-trivial extensions for the GNAT project.

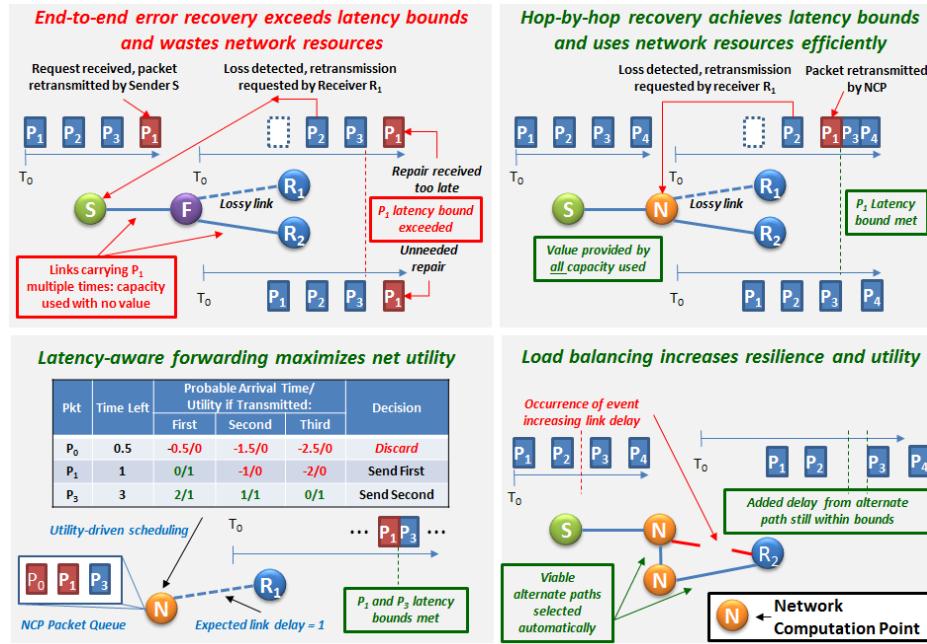


Figure 1 . GNAT optimizes the utility of delivered information in a dynamic multicast environment, where utility includes per-receiver bounds on delivery latency

2 Architecture, Protocols, and Design

2.1 Introduction

GNAT's goal is improving the performance of applications that exchange information over a wide-area network (WAN), with a particular emphasis on latency-sensitive, multicast-based applications.

As shown in Figure 2, our approach leverages available computation both at the edges of the WAN and within the WAN itself. Each GNAT node within the interior of the WAN contains a basic forwarding engine, here shown as a backpressure forwarder, which exchanges information with other backpressure forwarders in neighboring GNAT nodes, to affect latency-constrained, utility-based throughput optimization. Since GNAT nodes may not, and in general will not be located at every route point within the WAN, GNAT node pairs are interconnected via tunnels, here shown as Capacity Adaptive Tunnels (CATs). CATs provide basic latency-constrained hop-by-hop error control and low-latency hop-by-hop congestion control, and in the process estimate quantities such as available link/path capacity, loss rate, and delay which are used by other GNAT components for making forwarding decisions. In particular, CATs allow GNAT to operate over paths with very high loss rates ($>40\%$ Packet Error Rates), which in combination with backpressure's use of multiple pathways enables GNAT to maximally leverage all available network resources.

GNAT nodes located at the edges of the network also include proxies that acts as "impedance matchers" between user applications and the GNAT-enabled network. Among other aspects, these proxies contain admission controllers (which manage the rate at which packets from each flow are allowed into the network) and end-to-end error controllers (e.g., TCP-like retransmissions or forward error correction). The proxies also provide other services such as packet reordering and jitter reduction at the receivers to make GNAT's multipath-forwarded tenable to end applications.

A GNAT node can be thought of as a set of collaborating network control subsystems that each perform a sequence of analyze, decide, and adaptation steps. CATs *analyze* the available capacity along their assigned path, *decide* how to allocate tunnel capacity including the required level of error control, and dynamically *adapts* the tunnels to affect those decisions. Backpressure Forwarders (BPFs) implements an integrated *analyze*, *decide*, *adapt* loop wherein BPFs exchange queue depth information with neighboring BPFs to determine which packets are forwarded along which path and when. Admission controllers *analyze* local queue lengths, leverage utility functions to *decide* how much of each flow should be admitted to the network at any given time, and *adapt* flow controllers that implement these decisions. The Admission Planner (AMP) *analyzes* resource requirements across the set of local application flows, dynamically *decides* how to apportion resources between flows, and affects these decisions by adapting individual utility functions used by the application flow controllers

To provide insight into the GNAT architecture and its operation, we start with components closest to the network and operating at the fastest times scales, and work upward towards supervisory control running at the slowest time scales.

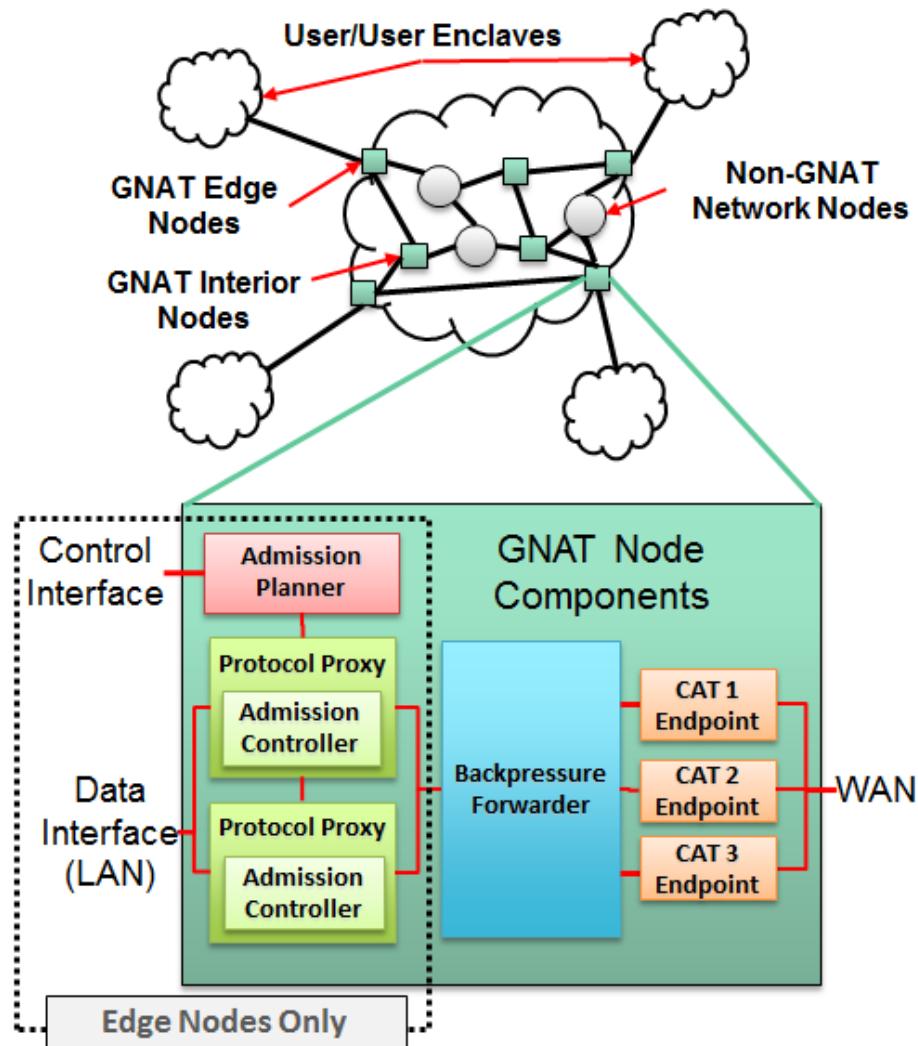


Figure 2. High level GNAT Architecture

2.2 Backpressure Forwarders (BPFs)

BPFs provide load balancing between traffic to different destinations including traffic being sent to multiple destinations in the form of multicast. Each BPF consists of:

1. N-1 random access queues for storing and retrieving packets both from the local enclave and from other enclaves for which the local node is acting as a forwarder (N is the number of GNAT nodes in the network);
2. A cache of sets of queue depths, one from each neighboring GNAT node; one for each active unicast destination and one for each active multicast group; and
3. A packet scheduler that decides which packets are forwarded to which other neighboring GNAT nodes, and when

The backpressure forwarder embodies GNAT's essential adaptive load balancing characteristics, in that in response to events (e.g., the arrival of new packets, updates to queue depth information from adjacent nodes) it selects a packet from one of its per-destination queues and forwards over an appropriate interface, represented here by the set of Capacity Adaptive Tunnels (CATs)

In a broad sense, backpressure forwarding relies on two simple rules.

1. A node never forwards packets to a neighbor whose queue depth to the destination is longer than its own. Congestion “backs up” toward the source until a less congested alternative path is found – hence the name “backpressure”.
2. When there is an opportunity to forward a packet to a neighbor, the packet is chosen from the destination queue with the largest relative difference (the local node’s queue minus neighbor’s queue). If multiple CATs are idle, the packet is transmitted over the CAT with the largest gradient to the destination.

The difference between queue lengths can be interpreted as a gradient: hence packets are forwarded “downhill” via nodes which have shorter and shorter queue lengths until they reach the destination. Figure 3 illustrates how packets injected at the source (node **s**) make their way to the destination (node **d**). Node **s** is able to send to forwarder node **f₁** when its queue for destination node **d** is longer, and so on.

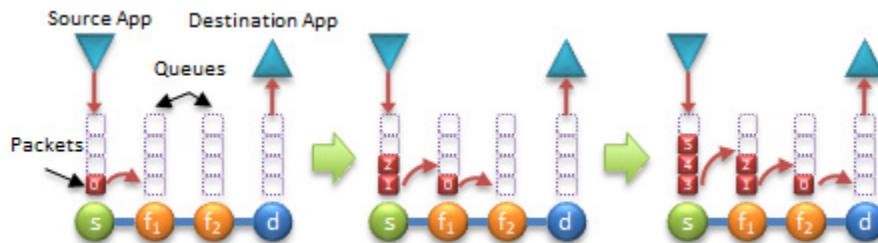


Figure 3. Backpressure works by forwarding packets from more congested (longer queues) to the less congested (shortest queues) neighbors.

Backpressure uses multiple paths to forward traffic to destinations without requiring explicit information about traffic flows, topology or link capacities. The backpressure forwarding algorithm automatically and rapidly adapts to changes in these factors based solely on the exchange of queue length information. Hence each Load Balancing Controller is responsible for sending its own queue depth information to Load Balancing Controllers on adjacent GNAT nodes, as well as receiving and caching reports of queue depth information from adjacent GNAT nodes.

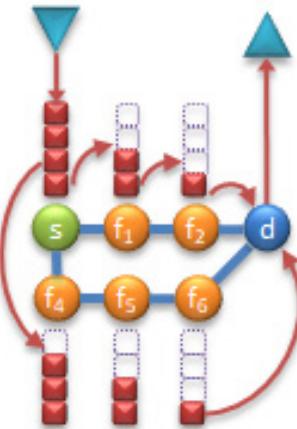


Figure 4. Backpressure's simple forwarding rules allow leveraging multiple paths to improve throughput.

While the above examples illustrate GNAT behavior with a unicast flow, support for multicast is largely a vector extension of basic backpressure forwarding. At a high level, the queue differential for each member of a multicast group – i.e., each GNAT destination node – are computed separately and then summed, with contributions for members where the queue differentials are negative (i.e., a neighbor has more packets enqueued for a given member than the local node) set to zero. This is repeated for each multicast group with each neighbor. The multicast group with the greatest summed queue differential across all neighbors is selected and a packet for that multicast group is forwarded to the selected neighbor, marked for delivery to only those group members with positive queue differentials. The details here are somewhat more involved, but the core backpressure forwarding mechanism for multicast is consistent with the unicast model.

2.3 Capacity Adaptive Tunnels (CATs)

CATs allow backpressure to work in an overlay environment, where shared bottlenecks, substantial amounts of cross traffic and network events such as Denial of Service attacks and link outages are otherwise problematic. CATs estimate the available capacity over the path between two backpressure forwarder nodes and control transmission rates to prevent congestion losses within the network. CAT operations are TCP-like in that no explicit coordination or joint estimation is necessary for capacity sharing over shared bottlenecks. The CATs also provide per-application adjustable reliability to improve performance and efficiency over the tunnel.

To understand the role played by *tunneling* within the CATs, consider the configuration in Figure 5, with standard (i.e., non-GNAT equipped) routers shown in purple. GNAT's base assumption is that it cannot configure or control these standard routers within the network, and hence must work either with or around a large part of an internetworked infrastructure. In this example, with conventional shortest path routing the route from the source to the destination traverses the reduced capacity link shown as a dotted red line. By using tunnels, GNAT can instead deliver traffic to the destination by first sending it to an intermediate forwarder (shown in orange) which then delivers it to the destination. Doing so admits a path that has 10 times the capacity of the direct path from the source to the destination, and hence throughput is improved by a factor of 10.

As a side note, in this simple example backpressure forwarding will use *both* paths, with a net throughput of 11 Mbps vs 1 Mbps achievable with standard routing.

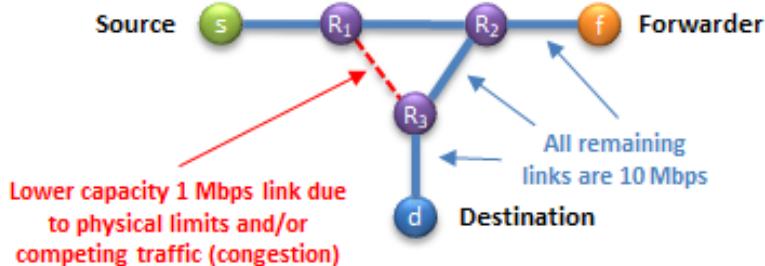


Figure 5. Backpressure's simple forwarding rules allow leveraging multiple alternative paths to improve throughput compared with standard shortest path routing

To understand the role played by *congestion control* within the CATs, consider the following, from the backpressure forwarding discussion, forwarding along a given path involves setting up a positive queue differential along the entire path. Along the direct path, the Destination always advertises a queue depth of 0 since it immediately consumes any packets delivered to it, hence the Source can forward to the destination over the lower capacity path as soon as it has at least one packet in its queue. In contrast, along the indirect path, since the Destination has a queue length of zero, the Forwarder requires at least one packet in its queue, and so the Source requires at least *two* packets in its queue. As a result, absent inputs from the CATs, the backpressure forwarder at the Source would always select the direct lower path.

Now the problem becomes apparent: if the Source always chooses to send only via the direct path, any load in excess of 1 Mbps will get dropped at the router R₁, due to the 1 Mbps link from R₁ to R₂. Hence a 10 Mbps load offered at the source can easily result in only a 1 Mbps throughput as measured at the Destination. Having queue depth information only at overlay node positions therefore does not provide sufficient information for backpressure to work correctly. Ideally all of the nodes in the topology, especially R₁, would provide queue depth information to avoid overdriving lower capacity links; however, when operations are restricted entirely to overlay nodes, an alternate method is required for regulating transmission rates on overlay paths.

Fortunately a large body of work on TCP congestion control algorithms has focused on finding and using the available capacity in a dynamic setting, as well as sharing capacity fairly between applications that do not explicitly exchange information. By leveraging these techniques, CATs are able to solve both the resource estimation and distributed resource sharing problem, and as a result become TCP-like in their behavior. CATs provide feedback to their local backpressure forwarder in the form of transmit opportunities, which acts as another form of congestion signal to be used for traffic forwarding decisions. At any given time, backpressure forwarders then only consider neighbors for which there is an available transmit opportunity.

2.4 Protocol Proxies

Protocol proxies insulate host applications (which expect internet-like networking environments) from the consequences of GNAT's prioritized, rate-regulated, multipath forwarded, overlay network with its wider variability in round trip times, substantial levels of packet reordering and so forth. Each instance provides source-to-destination enclave error control as needed, and performs packet-level, per-flow **Admission Control** (AC). Admission control plays a critical role within the GNAT architecture in that it regulates use of the backpressure-forwarded network by individual applications. Backpressure forwarding is known to be throughput optimal whenever the combined rates of the application traffic are feasible – that is, when there exists *some* set of paths that support the collective set of offered application loads without congesting links. However, whenever demands exceed the resources available such that the traffic is not feasible, performance can be significantly degraded, and in fact in certain scenarios it's possible that *none* of the applications may achieve satisfactory transfer rates. Admission Control manages the demands placed on the network by each application to restore feasibility and do so in a way that maximizes Cumulative Network Utility.

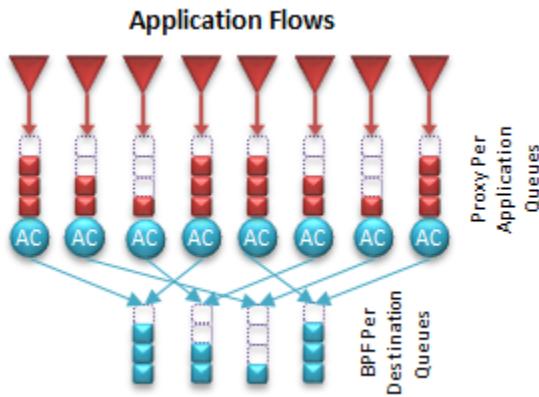


Figure 6. Application proxies, containing embedded admission controller instances, one for each flow leverage local queue depth information to regulate flow rates into the network

In general, the set of Admission Controllers collectively work to maximize the weighted Cumulative Network Utility $\sum_i w_i U_i(r_i)$ where $U_i(r_i)$ is the utility function for flow i at rate r_i and w_i is a weight that indicates the relative importance or priority of the flow. Admission control algorithms exploit the fact that queue lengths are proportional to the cost of sending traffic into the network, allowing each controller to locally maximize its *net utility* (the utility of sending minus the cost of sending) which is given by $\max_{r_i} w_i U_i(r_i) - (\frac{Q_i}{K})r_i$ where Q_i is the local queue length for flow i , and K is a tunable scaling parameter that determines the trade-off between optimality and queue length (which impacts delay). Hence each Admission Controller uses only its local queue lengths to determine how much traffic to let into (i.e., admit to) the network. In this way each Admission Controller works very much the same as backpressure forwarders do in their use of queue depth information as an indication of downstream congestion in making forwarding decisions.

GNAT leverages two distinct types of Protocol Proxies, one for each of the two main types of transport protocols: UDP and TCP. GNAT's TCP and UDP Proxies have similar high level functions and are described below.

2.4.1 UDP Proxy (including best effort, semi-reliable multicast, and reliable multicast)

The GNAT UDP Proxy provides a packet-level admission control capability which buffers out-bound packets and sends them to the local backpressure forwarder based on the packet-level admission control algorithms. Admission control decisions leverage queue depth information from the backpressure forwarder. The primary traffic type handled by the UDP PEP is inelastic traffic where maintaining minimal delivery rates is essential, and reliable delivery is not typically required. The UDP Proxy does, however, also handle elastic traffic such as the Navy Research Laboratory's (NRL's) NACK-Oriented Reliable Multicast (NORM) protocol which is layered on UDP.

The UDP Proxy also handles packet reordering over longer timescales than a single FEC group time. This is critical for masking the types of packet reordering that naturally occurs over multipath routed networks from the applications.

Finally, the UDP Proxy also supports end-to-end error control to offset the impact of loss (due to either congestion or corruption) and/or the long relative delays that can occur in a multipath routed network. Applications using UDP generally are willing to accept some loss in exchange for more timely delivery; hence *Proactive Error Control* in the form of packet level Forward Error Correction (FEC) is appropriate. FEC increases the probability of packet delivery (but does not *guarantee* delivery) without imposing the end-to-end delays associated with retransmission-based error control approaches – at the cost of decreased network efficiency compared to retransmission-based mechanisms. While our UDP Proxy does support end-to-end FEC, GNAT generally relies on hop-by-hop error control within the CATs and so end-to-end losses are small; hence end-to-end FEC is generally not used.

2.4.2 TCP Proxy

As with the UDP proxy, the TCP PEP also provides a packet-level Admission Control capability. The primary traffic types handled by the TCP PEP include both elastic traffic (traffic where delivery rate variation is quite acceptable) and traffic for applications with aggregate delivery deadlines (e.g., file transfers with deadlines, where rate variation is allowable as long as the entire transfer is completed within a specified amount of time).

The TCP Proxy also ensures in-order packet delivery at the receiver, again primarily to mask the types of packet reordering that naturally occurs over multipath routed networks from the applications.

The TCP proxy does support end-to-end error control functions consist of the acknowledgement-based retransmission mechanism basic to TCP appropriately – enhanced to accommodate the

GNAT's unusual network characteristics. Since GNAT's multipath routing and queuing can result in substantial amounts of reordering, the TCP Proxy incorporates a mechanism for dynamically estimating and adapting retransmission scheduling to maximize efficiency.

2.5 Admission Planner (AMP)

AMP plays the role of a supervisory controller, with purview over all flows across both proxies, and has the goal of providing a holistic, cross-flow, cross-proxy admission control capability. Functions include the following basic tasks.

1. It determines whether and when to admit application flows as a whole onto the network. It does this by considering other extant competing application flows, their associated priorities, deadlines or required flow rates, and progress or success estimates provided by their proxy.
2. Based on a decision to admit one or more application flows onto the network, it selects and configures an appropriate utility function for each flow associated with the given application instance, and provides this information to a relevant transport proxy to handle packet-level admission control for the flow.
3. It adjusts utility parameters as needed to maximize cumulative network utility. For example, it may increase the priority of a flow that appears to be falling behind its deadline (e.g., for file transfers with deadlines), or it may terminate a flow if its deadlines are deemed no longer reachable in order to free up resources for other competing flows.

When there is insufficient capacity to support all concurrent inelastic flows, competition between the flows can easily lead to a situation where none of the flows receives sufficient capacity to achieve non-zero utility. To make better use of the available resources, GNAT as an integrated system will prevent admission of some of the flows into the network in order to free up sufficient resources so that other flows can achieve non-zero utility through a process called *triage*.

Triage responsibilities are shared between the proxies and AMP. Notionally, admission controllers within the proxies have purview only over individual flows and act independently for each flow over very short time scales (on the order of packet inter-send times). AMP has purview over all of the flows and will consider triage actions for the set of flows as a whole, acting at longer timescales (e.g., multiple seconds).

As an example, consider that in a typical over-subscribed scenario, the local backpressure queues will start building up, which has the effect of reducing the send rates calculated by each flow's admission controller. When the send rates drop below the minimum rate to achieve non-zero utility for a specified amount of time, the admission controllers within the proxy (using trapezoidal (TRAP) utility functions) will begin triaging flows. Reducing the number of flows reduces the arrival rates at the local backpressure queue, and allows the queues to drain -- which in turn may cause previously triaged flows to restart, causing the queues to build once more. This process, described as thrashing, will repeat and is suboptimal since capacity can be consumed that does not increase utility. By looking across flows AMP is able to recognize thrashing and take steps to prevent it from happening.

3 Backpressure Forwarder

3.1 Summary of Module Goals and Objectives

The main goals of the Backpressure Forwarder (BPF) are:

- Providing adaptive load balancing of outgoing packets over multiple paths
- Forwarding packets without the need for explicit information about traffic flows, topology, or link information yet still make globally optimal forwarding decisions.

3.2 High Level Design

Backpressure Forwarding (located within the BPF) and Admission Control (located within the proxies) combine to provide prioritized, distributed dynamic load balancing and flow rate optimization within the GNAT architecture. Backpressure forwarding involves a simple protocol whereby neighboring nodes exchange queue length information, with the differences between local and neighboring queue lengths used to packet forwarding decisions on a packet-by-packet basis. When combined with Admission Control, the Cumulative Network Utility can be globally optimized without the need for any global knowledge of traffic patterns, packet priorities, or network topology. As shown in Figure 7, each Backpressure Forwarder (BPF) is comprised of three different components: a set of packet queues, a neighbor queue values cache, and a backpressure scheduler.

Multiple random-access packet queues within each BPF are used to store packets destined for other enclaves, and may contain packets both from the local enclave as well as packets forwarded from other enclaves using the local enclave as an intermediate hop. There is a set of queues for each *bin* (more fully described below), where a bin corresponds to either a unicast destination or a multicast destination; a multicast destination consists of the set of unicast destinations that are part of the multicast receiver group. Multiple queues are used for each bin to separate traffic into different traffic types in order to support service differentiation based on traffic type.

The neighbor queue length cache is used to store backpressure queue lengths from GNAT nodes in topologically adjacent GNAT nodes (i.e., overlay neighbors). In actuality the queue length cache contains information that may more aptly be described as queue *values*. Here, values may simply be the neighbor's physical queue lengths, or it may be weighted or scaled versions of the physical queue lengths (if heavyball or AQUA is in use), and may have an added bias term from queuing delay values (if the No Packet Left Behind – NPLB – algorithm is used) and/or additional contributions from the inclusion of *virtual queue lengths* (to assist in forwarding packets under low traffic volume conditions).

The backpressure scheduler compares its local set of queue values against those of its neighbors, selects a bin (either a unicast or a multicast bin) from which to pull packet along with a next-hop neighbor (from the set of neighbors with interfaces having transmit opportunities), pulls one or more packets from the appropriate packet queue associated with that bin, and forwards the

packet(s) to the selected neighbor using the CAT connected to that neighbor. In the base forwarding algorithm, each packet is forwarded to the neighbor with the greatest relative queue value difference for the destination *bin*. It favors packets with expedited forwarding markings and never forwards a packet to a neighbor with a corresponding packet queue value for the destination bin that is larger than its own. In the enhanced forwarding algorithm implementation, the backpressure scheduler selects low-latency packets first to the neighbor whose queue value difference for this latency-class is largest.

The high level BPF architecture is shown in the following Figure 7. Here, black lines represent backpressure queue length advertisement packets, purple lines represent proxied packets, and green lines represent control and information flows.

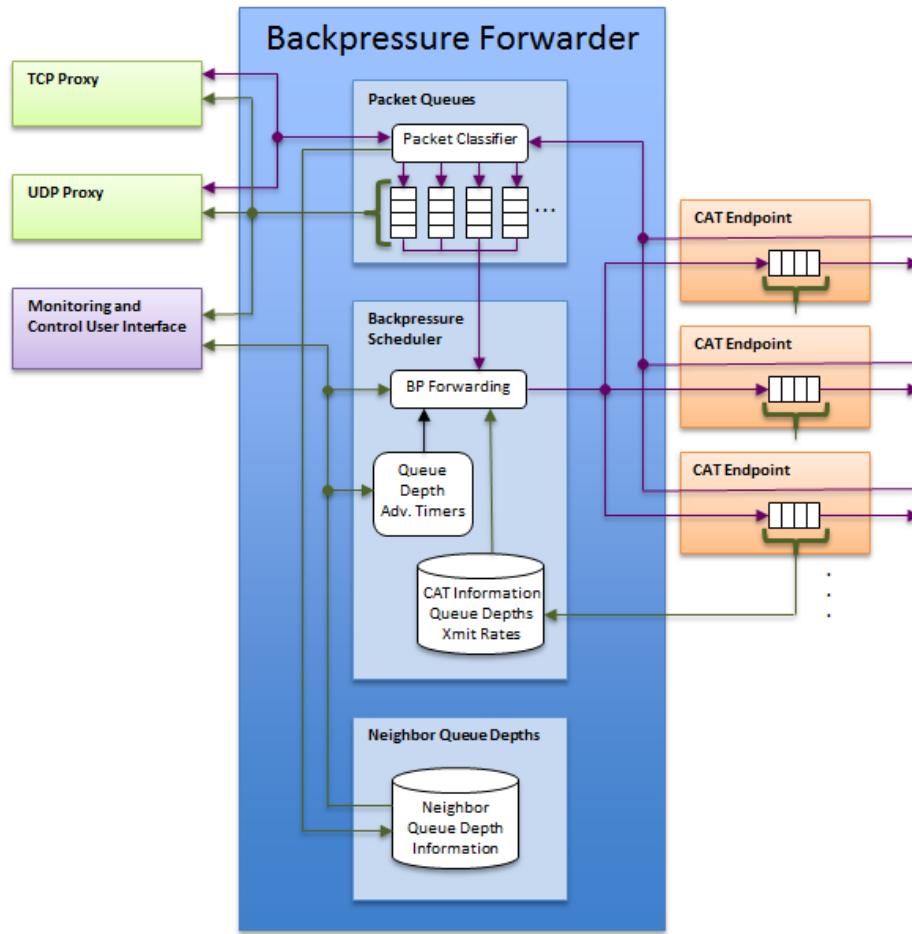


Figure 7. High level design of GNAT's backpressure forwarder showing major components.

Note that each GNAT node has a descriptive name, the GNAT node destination address, a set of subnet addresses for the enclave networks being supported by that GNAT node. Thus, each *unicast* bin identifier refers to a specific, physical GNAT node and a set of subnet addresses. *Multicast* bin identifiers simply reference a set of unicast bin identifiers.

3.2.1 Adjacent Modules and Components

Each Backpressure Forwarder (BPF) sends information to and receives information from peer BPFs via the CATs. Exchanges include both data plane (application packets) and control plane (forwarding state packets) plane information. It also sends application packets to, and receives application packets from local performance enhancing proxies within the enclave, as well as sending forwarding state information to the proxies as needed for their respective admission control processing. Each BPF can also be connected to a local GUI for user control and status monitoring.

3.2.2 Input Sources and Information Needed

3.2.2.1 Control Plane Inputs

Source	Input	Description
CATs	Queue Lengths	This information is used by the backpressure scheduler for controlling the transfer of packets to the CATs. The amount of data kept at a CAT is intentionally kept as small as possible, and BPFs will not hand packets off to CATs for delivery if the transmit queue size exceeds a preconfigured value.
CATs	Tunnel Available Capacity estimates	This information is used by the backpressure scheduler to help it adapt the rate for sending queue length advertisement messages. The system is typically configured so that Queue Length Advertisement Messages (QLAMs) do not exceed a pre-configured percentage of the available capacity.
CATs	Packet Delivery Delay (PDD) statistics	This information describes the one-way forwarding delay for delivering a packet to a neighbor, and is used for making forwarding decisions for latency-sensitive traffic. The statistics includes the mean and variance of the measured delivery delay to reach the neighbor at the other end of the CAT, and includes: packet enqueueing delay in the CATs; one-way propagation and serialization delays over the paths; and the effects of any error control e.g., retransmissions over lossy links) and congestion control operations.
Neighboring BPFs via CATs	Advertised queue values	This information arrives at the CATs in packets ("Queue Length Advertisement Messages" or "QLAM"s), and these packets are passed to the backpressure packet queues. The packet queues component identifies the packets as those containing neighbor queue value information, parses the packets, and passes the queue value information to the neighbor queue value cache in shared memory for processing. These values represent the "backward pressure" in the system, and represent a combination of queue depths (possibly smoothed over time) and queuing delay. Note that queue values are referenced by the BIDs on all GNAT nodes.
All BPFs via CATs	Neighbor PDD and path capacity estimates	This information arrives at the CATs in Link State Advertisement (LSA) packets which are passed to the BPF. LSAs originate at each GNAT node and are flooded across the network to all other GNAT nodes. Information on path delivery delay and path capacity to each neighbor is used for making forwarding decisions for latency-sensitive traffic based on end-to-end delay. The collection of LSAs from all other GNAT nodes implicitly contains network topology information which, in addition to supporting the estimation minimum end-to-end path delays also allows constructing and maintaining "base virtual queue length" information used to improve forwarding over lightly loaded networks.

3.2.2.2 Data Plane Inputs

Source	Input	Description
Local Application Proxies (TCP and UDP)	Proxied Application Packets	These are packets from local application hosts that are destined to application hosts residing in other GNAT enclaves. They are queued for transmission by the backpressure packet queues using the enclave network subnet masks and binning rules.
Neighboring BPFs via CATs	Proxied Application Packets	These are packets forwarded by an adjacent GNAT enclave and received by the CATs. These may be packets destined to the local GNAT enclave or to another GNAT enclave. Packets destined for local application hosts are immediately handed off to the local proxies. Packets destined for other GNAT enclaves and temporarily held in backpressure packet queues.
Neighboring BPFs via CATs	Zombie Packets	These are essentially padding used to mimic application traffic flowing through the network when no valid (non-expired) application packets exist. Zombie packets are sent whenever the backpressure gradient is largest for a bin that has no valid (non-expired) packets in the queue. (In other words, when the gradient, which is normally based on queue depth, has been artificially increased by the GNAT algorithms.) Zombie packets are described in more detail below.

3.2.3 Output Destinations and Information Provided

3.2.3.1 Control Plane Outputs

Destination	Output	Description
Neighboring BPFs via CATs	Queue Lengths	The backpressure packet queues provide the information, and the backpressure scheduler places the information into Queue Length Advertisement Message (QLAM) packets to send to neighboring GNAT enclaves via the CATs. The updates are sent to each adjacent GNAT enclave on its own optimized schedule based on the estimated transmit rate to that enclave.
Local Application Proxies (TCP and UDP)	Queue Lengths	The backpressure packet queue lengths provide the information necessary for making admission control decisions within the proxies, and is provided to the proxies via shared memory.
All BPFs via CATs	Neighbor PDD and path capacity estimates	The path capacity and packet delivery delay characteristics to each neighboring BPF are flooded to all BPFs in the overlay network via Link State Advertisement (LSA). This information is used for making forwarding decisions for latency-sensitive traffic based on end-to-end delay. The collection of LSAs from all GNAT nodes implicitly contains network topology information and supports the estimation minimum end-to-end path delays as well as constructing and maintaining "base virtual queue length" information used to improve forwarding over lightly loaded networks.

3.2.3.2 Data Plane Outputs

Destination	Output	Description
Local Application Proxies (TCP and UDP)	Proxied Application Packets	Packets received by the backpressure packet queues that are destined to the local enclave are passed to the corresponding proxies for delivery to the local application hosts.
Remote backpressure forwarders via CATs	Proxied Application Packets	Packets that are pulled from the backpressure packet queues or are queue value information packets generated by the backpressure scheduler are forwarded to the appropriate CAT based on the backpressure forwarding rules.

Destination	Output	Description
Remote backpressure forwarders via CATs	Zombie Packets	These are essentially padding in the GNAT backpressure forwarder used to mimic application traffic flowing through the network when no valid (non-expired) application packets exist. Zombie packets are sent when, based on the system state, we should have available traffic and must send it in order to maintain the backpressure relationships between enclaves. Zombie packets are described in more detail below.

3.2.4 Events

Backpressure Startup: Upon startup, each BPF retrieves configuration information from a set of local configuration file. This includes the list of Bin IDs for all GNAT enclaves. The backpressure packet queues sets up a packet queue for each Bin ID and configures its packet queuing logic to implement the necessary enclave filtering and binning rules. It creates queue depths objects in shared memory for its own per-bin queue size. The neighbor queue value cache initializes its cache for all of the Bin IDs, and sets up the base virtual queue lengths for all Bin IDs based on the hop count to the destination enclave along with a *hop count multiplier*. The base virtual queue length is the hop count multiplier times the number of hops to the destination enclave (as determined from the given GNAT enclave topology).

CAT Creation/Destruction: Whenever a CAT is setup or torn down, the backpressure scheduler is informed of the change. The backpressure scheduler stores the CAT information, indexed by the CAT's Bin ID, for use in its processing. It sets up a new local queue length advertisement timer for each new CAT, and cancels any existing local queue length advertisement timer for each destroyed CAT.

TCP or UDP Proxy Packet Transmission: The TCP and UDP proxies exchange packets with the BPF using an inter-process signaling FIFO to indicate that packets are available in shared memory. When the local TCP or UDP Proxy passes a packet to the backpressure packet queue for transmission, the packet is placed in the correct backpressure packet queue using the enclave filtering and binning rules. The backpressure scheduler is run to try to forward any packets currently in the packet queues until there are no more transmission opportunities. The backpressure packet queue value state as used by the proxies is updated after at most a given number of packets are sent or after a packet has been received. The queue values are exchanged via shared memory.

CAT Packet Reception: When one of the CATs receives a packet, it is passed from the CAT to the backpressure packet queues. If the packet is destined for the local enclave and it contains neighbor queue value information, then the packet is parsed and the information is passed to the neighbor queue value cache for updating its local state. If the packet is a data packet destined for the local enclave, then the packet is passed to the TCP or UDP proxies depending on the packet's IP protocol field. Otherwise, the packet is queued in the appropriate packet queue and the backpressure packet queue value state as used by proxies is updated. In the first and third cases, the backpressure scheduler is also run to try to forward any packets currently in the packet queues.

CAT Transmit Queue Length Change: When one of the CATs has a change to its local transmit queue depth, the backpressure scheduler is notified. The backpressure scheduler updates its local state and attempts to forward any packets currently in the packet queues.

CAT Network Transmission Capacity Estimate: When one of the CATs has an updated transmit capacity estimate for a path, the backpressure scheduler uses the estimate to compute a new optimal queue length advertisement interval for queue length advertisements for the adjacent GNAT enclave. It then updates the local queue length advertisement timer for the adjacent GNAT enclave.

CAT Network Round-Trip Time Estimate: When one of the CATs has an updated round-trip time estimate to its remote neighbor, the backpressure scheduler records this new value and uses it to make decisions on forwarding Low-Latency traffic.

Backpressure Transmission Scheduler: Numerous other events trigger the backpressure scheduler to see if any packets currently in the packet queues can be sent. From a high level, the backpressure scheduler computes the difference (gradient) between the local queue depth for each destination bin and the queue depth for that same destination bin at each neighbor. Then, considering only those neighbors for which the associated CATs currently have available transmit opportunities, the backpressure scheduler selects and forwards or more packets from the queue for the destination with the largest overall gradient to the neighbor associated with that gradient. Once the packet(s) are dequeued and passed to the selected CAT for transmission, the backpressure packet queue value state is updated.

Note that the backpressure forwarding rules includes a *queue depth hysteresis* value. This is a fixed configuration value that is added to all of the remote queue values before computing the gradient to an adjacent GNAT enclave. This is done to prevent packets from wastefully bouncing between peer enclaves, which we refer to as the "recirculating packet problem."

Local Queue Length Advertisement Timeout: When a timer expires for a local queue length advertisement, it is for a specific Bin ID, which corresponds to an adjacent GNAT enclave. The backpressure forwarder queries the backpressure packet queues for all of the current queue values and creates a packet with the collection of queue value information. It then hashes the packet. If the hash is the same as that for the last local queue length advertisement sent to that adjacent GNAT enclave, and if the time since the last local queue length advertisement sent to that adjacent GNAT enclave does not exceed a maximal time interval, then the packet is not sent. Otherwise, the packet is passed to the applicable CAT for the Bin ID for transmission. In either case, the local queue length advertisement timer for the Bin ID is reset.

3.2.5 Configurable Parameters

- A list of Bin IDs, including the enclave IP subnet information, for each of the GNAT enclaves (not just the adjacent GNAT enclaves). This includes the GNAT enclave topology. This information comes from local configuration files.
- A *hop count multiplier* and a *queue depth hysteresis* value. These are fixed integer values that are used in the backpressure forwarding rules.

- A string selecting whether to use the base BPF algorithm or its Low-Latency forwarding version (details provided below).
- A string indicating which queue depth mechanism to use: the base BPF (actual queue lengths), heavy ball (queue depths smoothed at periodic intervals: details below), or AQUA (queue depths smoothed using an event-driven model: details below)
- A Boolean indicating whether or not to use Zombie Latency Reduction (ZLR), described below, as well as some parameters specific to ZLR.
- A weight indicating how much to value queuing delay vs queue depth in the queue values used for backpressure forwarding and admission control. (See No Packet Left Behind section below.)

3.2.6 Performance Monitoring Statistics

The BPF provides the following statistics for use by the Monitoring and Control User Interface, as well as any other statistics gathering tool. These are also used by the Admission Planner (AMP) to allow broader-level decision making.

Local Backpressure Packet Queue Values: This is the number of bytes and packets in each of the packet queues, and the <bytes/weight + queuing delay> backpressure value along with the binning information. It may be used to monitor the current state of each queue.

Neighbor Backpressure Queue Values: This is the number of bytes (or weight) plus queuing delay value reported by each adjacent GNAT enclave, along with the binning information. It may be used with the local backpressure packet queue values to identify issues with portions of the black network.

CAT Capacity: This is the estimated capacity reported by a CAT to a remote neighbor.

Send Rates: This is the rate of data packets sent or received on each CAT for each destination bin.

3.3 Detailed Design

3.3.1 Bin Identifiers (Bin IDs)

Bin IDs are in essence the addresses of GNAT nodes (or in the case of multicast, groups of GNAT nodes) in GNAT's overlay network. As currently implemented, Bin ID values for individual, physical GNAT nodes are one byte quantities and so are limited to the range 0-255. Bin IDs for multicast groups are four byte quantities and are identically the IPv4 multicast addresses associated with the group.

There are three distinct types of Bin IDs:

- Edge node Bin IDs: i.e., IDs for GNAT nodes located at the edges of an internetwork that act as gateways between applications and GNAT's overlay network. Edge nodes have proxies associated with them which provide the ingress and egress points for application traffic.

- Interior node Bin ID: i.e., IDs for GNAT nodes located within the interior of an internet-work that do *not* provide gateway services for application traffic but merely act as way-points/forwarders. Interior nodes do not have proxies associated with them
- Multicast Bin IDs: each multicast group has a

3.3.2 Bin Maps

Bin Maps (`bin_map.h`) handle the mapping of application addresses (including multicast addresses) to Bin IDs. The following is an example of a bin map configuration file for a simple experiment.

- The first line specifies that there are three backpressure forwarder nodes located at the edges of the network, assigned Bin IDs of 0, 1, and 2.
- The next three lines specify the application address ranges (subnets) that are supported by each individual backpressure edge node. Each line specifies one or more address/prefix length subnet specifications, separated by commas
- The fifth line specifies that there are two backpressure interior nodes assigned Bin IDs of 3 and 4. Since interior nodes do not act as gateways for application traffic, no subnets are assigned or used. Note that there need not be any interior nodes and hence this line is optional
- The sixth line specifies the number of preconfigured multicast groups. Pre-configuration of multicast group addresses and associated membership is sometimes convenient but is not required and hence this and the following set of lines are optional. Multicast group management can be entirely automatic using the Multicast Group Management Sniffer (MGMS) described later in this document
- The seventh line specifies the multicast group address for the one preconfigured group
- The eighth line specifies the Bin IDs of edge nodes that are part of the multicast group.

```
BinMap.BinIds 0,1,2
BinMap.BinId.0.HostMasks 172.16.1.0/24
BinMap.BinId.1.HostMasks 10.1.1.0/24,172.16.2.0/24
BinMap.BinId.2.HostMasks 10.1.3.0/24,172.16.3.0/24
BinMap.IntBinIds      3,4
BinMap.NumMcastGroups 1
BinMap.McastGroup.0.Addr 227.9.18.31
BinMap.McastGroup.0.Members 0,1,2
```

BinMap instances are initialized by parsing the information in the configuration file `bin_map.cfg`.

3.3.3 Bin Indexes

Whereas Bin IDs are external representations of both GNAT node addresses and multicast addresses that are consistent across GNAT's overlay network, internal referencing within the backpressure forwarder is instead done using Bin Indexes. Bin Indexes are explicitly designed to provide for compact storage as well as enable rapid lookups and retrievals via array-based access within the backpressure forwarder. This is especially useful when the configured set of Bin IDs are sparse and non-contiguous.

Unlike Bin IDs, Bin Indexes are not guaranteed to be the same between any pair of backpressure forwarders. A key reason for this is the ability to handle dynamic multicast groups without requiring a complex consensus protocol. Consider the case of when two different multicast receivers each join a different multicast group through their local GNAT nodes, and neither multicast group has previously been seen by either GNAT node. In this case each GNAT node will independently assign the first available Bin Index to their respective groups, without needing to coordinate or negotiate with any other GNAT nodes. When these two GNAT nodes subsequently exchange multicast group membership information, each assigns the "new" multicast group to the next available Bin Index. Again, the multicast group to bin index mapping is different for each GNAT node.

GetDstBinIndexFromAddress(ip_addr) is the primary method in the Bin Map class for determining the association between a packet's destination address and a Bin Index. The Bin Index can be converted to a Bin ID using the Bin Map methods *GetPhyBinId(bin_index)* for unicast Bin Indexes and *GetMcastId(bin_index)* for multicast Bin Indexes respectively. The reverse mapping from Bin IDs to Bin Indexes is done using the Bin Map methods *GetPhyBinIndex(bin_id)* for unicast Bin IDs, and *GetMcastBinIndex(bin_id)* for multicast Bin IDs, respectively. The Bin Map class also provides the method *IsMcastBinIndex(bin_index)* for determining whether a Bin Index is for a multicast group.

GetMcastIdFromAddress(ip_mcast_addr) is a lighter weight version of *GetDstBinIndexFromAddress(ip_addr)* that is used when it is known that the packet's IPv4 address is in fact an IP multicast address. This can be determined using the *IsMulticast()* method from the *IPv4Address* class.

3.3.4 Destination Vectors

Destination vectors (type DstVec in iron_types.h) are used to specify a set of unicast destinations (i.e., GNAT edge nodes) for multicast packets. Since destination vectors are carried in packet headers, they must (a) be interpreted the same at all nodes (i.e., it cannot carry Bin Indexes), and (b) be a fixed size.

A DstVec has the following properties:

- Currently a DstVec is a uint32 (defined in iron_types.h) with the first 8 reserved for the packet header type byte. Hence only 24 bits of the DstVec are currently usable, limiting the number of destination/edge nodes to 24.
- Each bit in the DestList is set to 1 if and only if the destination whose Bin ID matches the index of the bit is a destination of the packet.

NOTE: the 24 destination limit is a current implementation for simplicity. However, all DstVec operations within the GNAT codebase leverage abstraction methods to avoid hard-coding the DstVec format throughout the code. This allows the DstVec definition to be easily modified, with changes to the supporting code limited to this set of abstraction methods. The currently-defined DstVec methods include:

- *IsBinInDstVec* (returns true if a given Bin Index matches a destination included in the given DstVec)
- *IsOnlyBinInDstVec* (returns true if a given Bin Index is the ONLY bin included in a given DstVec)
- *GetNumBinsInDstVec* (returns the number of bins specified in a given DstVec)
- *AddBinToDstVec* (returns a new DstVec with the provided Bin Index added to an existing DstVec. Does not alter the passed in DstVec)
- *RemoveBinFromDstVec* (returns a new DstVec that is equal to the provided DstVec with a provided Bin Index removed.)
- *DstVecSubtract* (returns a new DstVec that is equal to the provided DstVec with all Bin IDs in a second given DstVec removed from it)

Note that although the DstVec is defined in terms of Bin IDs, most of these operations use Bin Indexes. Translation from Bin Index values to Bin ID values is done within the methods.

3.3.5 Multicasting within GNAT

Traditional IP multicast is receiver-oriented, whereby each receiver decides if and when to join or leave a multicast group, and senders do not know or care about who or where the receivers are. Routing and forwarding is handled entirely within the network, and is performed by constructing, maintaining, and leveraging a multicast routing tree rooted at each sender. In contrast, GNAT is sender-oriented in that the set of receivers is known a priori by the sender, and the sender provides an explicit list of receivers in each packet at the time of transmission.

There are shortfalls and advantages of either approach. The traditional IP multicast approach is intrinsically more scalable, particularly when the set of possible receivers (potentially the entire set of physical GNAT nodes) is large. Alternatively, GNAT's approach can make better use of the available network resources by finding and exploiting *multiple* trees from each sender concurrently, and hence is more consistent and compatible with backpressure's use of multiple paths for load balancing. GNAT's sender-oriented approach also facilitates advanced functionality; for example a sender can choose at the time of transmission which receivers in the group should receive any given packet to provide a type of multi-resolution delivery service. This latter feature is particularly attractive for applications such as distributed simulation where not all simulated entities require the same state update rates, allowing for more efficient use of the network.

It is important to remember that GNAT operates as an *overlay* network, where each GNAT node acts as a transparent gateway for a potentially large number of application hosts. Since the list of "receivers" carried in each packet specifies edge GNAT nodes rather than individual application hosts, GNAT's scaling limitations do not in fact restrict the number of application hosts, but rather simply the number of edge gateways in the overlay network. Nonetheless, the overhead of identifying GNAT receiver nodes in each packet can still represent considerable overhead, so to make this as lightweight and efficient as possible, receivers are encoded with the multicast packets using DstVecs with one bit position for each GNAT node. As described above, this bit vector occupies 3 bytes in the multicast metadata header, hence the maximum number of multicast receivers is currently limited to 24. Additionally, these GNAT receivers must be assigned Bin IDs between 0 and 23 inclusively.

3.3.6 Queues and Queue Lengths

The queue lengths used for making backpressure forwarding and admission control decisions are a sum of several terms, including:

- The physical queue lengths (or weights),
- Packet-less zombie queue lengths (or weights)
- Two types of virtual queue lengths

These are summarized in this section.

3.3.6.1 Physical Queues

Each backpressure forwarder maintains multiple sets of queues containing physical packets waiting to be sent, where each queue set corresponds to a different destination bin (either a unicast bin or a multicast bin). Each queue set consists of multiple distinct queues, one for each traffic type, as needed to support service differentiation (e.g., for handling latency-sensitive traffic). There are currently three traffic types associate with handling physical packets, including: latency-insensitive (for normal traffic); latency-sensitive (for traffic marked for Expedited Forwarding – EF); and critical latency sensitive (used to handle special urgent conditions in forwarding EF traffic). Critical latency sensitive packets are discussed later in this document.

3.3.6.2 Packet-less Zombie Queues

In addition, there is a fourth traffic type which we refer to as Zombie packets. Zombie packets contain no usable payload and may be created, for example, as a result of dropping expired EF packets in order to avoid disrupting proper backpressure behavior. Since their payload is not/no longer usable, Zombie packets need not be stored in a queue but instead are more efficiently maintained as a total byte count associated with a given bin and traffic type. When Zombie packets are forwarded to a given neighbor, a physical packet is created having a length which is the smaller of the Zombie byte count for the given bin or a maximum packet size (e.g., 1250 bytes).

3.3.6.3 Virtual Queue Lengths

There are also additive bias terms described as "virtual queue lengths", which can be divided into "base virtual queue lengths" and "floating virtual queue lengths." Base virtual queue lengths are computed from topology information, and are primarily used to ensure there is a clear "downhill direction" for lightly loaded networks – i.e., when there is otherwise insufficient traffic to build up a usable backpressure gradient. The base virtual queue length for a given destination is computed as the minimum number of hops from the GNAT node to reach a given destination GNAT node multiplied by a "hop count multiplier" (currently 1100 bytes, but configurable). These quantities are relatively static, changing only when the network topology changes.

For forwarding of unicast traffic, the base virtual queue length to reach a given destination is simply added to the physical queue length for that destination bin. For forwarding of multicast

traffic, the base virtual queue length to reach each destination in the group is added to the corresponding queue length for that destination in the multicast group, and are maintained and used as vectors of queue length information.

Floating virtual queues are dynamically adjusted values used to reduce the number of physical packets needed to reach and maintain forwarding equilibrium. Specifically, for any forwarding situation there is some set of queue lengths for which backpressure achieves its optimal throughput characteristics. Depending on the network topology and the specific forwarding configuration, these queue lengths can be quite large. This in turn can result in long end-to-end packet forwarding delays if most or all of the queue lengths are due to physical packets. Use of floating virtual queues allows achieving the same effective queue lengths by substituting virtual packets for physical packets, so that the sum of the is the same, yet only a small number of physical packets need be enqueued at any given time. Hence backpressure forwarding retains its optimal throughput characteristics, yet the queuing delay is reduced.

As floating virtual queues are managed as byte counts in the same way that packet-less zombie queues are managed, we also refer to virtual queue as zombies. Floating virtual queue lengths are dynamically adjusted in order to maintain a gradient to the destination when the load or networking conditions change. This process is described in greater detail in the section on Zombie Latency Reduction (ZLR).

3.3.6.4 Queue Scaling and Smoothing

GNAT also supports several exploratory mechanisms for scaling and smoothing physical queue lengths. Scaling provides an alternate means for reducing queuing delays when otherwise large queues are needed for backpressure to achieve optimal throughput characteristics. Smoothing dampens the oscillation from the system that naturally arises from using delayed feedback of queue length information from neighboring nodes. Dampening oscillations further acts to reduce queuing delays, and provides a type of momentum towards an equilibrium and stability once equilibrium is reached.

GNAT supports two different algorithms, referred to as Heavyball and AQUA, for scaling and smoothing operations. For both algorithms, backpressure forwarders report and use scaled values instead of actual queue depths; we refer to such scaled values as weights.

In general, we have found that the multiplicative mechanism intrinsic in scaling scaling does not work as well as the additive mechanism embodied in ZLR, as ZLR allows finer grained control over the number of actual physical packets enqueued, especially when the queue lengths needed to achieve optimal throughput are large. Nonetheless, these algorithms are supported as queue scaling has been a popular delay reduction mechanism in the community and hence provide a basis for comparing with our work.

Queue smoothing, however, does help reduce queueing delays.

3.3.6.5 Organization of Queues within the BPF Implementation

Queues within the BPF are organized as follows. The relationship between the various components is shown in Figure 8 below.

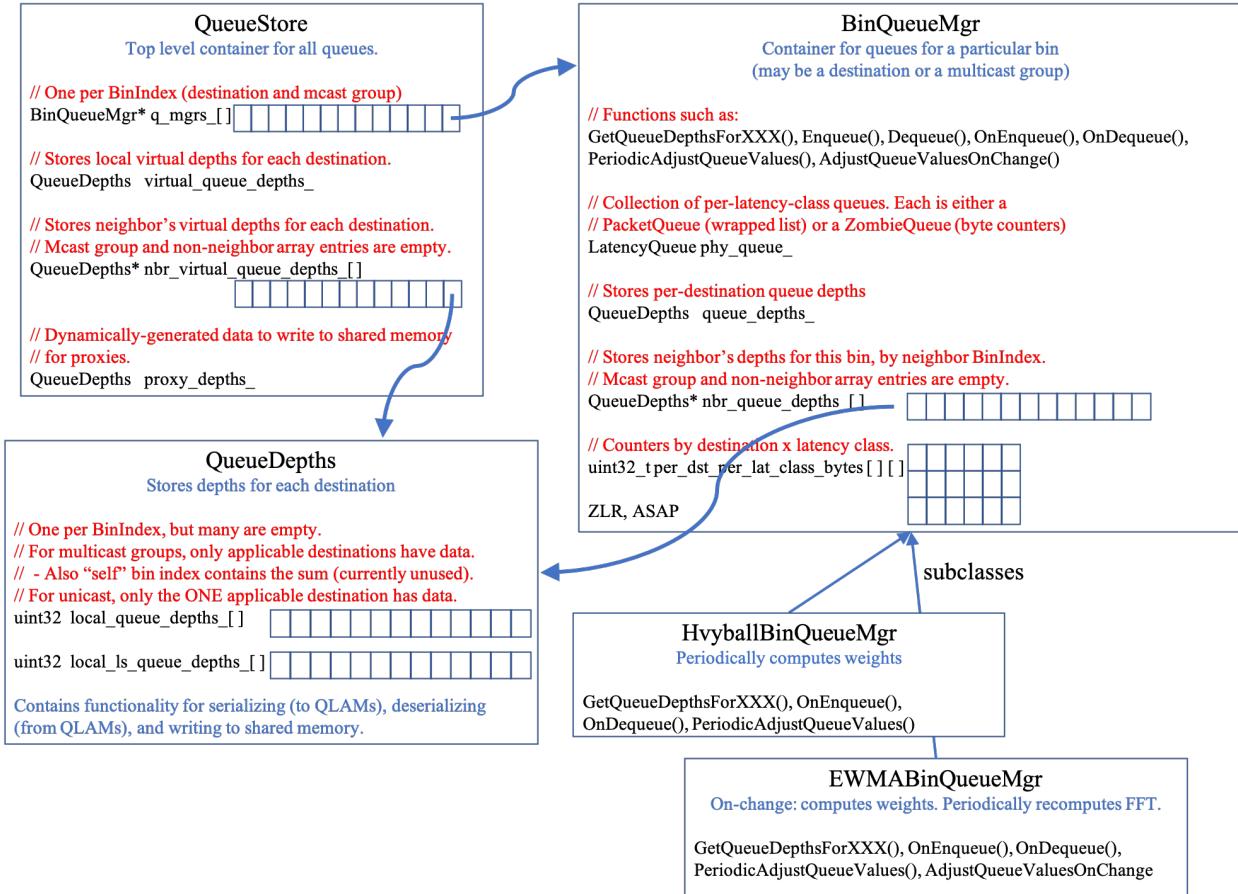


Figure 8. Organization of queues and queue management structure within the BPF.

3.3.6.5.1 QueueStore

Purpose: The queue store is the basic container for all queues used by backpressure forwarding.

Key fields: The queue store holds the per-bin `BinQueueMgr`s as well as data that is shared across all bins (namely, virtual queues, which are defined by destination and shared across all bins). The queue store is also responsible for collecting a single "depth" value for each bin to provide to the proxies for admission control.

Key functions: Most of the functions in the `QueueStore` are just pass-through functions into the per-bin `BinQueueMgr`s.

3.3.6.5.2 BinQueueMgr

Purpose: The bin queue manager manages the queues for a single bin index, which may be a destination bin index or a multicast group bin index. This includes managing the physical queues of packets as well as all weights, depths, etc.

Key fields: The bin queue manager contains a LatencyQueue (`phy_queue_`), which wraps all physical queues of packets for this bin index, including one queue for each latency class. The bin queue manager also tracks the queue depths in two ways: the total depths (and total latency sensitive depths) for each destination bin index are stored in a queue depths object. Also, the individual latency class depths are stored in a 2D array. Most logic is performed using the total per-destination depths, and these depths are shared with other entities (other BPFs via QLAMs and proxies via shared memory), but queue depth graphs and ZLR both use per-latency-class depths. The bin queue manager also stores the queue depths for this bin index from each neighbor. Lastly, the bin queue manager includes references to the classes performing ZLR and ASAP if those features are enabled for this bin index.

Key functions: The bin queue manager includes several types of functions.

- Pass-through functions for viewing and manipulating queues. (e.g., `Peek`, `Dequeue`, `Enqueue`)
- Functions for moving packets between queues (e.g. `ZombifyPacket`, `CriticalizePacket`)
- Accounting functions to update queue depths and trigger appropriate algorithms after packets are enqueued, dequeued, or moved (e.g. `OnDequeue`, `OnEnqueue`)
- Functions for accessing the correct queue depths for a specific purpose (e.g. `GetQueueDepthsForBpfQlam`, `GetQueueDepthForProxies`). Note that "GetQueueDepthForProxies" returns a single depth value that will be collected across all bin indexes in the Queue Store. These accessor functions are useful for subclasses that do not use straight depth counts for all purposes.
- Functions to trigger algorithms, both periodically and event-driven (e.g. `AdjustQueueValuesOnChange()`, `PeriodicAdjustQueueValues()`). These are hooks for subclasses.

Subclasses: There are three subclasses of `BinQueueMgr` that may be used instead of the base `BinQueueMgr` implementation.

- `HvyballBinQueueMgr` maintains the heavyball queue weights
- `EWMABinQueueMgr` does oscillation reduction and updates smoothed queue weights
- `NPLBBinQueueMgr` computes queue values that are a combination of actual queue depths and a delay term. This is the No Packet Left Behind algorithm, which cannot be used with ASAP.

3.3.6.5.3 QueueDepths

Purpose: The queue depths object is a simple wrapper for two sets of depths: one for the per-destination totals and the other for per-destination latency-sensitive totals. This wrapper is helpful for sharing queue depths with external entities, such as neighbors (via QLAMs) and proxies (via shared memory).

Key fields: The queue depths object primarily just includes two arrays: one for totals, one for Latency Sensitive (LS) totals. In addition, there are some fields used for iterating through the bins and some fields related to storing the depths in shared memory.

Key functions: The main functions in QueueDepths are the accessor functions: GetBinDepthsByIdx and SetBinDepthByIdx. Note that functions still exist for getting and setting by BinId; these should almost never be used, as Bin IDs are only to be used on-the-wire and for logging. QueueDepths also includes functions for Serializing and Deserializing for writing and reading QLAMs, and accounting functions for updating depths. **Note that the accounting functions both in QueueDepths and in BinQueueMgr.** (**Note: queue depths must never be recomputed from scratch.** This is different from the pre-multicast ION codebase, where depths were recomputed on every change. Because of this, every time a packet is added, removed, changed in size, or moved from one queue to another, accounting functions absolutely **MUST** be called.

Comments: Unlike the ION pre-multicast code, the queue depth arrays with QueueDepths are mostly 0s. For a multicast group bin, there are non-zero depths only for the destinations in the group. For a destination/unicast bin, the ONLY non-zero queue depth is for the bin index matching the destination's bin index.

3.3.6.5.4 LatencyQueue

Purpose: LatencyQueue is a struct define within BinQueueMgr.h that just contains a single array, indexes by LatencyClass, of pointers to Queue objects. Each Queue it points to will either be a PacketQueue or a ZombieQueue.

3.3.6.5.5 PacketQueue

Purpose: Wraps a linked list (which may be ordered by TTG or not ordered) containing pointers to the physical packets enqueued for a specific latency class and bin index.

Key fields: A list of packet pointers.

Key functions: The functions in a PacketQueue are variants of Enqueue, Dequeue, Peek, and Iterate.

3.3.6.5.6 ZombieQueue

Purpose: Implements the same Queue interface as PacketQueue, but does not contain real packets. This is used for zombies, where we don't need any actual packet information; rather, only a count of bytes is needed. Since zombie queues are stored with the BinQueueMgr, they are **NOT** per-destination for multicast, but are shared across all destinations.

Key fields: A ZombieQueue is primarily a wrapper for a QueueDepths object that stores the per-destination byte counts. For a destination/unicast bin, only one destination will have any values. For a multicast bin, there may be non-zero depths for multiple destinations. If a zombie is enqueued with a non-singular dest list, those bytes will be included separately for each destination.

Key functions: ZombieQueue implements the same functions as PacketQueue. However, instead of just Enqueue and Dequeue (which take/return packets), a ZombieQueue also includes functions AddZombieBytes and DropPacket (which takes a byte count). These functions avoid the need to ever create a physical zombie packet.

3.3.7 Queue Length Advertisement Message (QLAM) Packet Format

QLAM packets are UDP/IP packets containing a header and a variable length payload of one or more (multicast group ID, bin queue depth vector) pairs. Each bin queue depth vector is itself variable length, and consists of one or more (bin ID, bin queue depth pair) pairs.

About bins: forwarding within the GNAT network is entirely based on GNAT node IDs, with a single GNAT edge node possibly supporting multiple application hosts. Hence from a forwarding perspective we "bin" together the set of application host addresses supported by a given GNAT edge node, and use that bin ID for making forwarding decisions.

The QLAM header consists of:

- type - 1 byte; a value of 1 is used to indicate it is a QLAM
- source Bin ID - 1 byte. Identifies the node ID of the BPF node sending the QLAM
- sequence number - 2 bytes: ensures the most recent information is used when packets arrive out-of-order
- number of (multicast group id, receiver queue depth vector) pairs - 2 bytes

Each (multicast group id, bin queue depth vector) block consists of:

- multicast group ID - 4 bytes; this is identically the IPv4 multicast address (or 0 for the special case for handling unicast queue depths; see below)
- number of (receiver Bin ID, bin queue value pair) pairs - 1 byte

Each entry in the receiver queue depth vector consists of:

- receiver bin ID - 1 byte (this is identically the index of the GNAT node supporting an enclave at the network edge)
- NL queue depth - 4 bytes: queue depth for this bin for this multicast group for Non-Latency-sensitive traffic
- LS queue depth - 4 bytes: queue depth for this bin for this multicast group for Latency-Sensitive traffic

Note that within this vector-oriented representation, we handle unicast by grouping each of the individual unicast queues into a single "multicast" group assigned a special group ID value of 0. Hence when the multicast group ID is set to 0, each receiver bin-queue value pair specifies a separate and distinct unicast receiver node and associated queue depth.

This information is serialized into a byte array as shown below.

0	1	2	3
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Type Src Bin Id Sequence Number			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Sequence Number Num Groups			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Group Id 0 (all ucast)			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Num Pairs Dst Bin Id 0 Queue Depth for Bin Id 0			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Queue Depth for Bin Id 0 LS Queue Depth for Bin Id 0			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
LS Queue Depth for Bin Id 0 Dst Bin Id 1 QD for Bin Id 1			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Queue Depth for Bin Id 1 LS Queue Depth			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
LS Queue Depth for Bin Id 1 ...			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
~			~
~			~
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Group Id 1 (mcast)			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Num Pairs Dst Bin Id 0 Queue Depth for Bin Id 0			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Queue Depth for Bin Id 0 LS Queue Depth for Bin Id 0			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
LS Queue Depth for Bin Id 0 Dst Bin Id 1 QD for Bin Id 1			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Queue Depth for Bin Id 1 LS Queue Depth			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
LS Queue Depth for Bin Id 1 ...			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
~			~
~			~
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Group Id i (mcast)			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Num Pairs Dst Bin Id 0 Queue Depth for Bin Id 0			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Queue Depth for Bin Id 0 LS Queue Depth for Bin Id 0			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
LS Queue Depth for Bin Id 0 Dst Bin Id 1 QD for Bin Id 1			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Queue Depth for Bin Id 1 LS Queue Depth			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
LS Queue Depth for Bin Id 1 ...			
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			

All entries are in network byte order.

Note that each QLAM packet is hashed before being sent. If the hash is equal to that of the previous packet sent, then the packet is only sent if the time elapsed since the last packet sent is greater than the QLAM advertisement interval. This prevents redundant QLAMs from using network capacity.

3.3.8 End-to-end Latency Estimation, Time-to-Go Tracking, and Impeding Packet Circulation

The BPF estimates the end-to-end latency from each GNAT node to all other GNAT nodes in support of latency aware forwarding. To accomplish this, each GNAT node provides local information to every other GNAT node using a link-state mechanism, with the goal of collecting information from across the topology as needed to estimate end-to-end delivery delays. The local information includes a list of neighbor GNAT nodes (GNAT nodes that are directly connected to this GNAT node by CATs) and statistics on the packet delivery delay (*pdd*) to reach each neighbor GNAT node via the associated CAT.

3.3.8.1 Estimating the Packet Delivery Delay for a Given CAT

The *pdd* statistics constructed by each CAT include the mean and variance of sender-to-receiver delivery delays across the path segment managed by the CAT. Statistics are created from sample sender-to-receiver delivery delay measurements for individual packets, which include 1) queueing delays at the sender before a packet is first transmitted; 2) transmission/serialization and propagation delays; and 3) contributions from error control actions such as retransmissions.

A CAT computes the *pdd* for a packet as follows. When the packet is first enqueued in the CAT, the enqueueing time is recorded by the CAT's packet queue. When it is dequeued for transmission, the CAT computes a *time-in-queue* by subtracting the enqueueing time from the dequeuing time, then records the *time-in-queue* value for the packet. The time that the packet is first transmitted is also recorded.

When a packet is successfully received (or equivalently, decoded if FEC is being used) at the CAT receiver, the receiving CAT sends an ACKnowledgement back to the sender. Upon receiving the ACK, the sender takes the time that the ACK was received and subtracts the time of the first transmission to form a base delivery-and-acknowledgement delay for the packet. From this quantity it then subtracts half of the running estimate for the round trip time (*rtt*) to account for the delay associated with sending and receiving the ACK packet. Finally it adds the *time-in-queue* value to arrive at a measure of the *pdd* for this packet.

Each such measurement is passed to a *pdd* estimator within the CAT which updates the running estimate of the mean and variance of the *pdd* using exponentially weighted averaging. The mean and variance values are periodically passed to the BPF, which periodically broadcasts the mean and variance (actually the mean and standard deviation) of each of its CATs to all other BPFs using Link State Advertisement (LSA) messages, as described below.

3.3.8.2 Using Path Latency Information for Latency-aware Forwarding

Each latency-sensitive data packet carries with it a *Time-To-Go* (TTG) value that indicates the amount of time remaining before it is considered "expired" -- i.e., when enough time has been spent in transit that delivering it to the receiving application will yield no utility. Upon admission into the network, packets are assigned a configurable TTG that is subsequently decremented at each intermediate GNAT node by 1) the amount of time spent enqueued at each node along the

forwarding path and 2) by the packet delivery delay (pdd) estimate over each CAT. A negative time-to-go indicates that the packet has expired.

TTGs are used in conjunction with *Time-To-Reach* (TTRs) estimates for latency-aware forwarding, basically to avoid sending packets along paths with expected delays that are likely to result in the packet arriving late. TTRs are constructed using the topology and packet delivery delay estimates provided via the Link State Advertisements (LSAs). Each GNAT node uses the collection of topology and latency information obtained from LSAs from each GNAT node to compute a minimum time to reach a given GNAT edge/destination node through each of its interfaces.

The minimum time-to-reach a given GNAT destination node via a given interface is computed using Dijkstra's algorithm. Starting with the destination node and with no other nodes selected, the Dijkstra process incrementally selects the node from the pool of as-yet unselected nodes which has the smallest incremental time-to-reach TTR_{incr} . The incremental time to reach TTR_{incr} the destination over a sequence of path segments associated with previously selected nodes is computed as follows:

$$TTR_{incr} = \sum_{i \in \text{paths used}} mean_i + 2.2 \sqrt{\sum_{i \in \text{paths used}} variance_i}$$

This formula assumes that the *pdd* measurements over each path segment are independent, such that the mean and variance of the delay along a path are the sum of the means and variances respectively of the individual path segments. Then, assuming a normal distribution, the multiplicative factor of 2.2 specifies the point where ~98.5% of the packets will arrive by the given TTR.

We note here that this calculation does **not** include queuing delays at individual hops. While it is fairly straightforward to add the means and variances of delays for those packets that use a given interface (and in fact there is some support for this in the current code base), we elected not to do so. Our premise is that, consistent with current provisioning practices for low-latency services, latency-sensitive packets should be serviced first if at all possible, and queuing delays should be kept close to zero.

TTRs are computed for each CAT interface separately to determine which neighbors can be used to reach a given destination GNAT node in time, and which cannot. Only those CAT interfaces for which a packet can be delivered on-time will be considered by the forwarding algorithm.

To make the latency aware forwarding process as efficient as possible, TTR values are cached, indexed by destination and interface. We note here that GNAT nodes keep only the most recent topology and associated latency estimates from each of the other GNAT nodes.

Rather than being dropped, packets that have expired are converted to "Zombie" packets and are assigned the lowest forwarding priority. This is necessary to preserve the queuing relationships within the network necessary for backpressure to work properly; dropping packets can otherwise make a "dead end" node appear as if it is the destination and is consuming packets – causing more packets to be forwarded and subsequently dropped.

3.3.8.3 Impeding Packet Circulation

With backpressure, the ebb and flow of backpressure queues can sometimes induce packet circulation, whereby a packet is forwarded to a given GNAT node multiple times; packet circulation typically occurs with under loaded networks.

The topology information provided by the above LSA mechanism also provides a means for reducing or eliminating packet circulation. Here, each packet carries with it a *history vector*, which is a bit map that records which GNAT nodes that it has already visited and is updated whenever the packet arrives at an GNAT node. Packet forwarding (described below) uses the topology and the history vector to restrict which nodes are considered as next hops.

3.3.8.4 Link State Advertisement (LSA) Messages

As described above, LSAs provide link latency (and optionally capacity) information from a node to each of its one-hop neighbors. LSAs are flooded across the GNAT network, such that each GNAT nodes receive LSAs from all other GNAT nodes.

The format of LSA messages is provided below. The first block shows the structure of LSA messages when capacity information is not included; the second block shows the structure of LSA message when capacity information is included. Note that to keep LSAs small, capacity information is encoded in a 16-bit floating point representation, as shown below the second block.

C: Flag indicating whether link capacity estimates are included.
C = 0, estimated capacity not included.

0	.	1	.	2	.	3																	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1		
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	
Type (LSA)	Src Bin Id	Sequence Number																					
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		
Num Nbrs	Num bins	C Padding																					
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		
BinId	Latency Mean (in 100us)																		Latency Std				
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		
Dev (in 100us)	BinId	Latency Mean																					
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		
Latency Std Dev	BinId	Latency Mean																					
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		
Num Bins: If 0, queue delays are not included.																							

$C = 1$, estimated capacity included.

0	.	1	.	2	.	3																
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	Type (LSA)		Src Bin Id		Sequence Number																	
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	
	Num Nbrs		Num bins		C				Padding													
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	
	BinId		Latency Mean (in 100us)													Latency Std						
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	
	Dev (in 100us)		Estimated Capacity (bps)													BinId						
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	
	Latency Mean													Latency Std Dev								

```
+++++-----+-----+-----+-----+-----+-----+  
| Estimated Capacity (bps) | BinId | Latency Mean  
+++++-----+-----+-----+-----+-----+-----+
```

Each estimated capacity is encoded as: $C = (i.d) \times 10^e$.
i (4bits): Integer part (from 1 to 9).
d (8bits): Decimal part (from 0 to 0.996)
e (4bits): Exponential part (from 1 to 16)

Type: The 1-byte packet type (must be LSA).

Src Bin Id: The 8-byte source bin id identifying the LSA source.

Seq. Num: The 2-byte sequence number to determine information freshness.

Num.Nbrs: The 1-byte number of entries in the LSA (refers to neighbors).

Padding: 3-byte padding.

BinId: A 1-byte destination bin id of the source.

Padding: 1-byte padding.

Latency: 2-byte latency value to neighbor in units of 100 microseconds.

Whenever an LSA is received by the BPF, the BPF checks for data freshness, and if more recent than the last record, enters it in a large table. At this time, the cache storing latency values to all destinations is reset since the new information invalidated it. New latency values will be recomputed when needed.

3.3.9 Backpressure Forwarding Methods

3.3.9.1 System Implementation Overview

The core of the Backpressure Forwarder (BPF, backpressure_fwder.h) is a select-loop intended to catch arriving packets from the Proxies, arriving packets from the BPF modules of other GNAT nodes, timers indicating it is time to send the next packet, and control messages. Arrival events (packets from proxies, packets from remote GNAT nodes, and control messages) are triggered using a select call on a set of file descriptors. Send events are triggered using a timeout on the select call, set up for the next time any CAT will be available to send a packet. When the select call returns or times out, the BPF processes all events. Then the file descriptors and time-out value are reset, and the BPF loops back to wait for the next trigger.

3.3.9.2 Receive Packet

Upon receiving a packet, the BPF invokes ProcessRcvdPacket, which first checks for a proper version of the packet and then determines packet disposition based on packet type (UDP, TCP, or GNAT control packet). Control packets (such as QLAM packets) are processed locally. Data packets intended for a local application are directed to the local proxy; otherwise they are placed in the BPF queue with the corresponding bin id for subsequent forwarding.

3.3.9.3 Timer Expiration

QLAM timer expiration invokes a callback to a method that sends a QLAM. In the objective system design, QLAMs are sent to neighbor backpressure forwarders over CATs while local proxies read queue values over shared memory.

With every select expiration, the BPF forwarding algorithm (`backpressure_dequeue_alg::FindNextTransmission`) is also executed, since queue value changes may alter the backpressure decisions about what, where and when to send.

3.3.9.4 Receive Control Message

A control message may be received for the Path Controllers or BPF to set a config info value.

3.3.9.5 FindNextTransmission Scheduler

FindNextTransmission is the main method computing the forwarding solution and is part of the select loop triggered by packet arrivals and transmissions. For every packet, FindNextTransmission computes a set of transmit solutions that comprise a dequeued packet, its destination bin b and the interface i on which it must be sent.

The method implements two variants and a number of techniques intended to handle various traffic conditions:

- Base, the forwarding method in which latency is ignored and only largest-gradient forwarding is considered,
- LatencyAware, the forwarding method that favors low-latency packet transmissions and ensures that packets deadlines will be met,
- ConditionalDAG and HeuristicDAG anti-circulation, the methods used to avoid capacity-consuming loops,
- Non-blocking queue-depth search, the technique that ensures that if a packets at the head of the queue is not a suitable transmit candidate, it will not block the following few packets from consideration for transmission.

3.3.9.5.1 Packet Queues and Traffic Types

GNAT contains four queue types based on latency requirements: Z (zombie), S (standard), L (low-latency), and C (critical).

The Z queue is a special queue that holds low-latency packets that have “expired”—i.e. they cannot be delivered in time on any interface—as well as other dummy packets used to shape backpressure computations. This queue contains packets that, if delivered, would provide no value to the end applications, and thus may be configured as a packet-less queue that discards the physical packets and stores only a byte count. The Z queue is necessary to inflate the queue depths or weights used for backpressure and admission control decisions. Packets may be added to the Z queue for three reasons: 1. Expired latency-sensitive traffic (see [Cleanup Expired Packets](#)), 2. To

add artificial queue value when a queue experiences excessive delay (see [No Packet Left Behind](#)), and 3. To build up a queue floor intended to reduce latency from queuing delay (see [Zombie Latency Reduction](#)).

The S queue holds packets not marked for expedited forwarding, and holds the bulk of the application traffic (we assume here that EF traffic is a small fraction of the total offered load).

The L queue holds packets marked for expedited forwarding (EF), with each packet in the L queue having a remaining time-to-go (TTG) that is computed by subtracting the time spent in transit from the original packet delivery deadline. Time spent in transit includes the hold time spent enqueued at the current BPF node, and so the TTG must be updated each time the forwarding algorithm is called.

The C queue is also a special queue, and holds EF packets that have not yet expired, but delivering these packets either on time or within other forwarding constraints requires that the packets be sent only to the next hop with the lowest delay to the destination. The primary reason for the C queue is supporting the use of *history vectors* as a means of reducing packet circulation. Packet circulation is an artifact of backpressure forwarding when the dynamic gradients allow a portion of the traffic to be routed through the same set of nodes multiple times. Circulation, while allowed by backpressure forwarding rules, consumes bandwidth capacity and increases latency.

A history vector is a bit map maintained in the metadata of each packet that records each GNAT node that has forwarded the packet. History vectors are used to prevent packet circulation by avoiding sending a packet to a node that has already forwarded it.

For simplicity, let C(b), L(b), S(b), and Z(b) refer to the critical, low-latency, standard, and zombie queues for destination bin b.

3.3.9.5.2 Basic Steps of the Forwarding Algorithm

The process for selecting the next packet to transmit works as described in the following sections. (Note: These sections are called out explicitly in comments within the `backpressure_dequeue_alg.cc`.)

3.3.9.5.2.1 Cleanup Expired Packets - S1

Zombification Clean-up

Each latency-sensitive packet has an assigned time-to-go (TTG) indicating how much time remains before it will be considered ‘late’ and unusable by the destination. As a first step, we move any low-latency packet that have a TTG value less than the latency of the lowest latency path to the destination from queues L(b) and C(b) into queue Z(b) for each bin b. (This is affectionately known as “zombification”, a process by which expired packets are still allowed to move in the network, but are ultimately dropped somewhere in the network). Note that this particular step can be done outside of the packet selection process, but we include it here for clarity.

Time-to-Reach and Definition of “Expired” for ConditionalDAG and HeuristicDAG Anti-Circulation

We retrieve or recompute the time-to-reach (TTR) each destination via each path controller and retain the minimum TTR and corresponding path controller i . TTR values are computed using a Dijkstra algorithm over path latency updates that are periodically flooded to the network using LSAs.

A packet to bin b is considered expired whenever its TTG value is less than the minTTR to reach b (no interface can physically deliver it on time). For each bin, we first walk the critical and latency-sensitive queues to move expired packets into the Zombie queue.

A packet may have expired because there is no feasible path to the destination. In the case of ConditionalDAG-based anti-circulation forwarding, a path is deemed feasible if 1) none of the GNAT nodes along the path have already been visited, and 2) the path admits a TTR that is less than the packet's TTG. The ConditionalDAG approach computes the lowest-latency route using Dijkstra's algorithm and excluding already-visited nodes as potential relays. This solution is consistent among all nodes in the network (barring transient inconsistencies during the propagation of latency updates), although it can be computationally expensive.

While the ConditionalDAG-approach allows a node to compute the full path to the destination, it faces severe scalability constraints with network size. HeuristicDAG approximates the ConditionalDAG behavior by excluding routing packets to next-hops that have already been visited without considering the entire path a packet has taken.

Figure 9 and Figure 10 below illustrate how the time-to-reach a destination is computed. The forwarding node N computes the lowest-latency path from each of its neighbors to the destination, making sure to exclude either 1) all nodes (Figure 9) or 2) only neighbors (Figure 10) that have already been visited. All path computation excludes the links from the node to the neighbors to ensure forward progress toward the destination. A history vector inside packets marks the nodes in the network that have already been visited. Note that the lowest-latency path is a per-packet solution: two packets to the same destination but with different history vectors may lead to two separate lowest-latency path computations. A cache stores the calculated results to avoid costly per-packet recomputations.

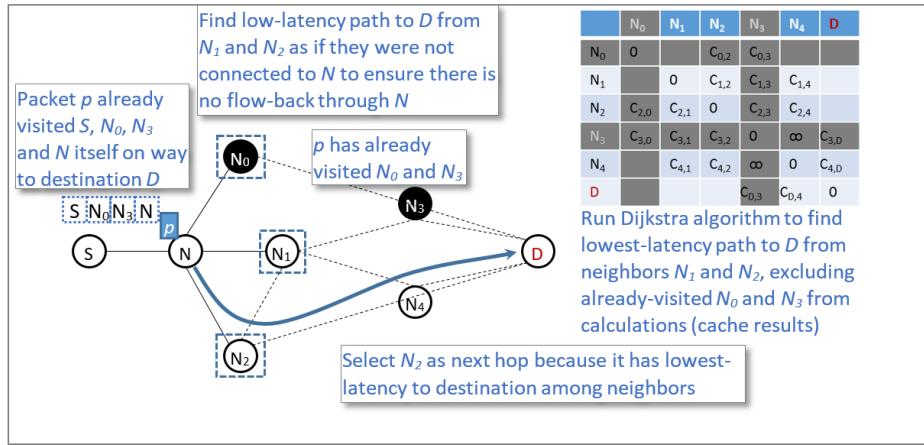


Figure 9. ConditionalDAG - The node selects the neighbor with the lowest-latency path to the destination, avoiding all nodes that the packet has already visited.

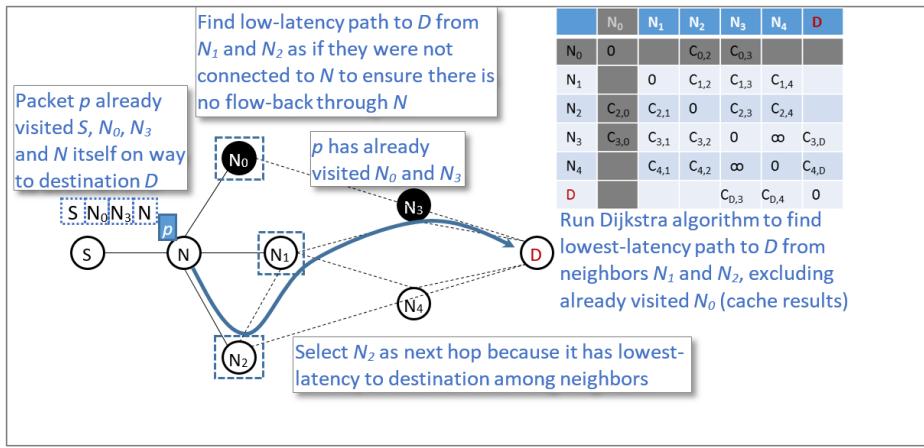


Figure 10. HeuristicDAG - The node selects the neighbor with the lowest-latency path to the destination, excluding only neighboring nodes that the packet has already been visited.

At this point, the only packets remaining in the low-latency and critical queues can reach their destination through at least one of the interfaces.

3.3.9.5.2.2 Move Dead-ended EF Packets to the Critical Queue - S2

If we are using the HeuristicDAG algorithm, some packets may end up in a dead-end state because the full network state is not taken into account when computing routes to the destination. Being in a dead-end state means that a packet's history vector precludes sending the packet to previously visited neighbors (as indicated by the history vector) yet those neighbors are the only means by which the packet may reach its destination within its combined forwarding path and latency constraints. We determine whether each packet in the low-latency queue has at least one feasible path to the destination that does not start with an already-visited neighbor.

Identification of dead-end packets is somewhat complicated since dead-ending can occur as a result of a packet staying too long in a queue rather than something that is true when the packet is first enqueued. For example, consider the case where a low-latency packet is enqueued with a

history vector that 1) precludes the use of a neighbor with the lowest latency path to the destination, but 2) allows the use of a neighbor with an otherwise acceptable latency to the destination that is not blocked by its history vector. Here the packet is not enqueued in a dead-end state; however, after sitting in the queue for some amount of time, the neighbor that was allowable by the history vector can no longer be used to deliver the packet on time due to a subsequently reduced TTG. Hence this processing must be done each time the packet selection method is called.

Dead-end packets for bin b are positively identified when their TTG value is less than the set of TTR values for each path controller connected to a neighbor not already marked in the packet's history vector — none of the allowable interfaces can be used to deliver the packet on time.

Dead-end packets are moved to the critical queue C(b).

This step and the next is absent from base forwarding or ConditionalDAG forwarding since latency is not considered with the former, and dead-ends are rare with the latter.

3.3.9.5.2.3 Handle Critical Packets with Latency Solution - S3

Critical packets are in a very precarious situation: all the feasible paths to the destination start with an already-visited neighbor and additional delays could cause the packet to miss its deadline. For packets in the critical queue, greatest-gradient forwarding does not apply: instead, the packet will follow the lowest-latency path to the destination.

We reserve these actions for low-latency critical packets to adhere to backpressure gradient-based forwarding rules (we assume only a small portion of our traffic is low-latency), that is, base and ConditionalDAG anti-circulation exclude this step altogether.

If the critical queue is not empty, the packet with the tightest deadline is forwarded on the path controller with the smallest latency to the destination and the forwarding algorithm can stop here. Otherwise, we investigate gradient-based solutions.

3.3.9.5.2.4 Constructing Forwarding Gradients for Unicast Destinations - S4

In this step, there are no transmit solutions for packets in the critical queue or it is empty.

We compute gradients along a matrix of B bins by I' available path controllers ($I' \in I$), initialized to 0. Path controllers whose transmit buffer is above a low-water level are considered unavailable for transmission, and they are omitted from the gradient computations (0-gradient).

For each bin b and path controller i, the gradient is the difference between the node's potential and that of the neighbor at the other end of i. A node learns of its neighbors' queue potentials via QLAMs. A queue potential is the sum of a) the queue depth for destination b, b) the virtual queue depth to b, and c) a time-in-queue-derived scalar computed by the No Packet Left Behind extension to basic backpressure.

The gradient is further modified whether path controller i goes to the destination b: if so, the neighbor's potential is 0 and the gradient is simply the potential of the node itself. Otherwise, the

gradient is set to 0 if it cannot exceed a hysteresis value, usually set to a few dozen bytes. The hysteresis value is the minimum number of bytes that a gradient must reach in order to forward packets and is intended to help curb flow-back.

Gradients are inserted in an ordered queue by decreasing value.

3.3.9.5.2.5 Constructing Forwarding Gradients for Multicast Destinations - S4a

Computation of gradients for multicast bins is slightly more complicated and works as follows.

First we need to provide some details describing on how multicast is supported within GNAT. We start by assuming there are some number M different multicast groups, where each group m has a set of destinations D_m . Here D_m is identically the set of Bin IDs of edge GNAT nodes supporting those multicast receivers that are members of the multicast group.

When handling multicast packets, GNAT adds a header to each packet specifying the set of destination Bin IDs deD_m to which the packet should be delivered. At the GNAT node nearest the sender, the header specifies all destinations in the group. Subsequent forwarding operations will, in general, specify subsets of the group destinations vs. all destinations in D_m .

Next, each node with Bin ID n maintains a separate packet queue and a vector of queue lengths q_{n,m,deD_m} for each multicast group m . When a multicast packet is received, the packet length is added to the queue length for each destination specified in the packet header. When a multicast packet is sent, the packet length is subtracted from the queue length for each destination specified in the packet header. Examples of these operations are shown in Figure 11.

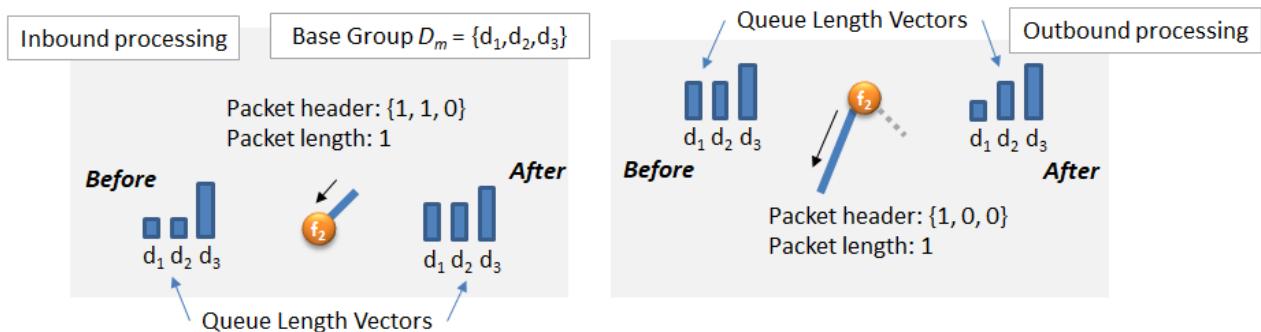


Figure 11. Before and after queue length vector values resulting from receipt of a multicast packet (left) and the transmission of a multicast packet (right).

As described earlier, each GNAT node with Bin ID n periodically reports its queue length vector q_{n,m,deD_m} for each multicast group m to each of its neighbors. The multicast gradient $g_{n,m}$ at local GNAT node l to neighbor node n for multicast group m is then given by

$$g_{n,m} = \sum_{d \in D_m} \max((q_{l,m,d} - q_{n,m,d}), 0)$$

This gradient term, which is a scalar, can now be compared to all other unicast and multicast gradients for the purposes of neighbor and destination (unicast) or group (multicast) selection.

Note that if a packet is selected based on this gradient, it is forwarded to neighbor n and is assigned a multicast destination vector with "1"s in each position d where $q_{l,m,d} - q_{n,m,d} > 0$. Example queue length vector reporting, gradient calculation, forwarding decision and resulting queue lengths are show in Figure 12.

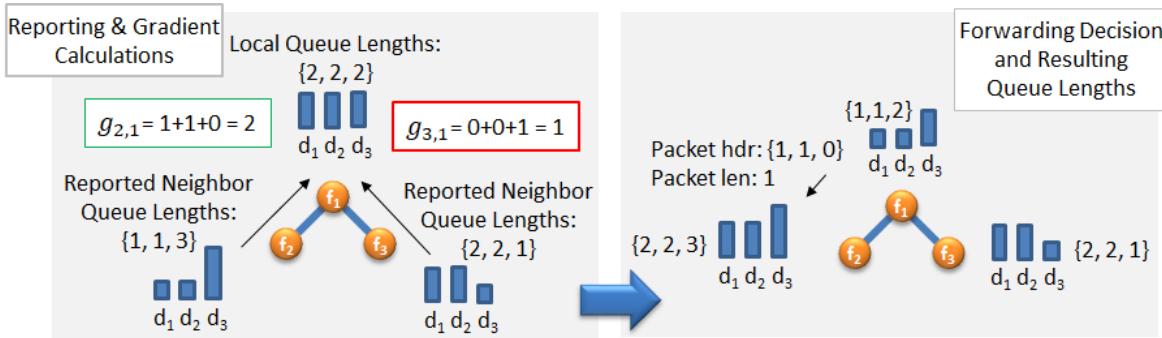


Figure 12. Example queue length vector reporting and gradient calculations made at local forwarding node f_1 with respect to two forwarding neighbors f_2 and f_3 (left), and the subsequent forwarding decision and post-forwarding local queue length vector (right)

We note that in Tracy Ho's original paper ("Dynamic Algorithms for Multicast with Intra-session Network Coding"), once a forwarding decision has been made, the packet sent is a *network coded* version of the packets received. For example, suppose a forwarder receives a packet containing the bit vector $\{0,1\}$ from one interface and a second packet containing the bit vector $\{1,0\}$ from a different interface. The gradient may be such that a packet should be sent to $\{1,1\}$, which the forwarder does by combining the contents of the two packets as a random linear combination – i.e., implementing a network coding of the two packets -- then sending the single combined packet.

In our implementation, since minimizing delivery delays for latency-constrained packets is a primary focus, we avoid network coding and instead perform a search within the packet queue to find the packet which best matches the gradient. The motivation here is that:

- Our best match approach avoids any decoding delays that otherwise result from network coding. Network coding requires receiving sufficient degrees of freedom (i.e., number of independent packets) to support decoding. With random linear network codes (typically used in implementing network coding) until such number of packets are received, none of the packets can be decoded or released; hence the first packet in a network coded group of packets is delayed at least by the last packet to arrive from the group. When packets taking multiple different paths (an intrinsic characteristic of backpressure forwarding), the decoding delay will be gated by the longest latency path taken.
- Our best match approach allows discarding any packet (say, due to expired TTGs) without needing to worry about the downstream impact -- i.e., we do not need to ensure that

all packets (or a sufficient number of random linear combinations of all packets) are delivered to each receiver. All packets received by the destination enclave can be forwarded to application hosts immediately. This characteristic is much more consistent with applications that use best-effort transport where reliability is given up in favor of lower delivery latency.

- Our best match approach avoids the "entanglement" caused by network coding, and so both enables and facilitates sender-based selective multicast delivery. For example, suppose we use multicast to implement a publish-subscribe capability that supports multi-temporal resolution publishing: a capability that directly supports distributed simulation environments where some subscribers need very frequent state updates and others need state updates less frequently. In this situation the sender can specify which subscribers are to receive any given packet simply by specifying those destination GNAT nodes in the multicast support header. If network coding were used, these two different packets destined for two different destinations may inadvertently be combined somewhere within the overlay network, with the result that either 1) the received combined packet is not decodable, since at least two different linear combinations of these packets must be sent to each subscriber in order to decode, or 2) additional, extra packets get delivered to both subscribers as required to support decoding and thereby wasting network capacity.

3.3.9.5.2.6 Handle EF Traffic with Gradient-Based Solution - S5

Note that this step is completely omitted in base-forwarding where there is no EF traffic.

Finding a Fit - S5.1

We explore the list of ordered gradients from largest to smallest, and look for packets that meet the latency constraints on the corresponding path controller i to destination b . For a given largest gradient, we inspect the $L(b)$ queue for packets that can be delivered in time via interface i , i.e., for packets with TTG values larger than the time-to-reach via interface i , and with history vectors that do not include i . Candidate transmit solutions are added to an ordered list in increasing TTG value such that the first packet in the list has the tightest deadline.

Non-Blocking Search - S5.2

We allow looking past the first element in queue $L(b)$, where the first packet may already have visited i and therefore not be eligible to be sent on this interface. Packets located farther down the queue are considered for transmission so that the packet at the head of the queue not block transmissions for the largest gradient. The queue depth search is configurable, but is set to 5 packets by default.

Multi-Packet Dequeue - S5.3

If there is at least one candidate matching the largest gradient, we select enough packets from the candidate list to either:

- fill the path controller's transmit buffer, which is configurable and set to 6,000B by default,

- be equal to the difference in bytes between the largest gradient and the next (not including gradients of equal value),

before the algorithm stops.

3.3.9.5.2.7 Handle latency-insensitive traffic with gradient-based solution - S6

In the absence of a feasible low-latency packet transmission, we explore the S(b) and Z(b) queues for transmit candidates. Again, we explore gradients in decreasing value to preserve largest-gradient forwarding, and search for a possible candidate to send. Suitable packets have not already visited the path controller under consideration, but latency is ignored. Candidates are added to a list such that packets to be sent directly to the destination b are listed first.

Again, we allow searching deeper in the queues so as not to block the following packets from being considered for transmission if the first is not suitable.

We dequeue enough packet candidates to fill the path controller's transmit buffer or be at least the difference in bytes between the two largest gradients.

3.3.9.5.3 Algorithm Variants and Code Organization

All Backpressure Forwarder algorithms are unified under the same class and file (`backpressure_dequeue_alg.cc`). A `ConfigInfo` object is used to initialize the algorithm to conduct backpressure forwarding using the Base algorithm (no latency-considerations) or LatencyAware algorithm. In its LatencyAware configuration, the algorithm can be configured with additional features like the ability to search deeper into the packet queues for solutions to the transmission (`QueueSearchDepth`), and to select one of two packet anti-circulation methods (`AntiCirculation`).

A large portion of the BPF's `FindNextTransmission` method is common to both Base and LatencyAware forwarding algorithm variants. Common aspects include computing gradients for every path controller with a transmit opportunity to every destination bin (S4), and searching through the forwarding queues in order of decreasing gradient until candidate packets are found (S6).

LatencyAware and Base algorithms primarily differ in how packet queues are maintained and the order in which packets are considered for transmission.

3.3.9.5.3.1 Base Forwarding

The Base Forwarding algorithm does not consider latency; hence even if a packet is marked for low-latency delivery and has a non-infinite time-to-go, the packet is simply enqueued with other latency-insensitive data traffic.

As such, the Base algorithm performs no cleanup of its queues since packets may not expire. As done for other instantiations of the algorithm, gradients are computed for all non-busy path-controller and bin combinations, and inserted in decreasing order (S4).

The algorithm then explores gradients in decreasing order in search of packets that will match. If the current gradient is negative or strictly less than the previous gradient, the search for matching packets stops. In the case of the Base algorithm, matching a packet is simply finding one that goes to the destination bin (S6).

The search for a matching packet simply looks into the corresponding bin queue (it cannot be empty, or there would be no positive gradient to this bin). Therefore, the first packet in the corresponding queue is added to the pool of candidates, which may expand if other gradients of equal value are considered.

The algorithm selects enough candidate packets to fill the path controller's transmit buffer or to be at least equal to the difference between the largest gradient and the next.

3.3.9.5.3.2 *LatencyAware Forwarding*

This variant of the FindNextTransmission method starts by cleaning up queues (S1). Here, packets that have expired or whose time-to-go is less than the lowest latency path, are Zombified, *i.e.*, they are moved to a latency-insensitive queue and marked to be dropped at the destination. Other low-latency packets receive a different treatment depending on the type of anti-circulation enabled to prevent packets from going through the same node multiple times.

The algorithm then builds gradients and orders them (S4). It then considers the low-latency traffic first (L queue), trying to find a matching packet for a given gradient (S5). A matching packet is defined differently on the type of anti-circulation involved. Matching packets are added to a list, favoring direct transmissions. If at least a match is found, the algorithm stops here and one or several packets are offered for transmission. Otherwise, “normal” and Zombie traffic queues are considered in the search for a match (S6).

3.3.9.5.3.3 *ConditionalDAG Anti-Circulation*

The ConditionalDAG approach eliminates paths that include already-visited nodes at the sender. We artificially disconnect the nodes that were already visited, as if they were not reachable. This means that the computed latency values for each neighbor indicate the lowest latency to a destination bin routing through never-before-visited nodes. If all nodes possess the same latency information, the latency graph generated in this manner is unique and known everywhere.

This also means that finding a match to send on a given path controller is packet-dependent, including within the same bin queue. A cache stores the lowest latency to all destinations to avoid per-packet recomputations and is indexed by all the values taken by history vectors. The latter generates large cache sizes in large networks.

Following clean-up (S1) and gradient build-up (S4), low-latency traffic is serviced first, in search of a match for the greatest gradient (S5). The packet-matching method looks for packets within QueueSearchDepth of the head of each queue (S5.1), five by default. This allows a greater chance of finding candidates for transmission (S5.2). Naturally, if a packet has already visited the remote end of a path controller, it does not qualify as a match. The same is true if the packet's

computed latency is infinity through that path controller, *i.e.*, it either goes through already visited nodes or there is no path to the destination. If a match is not found, the next packet in the queue is inspected. Otherwise, the packet is marked for future dequeue and added to the pool of candidates in increasing order of time-to-go value.

If, at this stage, the packet pool is not empty, we compute the number of bytes to be dequeued at once to be the smaller of the remaining transmit buffer size and the difference between the two largest (non-equal) gradients. We dequeue packets in the limit of this number of bytes (S5.3), and the method ends. Otherwise, the same packet-matching search happens for the normal and Zombie queues (S6).

3.3.9.5.3.4 HeuristicDAG Anti-Circulation

In this approach, the latency cache is not subdivided by history vector combinations. All packets are assumed to have visited no other node when making requests to the cache—with the caveat that they not be sent to previously-visited neighbors.

The reasoning behind this approach is that packets cannot be sent to neighbors they have already visited and packets with no gradient option (because all neighbors have been visited) are treated as special cases to be forwarded along the lowest-latency path.

Queue clean up (S1) involves one more step than with the ConditionalDAG approach (S2). The algorithm evaluates if packets still have gradient transmit opportunities. Each low-latency packet is inspected to determine whether it still has at least one non-already visited neighbor with suitable latency to the destination. If not, the packet is history-constrained: it will not be sent along a gradient and is moved to the critical queue during clean up. The critical queue is then checked for the packet with the smallest time-to-go, which is selected to be sent out on the path controller with the smallest latency to the destination (S3). The method stops if a packet is sent along the lowest-latency path.

Otherwise, the algorithm continues in gradient mode (S4) as with the ConditionalDAG approach: low-latency packets are inspected for a match of the largest gradient first (S5). If none is found, the search is carried over the normal and Zombie queues jointly (S6). If a send opportunity exists for a normal packet, the Zombie candidates are not considered.

3.3.9.5.4 Packet Transmission on CATs

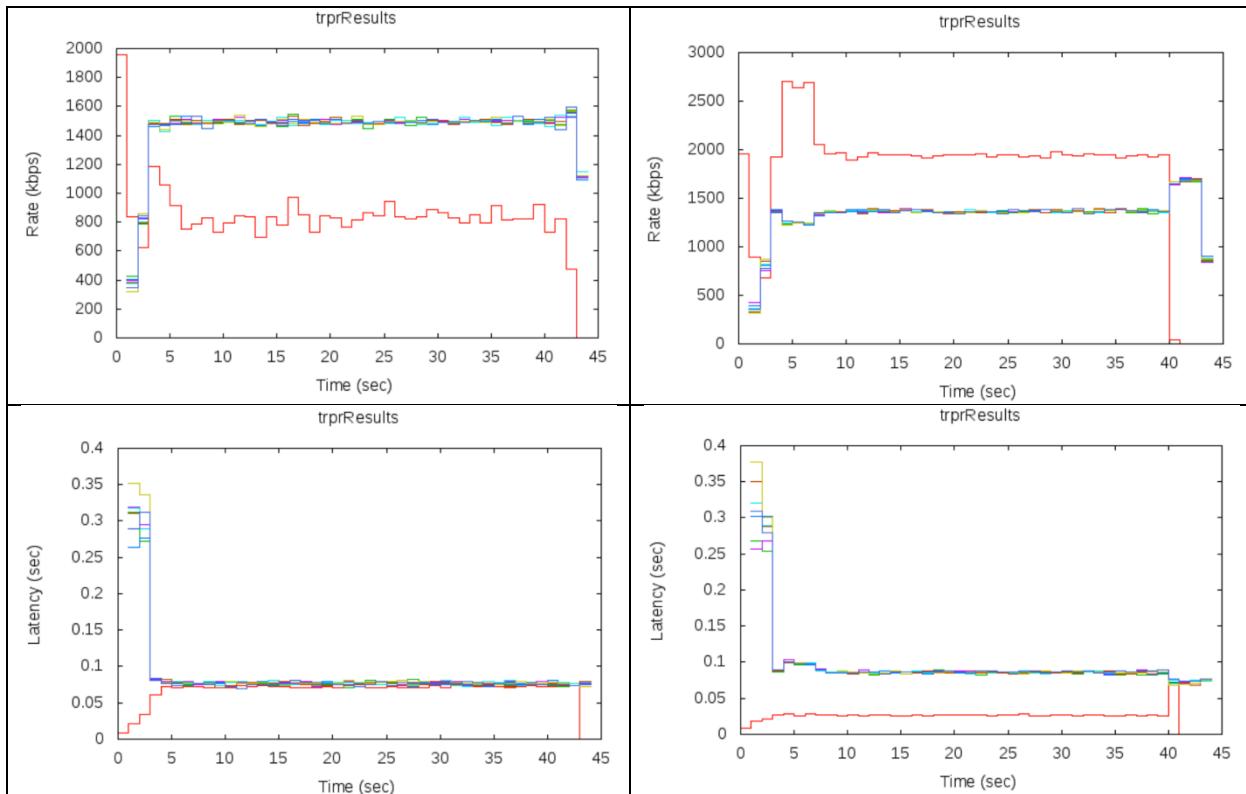
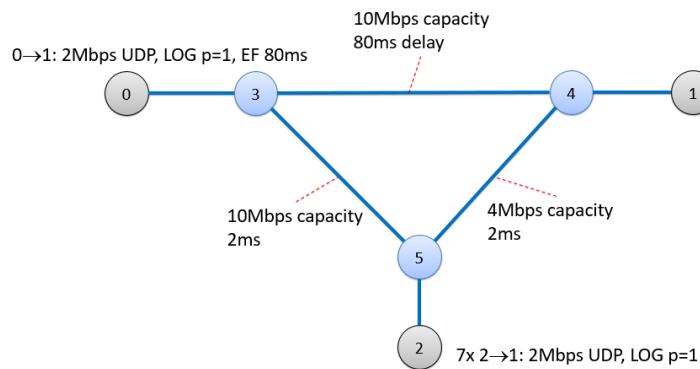
CATs estimate available link capacities and control transmission rates accordingly. CATs were implemented so as to support various congestion control protocols, including COPA and Cubic. COPA is a protocol that pushes back on the backpressure forwarder based on packet inter-arrival times. This ensures that the load not exceed the channel conditions. CATs also provide round-trip-time estimates so the backpressure forwarder can learn of path delays / time-to-reach.

CATs implement adjustable reliability for the delivery of packets, for instance QLAMs vs. data packets.

3.3.10 Hierarchical Forwarding

Latency-Aware Forwarding uses *hierarchical forwarding* to protect latency-sensitive traffic's forward move when the latency-feasible paths are against the gradient.

Consider the example below in which two sources send latency-sensitive and latency-insensitive traffic to the same destination. The latency-sensitive traffic has only one feasible path 3-5-4, whose capacity is so limited that the latency-insensitive traffic must be displaced onto path 5-3-4. The latency-insensitive traffic will create a strongly-negative gradient from 3 to 5, which could block the forward progress of latency-sensitive traffic.



Without hierarchical forwarding (left), we observe that the single Low-Latency flow tops at a rate of about 800kbps, which is far below its expected rate of 2Mbps. Other Latency-Insensitive flows are able to capture the remaining capacity and deliver traffic at around 1.5Mbps. Latency is around 80ms for all flows, in what is probably a long queue wait (80ms) for Low-Latency traffic.

With hierarchical forwarding (right), we now observe that the single Low-Latency flow receives the expected rate of 2Mbps and that its latency is far below 80ms. Other flows are delivered with a rate of around 1.4Mbps and a latency of a little above 80ms—that is, most of these packets do flow over the 'slow' 80ms link.

This clearly indicates that the Low-Latency traffic is prioritized over the Latency-Insensitive traffic. Hierarchical Forwarding considers Low-Latency gradients first (here, the flow from Node 0 to Node 1). From a gradient and an operational perspective, the BPF behaves as if Latency-Insensitive traffic is not present in the network when handling Low-Latency traffic.

3.3.11 Known Limitation: Partial Starvation of Traffic Generated by a Source of Latency-Heterogeneous Flows

One known limitation of the current Latency-Aware Forwarding algorithm is partial starvation of flows with higher deadlines to the latency-feasible capacity of flows with a tighter deadline.

Consider a source generates two latency-sensitive flows to the same destination. Flow 1 has a deadline of 200ms and the network has a capacity of 1Mbps over all 200ms-feasible paths. Flow 2 has a deadline of 1s and the network has a capacity of 10Mbps over all 1s-feasible paths. Our admission controller limits Flow 2 to around 1Mbps, which is the admission rate of the flow with the tighter deadline.

Flow 1 builds queues commensurate with its rate of 1Mbps (assuming no loss in the network). Since they go to the same destination, Flows 1 and 2 are admitted at the same rate, *i.e.*, 1Mbps. Therefore, Flow 2 fails to use 9Mbps of the network capacity and is admitted at a rate of 1Mbps only.

The same holds true if the long-deadline latency-sensitive flow 2 is replaced with a latency-insensitive flow (or its deadline is set to infinity)

3.3.12 Queue Depth Smoothing

GNAT includes two algorithms for queue depth smoothing, Heavyball and Aperiodic Queue Averaging (AQUA).

Heavyball is an implementation of the algorithm introduced in "Heavy-Ball: A New Approach to Tame Delay and Convergence in Wireless Network Optimization" by Jia Liu, Atilla Eryilmaz, Ness B. Shroff, and Elizabeth S. Bentley (2016). Instead of using exact queue depths to compute the gradients, heavy ball uses the formula:

$$\text{new_weight} = (\beta * \text{previous_weight}) + \text{current_queue_depth}$$

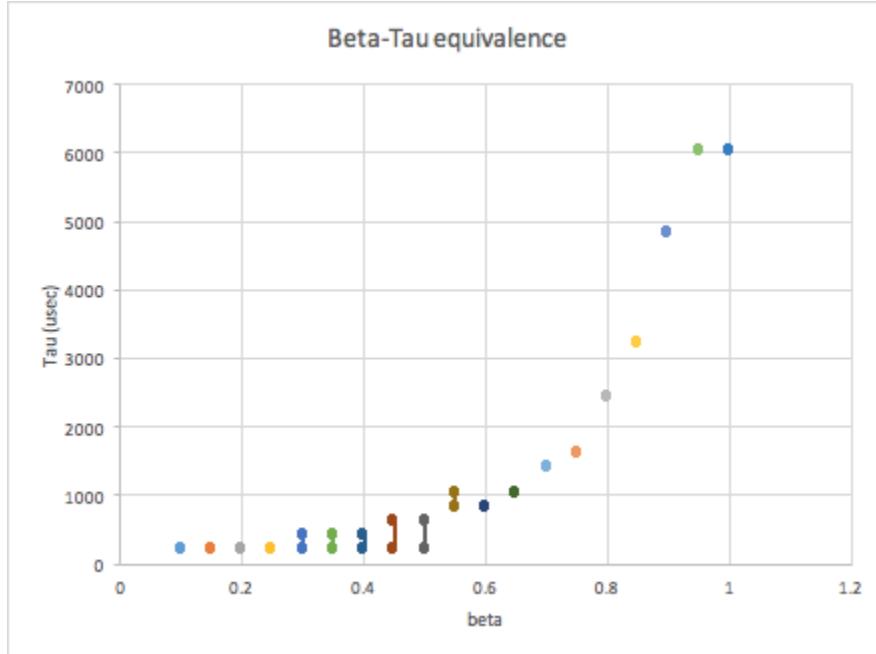
to periodically recompute weights (at a scheduled interval). The previous weight provides a "momentum" term (faster convergence), while the current weight adds a "potential" term (reaction). β provides the ability to trade-off between convergence and reaction time. High values of β result in faster convergence and more stability at the expense of reaction time and network utility optimality. In addition, when β is higher, the physical queue lengths end up being smaller by approximately a factor of β , thus reducing delay.

We observe that the rate computed by the backpressure forwarder when using HeavyBall is approximately pk/w for each log utility flow (where p is the priority, k is the system-wide scaling constant, and w is the HeavyBall weight). Multiplying pk/w by $(1-\beta)/(1-\beta)$ gives rate = $p*k*(1-\beta)/w*(1-\beta)$. By expanding w , we can see that $w = \sum_{i>=0} \beta_i Q_i$, where Q_i is the queue depth at i intervals ago, and if we compute $w' = \beta * \text{previous_weight} + (1-\beta) * \text{current_queue_depth}$, then w' expands to $\sum_{i>=0} \beta_i (1 - \beta) Q_i = (1 - \beta) w$. This means the computed rate will be the same if we use w' as the weight and a scaling constant $k' = (1-\beta)*k$. This makes the relationship more obvious between the convergence factor (β) from the lower delay (K).

Based on the above observation, GNAT also includes the **AQUA** algorithm, which takes advantage of the fact that w' is a exponentially weighted moving average to remove the need for a periodic computation interval. Instead, AQUA uses an event-driven EWMA computation paired with explicitly reducing K . The event-driven EWMA computation used in AQUA (shown below) is based on the paper "Algorithms for Unevenly Spaced Time Series: Moving Averages and Other Rolling Operators" by Andreas Eckner, First version January 2010, Latest version August 23, 2015.

$$\beta = e^{-(\text{current_computation_time} - \text{previous_computation_time})/\tau}$$
$$\text{new_weight} = \beta * \text{previous_weight} + (1 - \beta) * \text{current_queue_depth}$$

Thus, instead of specifying (configuring) a value for β as in HeavyBall, when using AQUA, we specify the value for the time normalization period, τ , which is used to compute β . The relationship between τ and β is given by the graph below, and discussed in [Smoothing for Queue Depth Management](#).



Instead of statically configuring τ , we can improve the performance and remove queue depth oscillations by using the period of oscillation (computed using an FFT) as the value of τ . This is discussed in detail in [Queue Depth Oscillation Reduction](#).

3.3.13 ASAP: Adaptive Starvation Avoidance Protocol

Backpressure sends packets from whichever queue has the largest difference in queue depth between the local GNAT node and a neighboring GNAT node. This works well as long as the queues are fed at a rate indirectly proportional to their depth. (GNAT admission control admits packets at a rate indirectly proportional to queue depth, so most situations work correctly.) However, there are circumstances where short queues cannot lead to faster admission simply because traffic isn't generated at a fast enough rate by the source. Perpetually short queues due to low sending rates mean the difference in queue value between this and any neighbor will similarly remain small. Thus, the destination with a very short queue will never overtake other destinations in terms of difference in queue value, so packets will never be sent. Or, if packets are added consistently at a low rate, the short queue may eventually become large enough that packets will be sent, but building the queue could take a long time, leading to long latencies. Examples include SYN packets for new flows, a handful of remaining packets at the end of a flow, or steady-stream low-rate flows (which may be elastic or inelastic).

The GNAT low latency forwarding algorithm doesn't fix this, because packets are almost always sent first for the destination, path controller pair with the highest difference in queue value, regardless of latency requirements.

We address this problem with a protocol called ASAP, focused on detecting and reacting to starvation using artificial ("zombie") packets.

ASAP effectively solves the starvation problem by dynamically detecting starvation and adding virtual packets to artificially increase the queue depths. These virtual packets are dequeued last to ensure they do not remove available capacity from ongoing flows. Starvation is both detected and responded to by head-of-line queue delay. We define this delay as the amount of time the packet at the head of the queue has been at the head of the queue. Note that this is different than the amount of time the packet at the head of the queue has been in the queue. With this definition, the sum of head-of-line queue delay over approximately one queue-depth worth of packets is a measure of the total queue delay amortized over the individual packets.

Starvation detection is based on monitoring the head-of-line queue delay. When this delay surpasses a threshold, the starvation reaction component of ASAP is triggered, and virtual packets will be incrementally added to increase the gradient, with the number of virtual packets computed based on the continually increasing delay for the packet. When the packet is sent, no additional virtual packets will be added until/unless the next packet surpasses the threshold, which is less likely because the virtual packets remain in the queue.

A primary goal of ASAP is to do no harm when starvation is not occurring. Therefore, ASAP does not attempt to take action until starvation is detected. In order to be effective at detecting starvation under a wide range of network characteristics, ASAP computes a starvation threshold dynamically using estimated capacity, number of local queues, and maximum expected packet size.

In particular, ASAP estimates the maximum expected head-of-line queue delay for a packet. This is the serialization time of the packet onto the link times the number of queues the backpressure router must cycle through (since it's possible for multiple queues to use the same link as a next hop).

Let M be the max expected packet size in bits, Q be the number of queues the backpressure router is servicing (one per destination), and c be the expected capacity in bits/sec. We note that GNAT provides capacity estimates in real-time to ASAP and specifies a maximum packet size of 1500 bytes. Let α be a “safety factor” which forces ASAP to be conservative in taking anti-starvation action, buffering any temporary queue processing slowdowns or ordering issues. This is nominally set to 50 in the ASAP system based on experimental results. We can then compute $s(M, c)$ as the starvation threshold in milliseconds (note the 1000 multiplier) as follows:

$$s_\alpha(M, Q, c) = 1000 \cdot \alpha \cdot Q \cdot \frac{M}{c}$$

Once the head-of-line queue delay for a packet has crossed this threshold, ASAP will trigger the starvation reaction algorithm.

Once starvation is detected, ASAP responds by using virtual packets to artificially increase the gradient. At the heart of this algorithm is a function that converts queue delay into virtual packets (more correctly, virtual bytes). There are three major benefits to adding virtual packets in this way: (1) the delay-to-bytes function provides fine-grained control of the gradient and allows for rapid starvation relief, (2) by adding virtual packets to the queue, rather than including a delay

term in the gradient computation, ASAP dramatically simplifies the implementation and retains the low overhead “queue depth only” spirit of backpressure forwarding, and (3) virtual packets allow the artificially-increased gradient to remain in place even after a starved packets is transmitted, allowing future packets for the same destination to be transmitted more quickly.

ASAP uses a non-linear delay-to-bytes function, which rapidly increases as the delay increases. This increase ensures that starvation is alleviated quickly, since many messages, such as TCP control messages, must be delivered in a timely fashion for the protocol to function properly. Let d be the head-of-line queue delay, and $f(d)$ be the virtual bytes to add to the queue. Let a be a scaling constant. ASAP defines $f(d)$ as follows, though we conjecture that this function ought to remain flexible, and can even be defined differently at different nodes, in order to allow an operationally-appropriate trade-off between anti-starvation and throughput-optimality.

$$f(d) = \begin{cases} 0, & \text{for } d < s_a(M, Q, c) \\ \text{Min}(P, a \cdot d^2), & \text{for } d \geq s_a(M, Q, c) \end{cases}$$

Note that d is in ms and a is in (bytes / ms²). This allows the result of $f(d)$ to be in bytes. The a coefficient allows for scaling; in the ASAP system we set $a = 2$ based on experimental results. These bytes are not added all at once; instead, they are continuously added as d increases – in other words, at any point in time, a total of $f(d)$ will have been added to address the starved packet in question.

In the equation above, P is a cap on the number of virtual bytes to add, and is based on all computed local gradients. This is to prevent unbounded growth, which is particularly important if the threshold is high causing a large amount of virtual bytes to be immediately added when the threshold is crossed. P is computed such that it will make the starved gradient slightly higher (5% in our experiments) than all other local gradients, which is the maximum needed to unstarve the packet.

Once a queue is built up with virtual packets, these packets remain in the queue until they become the BPF-defined best choice for transmission. In other words, the queue depth will remain inflated until it becomes the highest gradient. At that point, virtual packets will be dequeued until that queue depth no longer has the largest gradient. If the system reaches an equilibrium state, these virtual packets cause the queue depth to stabilize at exactly the threshold where packets are not sent, but where any additional packet added to the queue will be sent quickly. Therefore, not only is the original starvation detected by ASAP starvation detection sent rapidly, but any later packets for the same flow will also be transmitted rapidly

3.3.14 No Packet Left Behind (NPLB)

ASAP was inspired by the starvation avoidance technique from the paper "*No Packet Left Behind: Avoiding Starvation in Dynamic Topologies*" by Vargaftik, Keslassy, and Orda from Technion, Israel Institute of Technology. In order to assess ASAP performance relative to NPLB, we also implemented NPLB and provide it as an option within the GNAT codebase. We briefly summarize the NPLB algorithm here

Instead of using the queue depth as the queue value for backpressure decisions as in traditional backpressure, NPLB uses a combination of the queue depth and the current queuing delay. The exact formula for queue value (ignoring heavyball and AQUA for the moment) is:

$$\text{queue_value} = \text{queue_depth} + (\text{delay_weight} \times (\text{stickiness_value} + \text{delay}))$$

where `queue_depth` is the destination-bin-specific queue depth in bytes (including all latency queues), `delay_weight` is a configurable parameter indicating how much delay should matter relative to queue depth (`Bpf.QueueDelayWeight`), and `delay` is the time in microseconds that the longest non-zombie packet has been sitting in the queue for this destination. The concept of "stickiness" is discussed below. A delay-weight of 1 makes delay weighted approximately the same as depth for a drain rate of 10 Mbps. Note that using microseconds vs another unit for the delay measurement can be factored out by using different `delay_weight` values.

The notion of stickiness is to slowly increase the queue value for a bin that has consistently high queuing delay. This speeds up the process for rebuilding queue value after a dequeue for a bin that has an ongoing low-volume flow, thus decreasing latency for that flow. Otherwise, every packet entering that queue would have to sit there until the delay gets high enough to counterbalance the difference in queue length between this bin and other bins.

The stickiness value is increased whenever the difference in queue delay between a dequeued packet and the packet behind it in the queue surpasses a configured threshold (and it is increased by the amount this difference surpasses the threshold). This was chosen to replicate the above-referenced paper.

In order to ensure that stickiness isn't permanent, we implement it by adding zombie packets. These will sit in the queue until (a) there are no non-zombie packets remaining and (b) this queue value's difference with the neighboring value is large enough that a packet should be sent according to the result of the forwarding algorithm. At that point, the forwarding algorithm sends as many zombie bytes as are allowed by the forwarding algorithm. Note that this will happen ONLY when the zombies themselves (i.e., the stickiness values) are built up to the point where this destination wins even with no non-zombie packets.

3.3.14.1 NPLB Impact

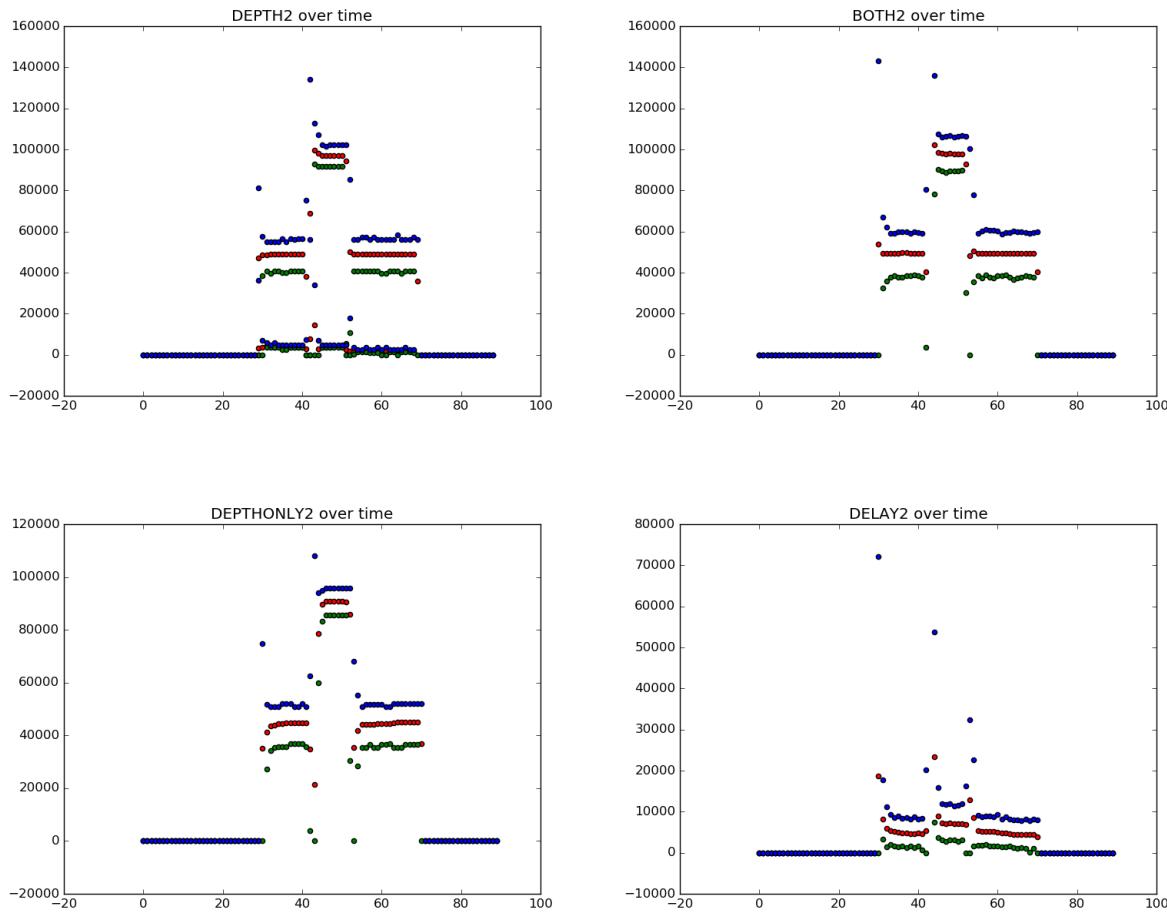
Under normal circumstances, the queuing delay will be proportional to the queue depth. Therefore, including delay as a portion of the queue value will have the same overall results (i.e., the same relative queue values) as if using only queue depth. This is shown in the graphs below.

Each of these graphs shows the minimum (green), average (red), and maximum (blue) value of the queue depth/delay/value for each second of an experiment run.

The graph on the left shows the queue depth when NPLB is disabled for an experiment that sends traffic from approximately second 25-65. The available bandwidth is cut in half from approximately seconds 40-50.

The second graph shows the combined queue value for the same experiment when queue delay is assigned weight 1. (i.e., delay and depth are approximately equally weighted.) The right two graphs show how that queue value is divided between depth and delay.

The pertinent observation from these graphs is that the queue value remains the same whether NPLB is enabled (the second graph, titled "BOTH2") or disabled (the first graph, "DEPTH2"). In other words, the fact that we are incorporating delay into the queue values does not affect the values used for distributed backpressure forwarding and admission control decisions. The right two graphs show that the delay counts for only a portion of these signaled queue values, while the rest is still the physical queue depth. These also show that the overall queue depth is slightly decreased, which should lead to slightly lower latencies.

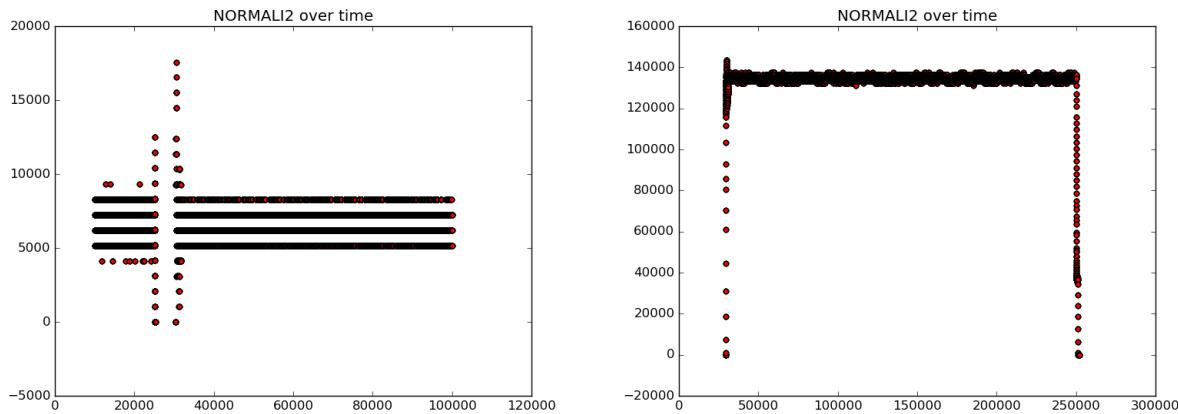


3.3.15 Zombie Latency Reduction (ZLR)

Assuming a nominal logarithmic utility function, the admission control performed by the UDP and TCP proxies sets the desired rate of a flow using the equation $r = pK/q$, where r is the computed rate, p is the flow priority, q is the local queue value for the destination bin, and K is a system-wide (network-wide) constant. For network tuning in general, there is a basic desired queue depth: long enough to handle small amounts of burstiness in the traffic without draining the

queue, but short enough to avoid adding significant queuing delay. The queue depth in a back-pressure network is a dynamic signaling mechanism, which leads to contention with the notion of a desired queue depth. In particular, we must set the constant K high enough that NO queue will be too short to handle burstiness, but the queue depths in the rest of the network are then determined dynamically based on the same value of K, and thus could be much longer, leading to significant queuing delay.

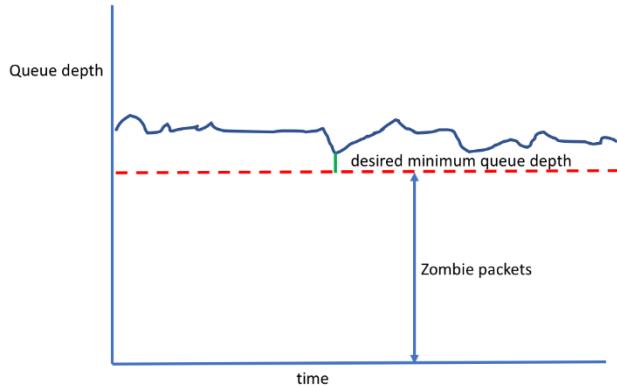
We can see this phenomenon in the following graphs. These show that the depth of one queue during an experiment with a steady but reasonably large amount of traffic. The graph on the left shows the queue depth when $k=1e11$. The graph on the right shows the queue depth when $k=1.5e12$.



In both cases, the queue depths are fairly steady. However, the value of the depths is significantly different, which will naturally lead to a much longer queuing delay when K is higher.

We observe that the deeper queues with higher K could be replaced with dummy packets (which we already have in the form of zombie packets) that are never sent. This would cause the actual packets to arrive, sit in the queue for a short time, and then be sent, all without (a) affecting the depths that are distributed and used by the backpressure algorithm or (b) requiring any coordination between GNAT nodes (since the shared values would still be identical).

However, this simple observation is not sufficient to implement a solution. Consider the illustration below.



In this case with a fairly stable queue depth, we can examine the desired minimum queue depth to allow an appropriate burstiness of traffic, and fill the remaining "floor" with zombie packets. By adding a zombie packet of the same size whenever we dequeue a non-zombie packet, we effectively keep a constant queue depth, which should then not change the send rate computed by the proxies. Packets added by a proxy will increase the queue depth and thus may very slightly decrease the send rate, but as soon as we've added enough zombies to fill the floor, the send rate will return to the expected value and the queue will (as desired) have a large floor of zombies (that are never sent) and a small dynamic queue of real packets that never drops below the desired minimum queue depth.

However, we still have the problem of how to compute this queue depth floor under different circumstances. The following theoretical queue depth over time graphs demonstrate this issue in a dynamic network.

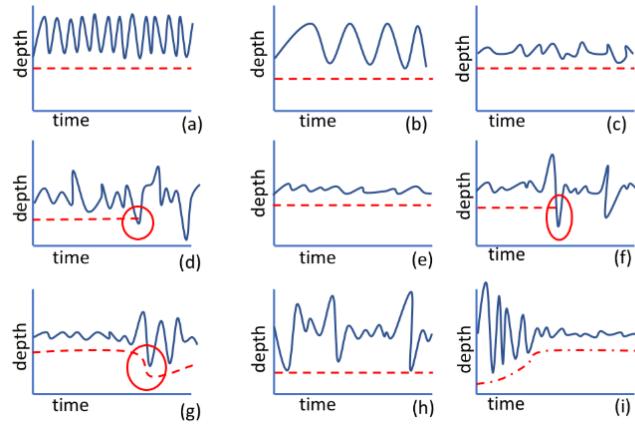


Figure 13. Hypothetical graphs of queue depth over time, demonstrating the challenge inherent in choosing the correct value for the ZLR floor. The red dotted lines show the "correct" answer for the how many zombie packets we want to fill. The red circles show points where we'd have to dequeue zombie packets. Graphs where the red dotted lines end partway through are cases where it's not clear from this small view of the data where the ZLR floor ought to be. These cases in particular demonstrate the difficulty of designing a robust algorithm for this purpose.

The red lines are drawn where (based on human observation) the queue depth floor ought to be. That is, we should be able to safely add enough zombies to fill that much of the queue with dummy data. However, even if our algorithm could compute these red lines, we'd still hit some points (circled in red) where zombie packets would have to be sent to maintain the backpressure equations as if these were real packets.

Example (e) looks much like the stable queue in the experiment shown above. In this case, as long as we have a threshold of non-zombie packets that is high enough to cover the peaks in the varying queue depth, we could simply use the instantaneous queue depth minus that threshold as the floor. However, the other examples make it obvious that using an instantaneous queue depth will be problematic. In example (a), we'd be adding zombie packets whenever the queue depth heads towards its peak unless our threshold is large enough to cover the entire amplitude.

We therefore choose a floor (the queue depth up to which we'll add zombie packets) by looking at the minimum queue depth over some window of time into the past. The necessary window size depends on the traffic (for example, the necessary window in example (a) is much smaller than in example (b)). Therefore, we use a dynamically-sized window based on observations of whether and how often we find the BPF sending zombie packets. If the window is too small (for example, using a window appropriate for example (a) when the queue depth looks like example (b)), then we'll end up adding zombie packets during a queue depth spike, and thus sending zombie packets during the corresponding dip. Therefore, sending zombie packets triggers an increase in window size. If the window is too big, we will allow a queue depth blip to affect the amount of zombies we add for too long, thus negatively affecting the latency. (For example, if the window in example (i) was too large, then the queue depth floor would not recover even once we reach a stable state.) We shrink the window size whenever we've gone a long enough time without sending a zombie.

The window size for the above theoretical queue depth graphs should converge to:

- (a) Approximately the frequency of the oscillations.
- (b) Adjust to the frequency of the oscillations, so that we converge to a larger window towards the start, and the window shrinks slightly over time.
- (c),(d),(e),(f) The length of the largest period from dip to dip.
- (g) The length of the largest period from dip to dip at the start, but growing larger when the large oscillations happen.
- (h) The length of time between the big dips.
- (i) The length of time between oscillations, which will naturally move the minimum queue depth value up over time as the network stabilizes.

When considering the minimum queue depth over that window of time, we need to subtract out any zombies that have been added during that window, since these are effectively chipping away at the remaining space under the red line.

When adding zombie packets, we also want to make sure we're in a period when the queue depths are generally stable or growing. Adding zombie packets when queue depths are shrinking will increase the likelihood that these zombies will eventually be sent.

To summarize, our ZLR algorithm does the following:

When dequeuing a non-zombie packet:

- Look at the minimum queue depth over the most recent window of time (minus any zombies added over that time). Is this value larger than our configured minimum threshold?
- Look at the recent rate of queue depth change. Is the queue depth growing or shrinking?
- If the queue depth is growing and the minimum queue depth over the window is larger than the threshold, add zombie bytes of the same size as the dequeued packet.
- If it's been at least the configured amount of time since we last adjusted the window size or sent a zombie, slightly decrease the window size.

When dequeuing a zombie packet:

- Slightly increase the window size.

Because of latency-sensitive forwarding, the ZLR zombie floor as described above does not reduce the queue delay of latency-sensitive packets. To illustrate this, consider a case where there is a large queue entirely made up of latency-sensitive packets. We'd like ZLR to add zombie packets to build up a floor, reducing the effective queue depth of the real packets. However, there are two reasons why this doesn't work in the latency-sensitive case. First, if latency-sensitive packets (the entire queue in this case) are being forwarded along a constrained path, while the zombies are being forwarded along the best available path, the ZLR-generated zombies will be transmitted as soon as they are created, and no floor will be built up. Second, even if the zombies *did* build up a floor and that floor successfully reduced the real-packet queue depth for latency-sensitive packets, the effective (real+zombie) queue depth would NOT be counted towards the latency-sensitive gradient because of hierarchical forwarding, so the latency-sensitive packets would be at a disadvantage with the ZLR floor compared to without it.

To counter these two problems, ZLR in IRON adds LS-zombies in addition to normal zombies. LS-zombies are different from other zombies in two ways:

- 1) They will never be dequeued if there are *any* latency-sensitive packets in the queue. This solves the first problem discussed above: these zombies will not be immediately sent but will instead build up a floor for latency-sensitive traffic.
- 2) They are counted as latency-sensitive for the sake of hierarchical forwarding. This solves the second problem discussed above: these zombies will be included in the latency-sensitive queue depth around which the system stabilizes. With this special treatment, latency sensitive packets and latency-sensitive zombies together create an echo of IRON's backpressure system as a whole.

LS-ZLR (Latency-Sensitive Zombie Latency Reduction) follows exactly the same rules as normal ZLR, except that decisions are made purely based on the latency-sensitive queue depths, and zombies that are added are LS-zombies. From an implementation perspective, this is just another

instance of the same code using different inputs for queue depths (so that these queue depths includes only latency-sensitive packets).

Like any other BPF algorithm, LatencyAware Forwarding computes the gradients to destinations on various available interfaces to neighbors and selects the transmit solution with the steepest gradient. An interface is deemed unavailable (or busy) if its transmit buffer is full. The core of the algorithm preferentially handles ordered latency-sensitive traffic to be sent over the largest gradient and, for each packet, reduces the set of next-hop neighbors to those with feasible paths to the packet's destination. We contend that the former is insufficient for most applications and networks and show that the latter improves latency-sensitive traffic delivery by 233%. We enhance this algorithm further by providing Hierarchical Forwarding, which supports latency-sensitive traffic delivery when the latency-feasible paths run against the gradient to the destination, as illustrated above in section 3.3.10. Our algorithm also reduces circulation by using a heuristic-based approach that excludes interfaces to previously-visited neighbors from the set of feasible paths. This simple description hides important requirements and insights. For instance, each packet is enqueued after having visited a different subset of nodes in the network, and therefore with different constraints or feasible paths to the same destination. This means that at any given point, the packet at the front of the queue may not have any positive gradient toward the destination and could block others with feasible paths from being forwarded.

4 UDP Proxy

4.1 Summary of Module Goals and Objectives

The three main goals for the UDP Proxy are: 1) providing an Admission Control capability for UDP-based applications to interface with the backpressure forwarded network, 2) reducing the packet loss rate and/or reducing delivery latency of existing UDP applications using erasure coding techniques, and 3) providing these capabilities without the applications having knowledge that the gateway is operating on their behalf or in fact, is even present in the packet delivery path.

Topologically, the UDP Proxy sits between two communicating hosts as shown on the left of Figure 14 below. From the perspective of applications, however, GNAT nodes appear only as intermediate routers in that no modifications to packet content or application processing are required. This application transparency is depicted on the right of Figure 14.

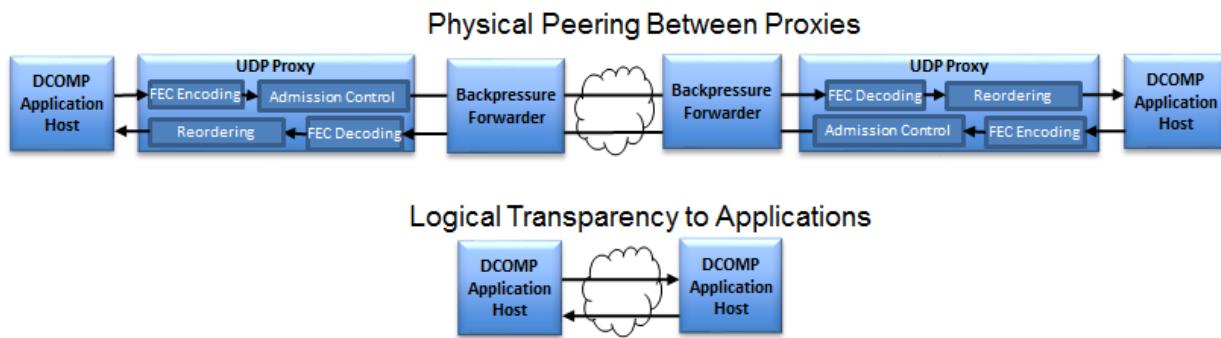


Figure 14. Transparent UDP Proxy relationship to network topology interconnecting two application hosts (top) and as seen by communicating applications on those hosts (bottom).

We note here that calling this module a UDP Proxy is somewhat limiting, since the UDP proxy is also used to handle IPSEC traffic, which is IP protocol 50 (ESP). It also supports UDP traffic embedded in VXLAN traffic by decapsulating the VXLAN packets, as well as multicast traffic encapsulated in PIM registration packets by decapsulating these packets and handling the multicast forwarding natively as well.

4.2 High Level Design

4.2.1 Adjacent Modules and Components

As shown in Figure 15 below, the UDP Proxy receives data packets from local UDP-based Applications and proxied data packets (from proxies in remote enclaves) from the local backpressure forwarder. It receives control information from: 1) the Admission Planner (in the form of utility functions and parameter settings); 2) the local backpressure forwarder in the form of queue length information; and 3) from proxies in remote enclaves typically embedded within the proxied packet headers.

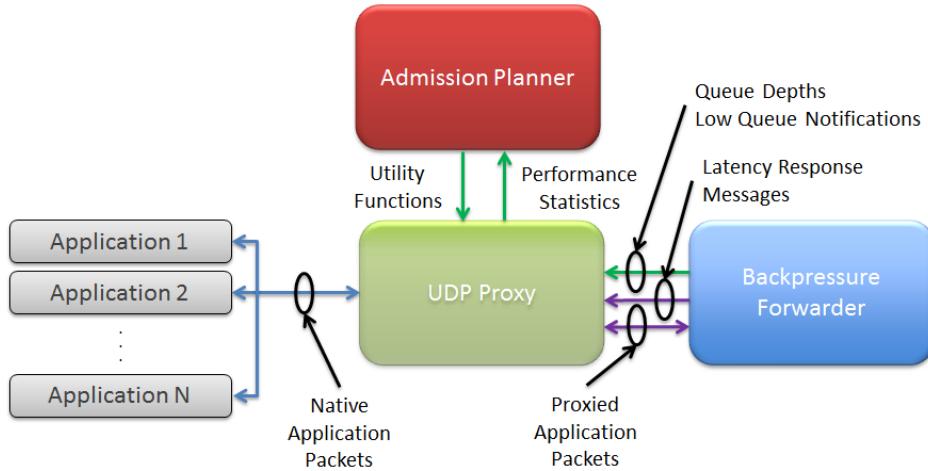


Figure 15. Relationships between the UDP Proxy, external applications, and supporting GNAT components. Blue connecting lines are application packets; green lines are local monitoring and control information; purple lines are proxied application packets and end-to-end proxy coordination messages

4.2.2 Input Sources and Information Needed

Control plane inputs include: 1) The backpressure forwarder periodically updates the shared memory regarding current queue depth information, 2) queries and assertions (gets and sets) regarding various processing statistics from the User Interface, and 3) directives from the Admission Planner. Directives from the Admission Planner may take the form of either 1) a new assignment of, or updates to, the utility function model and parameters used for a given four-tuple; or 2) (To Do) a termination command that prohibits further processing and forwarding for a given four-tuple.

Data plane inputs are either: outbound original packets from local application hosts, or inbound encoded packets from the local backpressure forwarder.

4.2.3 Output Destinations and Information Provided

Control plane outputs: UDP proxies do not provide control plane information to any other GNAT node components. It does however provide statistics on the various flows to the Admission Planner (if connected). It also will provide feedback to NACK-Oriented Reliable Multicast – NORM – based applications to improve NORM's congestion control processing and allow greater send rates to be achieved especially when NORM is operating over GNAT's multipath-forwarded network. Note that this requires using an enhanced version of the NORM code also developed under GNAT.

Data plane outputs include: 1) encoded application packets destined for the local backpressure forwarder, and 2) decoded application packets destined for local receiving applications.

4.2.4 Events

Queries and assertions (sets and gets) from the User Interface on timescales of the order seconds;

Notification of assignments or changes to assigned utility functions, on timescales of the order of application launch frequencies (10s of seconds to minutes)

Expiration of transmission timers as set by the admission controller for each flow; timescales are roughly proportional to the packet interarrival times of the source applications

Packet arrival, from either an outbound application or from the BPF for an unbound application. This can be in the order of microseconds, depending on the rate of the application.

Latency request message arrives from the BPF. LRM_s are generated when packets are dropped or arrive at the destination proxy.

Expiration of statistics reporting timer, on timescales of around a second. Updated statistics must be sent to the GUI via the Admission Planner.

4.2.5 Use Cases

UDP Proxy Startup - When first launched, the UDP Proxy reads a local file to retrieve any configuration information. It then retrieves any command line arguments that, if present, override corresponding configuration information from the file. Finally it loads the set of configured parameters into its configured state, overriding any default values. It configures the Linux forwarding engine to begin redirecting UDP packets meeting characteristics (as described by configuration parameters) to the UDP. Finally, it sets up a TCP server to accept connections on the control channel.

Application Packet Arrival: Send Side -- Using standard Linux packet redirection techniques, UDP packets from sending applications flow through the GNAT node towards the WAN are intercepted and turned over to the UDP proxy. Upon the presentation of a new packet to the proxy, it uses the four-tuple (source and destination port, source and destination address) to retrieve any existing encoding state. If no state is found, it creates an initializes a new state entry and stores it in its local cache. Using the encoding state, it then encodes the packet, and any resulting encoded packets generated by the encoder are delivered to a local admission control packet output queue for subsequent delivery by the proxy's embedded Admission Controller. It then notifies the embedded Admission Controller of a change in the queue state.

New Enqueued Packet Notification: Send Side -- Upon notification of a new packet in the admission control queue, the embedded admission controller that there is at least one packet waiting to be sent. If there is an admission already timer running for this flow then nothing needs to be done. If there is no timer currently running for this flow, one or more packets may be admitted and a timer will be set based on the calculated admission rate for the current queue size.

Encoded Packet Arrival: Receive Side -- Encoded UDP packets with local destinations within the backpressure forwarder are delivered to the UDP proxy for decoding by writing the packet to the shared memory and notifying the proxy. Upon the presentation of an encoded packet to the proxy, it uses the four-tuple (source and destination port, source and destination address) to retrieve any existing decoding state. If no state is found, it creates an initializes a new decoding

state entry and stores it in its local cache. Using the decoding state, it then decodes the packet if possible, and any resulting decoded packets generated by the decoder are delivered to a local output packet queue for reordering. In order to simplify operations, ordering information is carried along as part of the encoding header and is used for inserting the decoded packet into the reordering queue.

Decoded Packet Arrival: Receive Side – Once a decoded packet is placed in the packet reordering queue, the proxy determines whether the packet is at the head of the queue and can be released immediately, or whether the packet should instead be held in the reordering queue for some length of time awaiting the potential arrival of other decoded packets that may be earlier in the original packet sequence. Once a packet is released from the decoding queue, either due to time expiration, or due to being promoted to the head of the queue, all packets in the reordering queue with contiguous sequence numbers are also released.

Admission Timer Expiration – If an admission timer goes off, the proxy finds the encoding state associated with the timer based in the included four-tuple. The admission rate is calculated based on the current queue depths found in the shared memory. One or many packets may be admitted based on the admission rate, the number depends on availability and configured burst interval. An admission timer is then set to indicate when subsequent packets should be admitted to achieve the calculated rate.

Latency Response Message Arrival – Currently, latency response messages are only logged. In the future, this information can be sent to the admission planner to be used to inform flow admission decisions, including triage decisions if latency bounds are not being met as well as adjusting FEC settings if there are significant levels of end-to-end loss.

4.2.6 Performance Monitoring Statistics

Statistics available to the Monitoring and Control Interface include:

Summary statistics: Number of currently active send applications; total number of send applications handled since last statistics reset; currently active receive applications; total number of receive applications handled since last statistics reset;

Local send application statistics : indexed by four-tuple, start time; run-time; number of original input packets processed; number of encoded output packets generated; current encoding settings; current utility function assigned including parameters; recent average utility achieved; long term average utility achieved

Local receive application statistics: Indexed by four-tuple; start time; run-time; number of encoded packets received; number of decoded packets generated; current encoding settings; number of decoding failures; maximum difference between decoded out-of-order packet indices and associated delay.

4.3 Detailed Design

We first describe the mechanism used to intercept and modify packets as needed to perform the transparent erasure encoding and decoding functions. We then provide details of the internal state management and FEC processing approach need to accommodate a dynamic number of applications sending packets of different (and potentially varying) size at different rates over channels with potentially large variations in loss or delay characteristics.

4.3.1 Sender Side Processing

4.3.1.1 Getting Application Packets Into the UDP Proxy

Transparent intercept of application packets by the UDP Proxy is accomplished using a raw socket configured with Berkeley Packet Filters to intercept those packet types which can be handled by the UDP proxy, along with an associated *iptables* rule configured to drop the intercepted packets. The use of *iptables* is necessary since packets obtained via the raw socket interface are in fact copies of packets, so that without the drop rules the packets will also be delivered to the kernel.

The packet selection logic embodied in the Berkeley Packet Filter used by the UDP proxy is as follows (assume that the local interface IP Address is 10.1.3.1):

- Any UDP packets that are not being sent to the local interface address and not marked with a TOS value of 4 (which we use to bypass IRON/GNAT for demos) and not included in the user-identified bypass list and are not VXLAN packets carrying TCP traffic (these will be handled by the TCP proxy)

 $((\text{udp} \text{ and } \text{ip}[1] \neq 0x4 \text{ and not dst } 10.1.3.1 \%s) \text{ and not } (\text{udp dst port } 8472 \text{ and } \text{udp}[39]==6)) \text{ or}$
- Any ESP packets that are not being sent to the local interface address

 $(\text{esp} \text{ and not dst } 10.1.3.1) \text{ or}$
- Any PIM registration packets that are not being sent to the local interface address. pim registration packets encapsulate multicast traffic in order to deliver it (via unicast) to the pim rendezvous point. The UDP proxy decapsulates these packets and handles the multicast traffic directly

 $(\text{pim} \text{ and } \text{ip}[20]\&0xF==1 \text{ and not dst } 10.1.3.1)$

The first portion of the Berkeley Packet Filter specification above includes a *%s* item, which represents a user-configured bypass filter string (described in more detail in Section 3.3.1.2 below) that contains the information for the user-specified 5-tuples representing the packets for which the user does not want the UDP Proxy to process.

The corresponding iptables rules for dropping these packets (on a host with a LAN-facing eno2 interface and IP Address of 10.1.3.1) are given by:

```
iptables -A PREROUTING -t mangle -i eno2 -p udp ! -d 10.1.3.1 -j DROP
```

```
iptables -A PREROUTING -t mangle -i eno2 -p esp ! -d 10.1.3.1 -j DROP
```

4.3.1.2 UDP Proxy Bypass Filter Specification

As described above, all packets with a TOS value of 4 bypass the UDP Proxy. For applications and application hosts that do not easily support setting TOS values for transmitted packets, the UDP Proxy bypass filters provide for an additional mechanism for user-defined fine-tuning of the UDP Proxy's raw socket Berkeley Packet Filter to control the packets that are received and processed by the proxy. Bypass filters are represented as 5-tuples that have the following format:

```
protocol;saddr:[sport | sport_low,sport_high]->daddr:[dport | dport_low,dport_high]
```

where

- **protocol** identifies the protocol of the packet and **must** be either **tcp** or **udp**.
- **saddr** identifies source IP Address. A value of * may be used here to refer to any source address.
- **sport** identifies source port. A value of * may be used here to refer to any source port.
- **sport_low,sport_high** identifies a range of source ports.
- **daddr** identifies the destination IP Address. A value of * may be used here to refer to any destination address.
- **dport** identifies the destination port. A value of * may be used here to refer to any destination port.
- **port_low,dport_high** identifies a range of destination ports.

As an example, consider the desire to bypass all UDP packets with destination ports in the range 30900-30909. The following user-configured bypass specification achieves this:

```
udp;*:*->*:30900,30909
```

At run-time, the above bypass specification is converted into the following Berkeley Packet Filter string

and not (dst portrange 30900-30909)

which is then substituted for the %s item in the UDP Proxy Berkeley Packet Filter described above.

4.3.1.3 FEC Encoding Process

The FEC encoding submodule is responsible for 1) encapsulating application packets within an FEC control structure before handing them off to the Admission Controller submodule, and 2)

computing zero or more repair packets using the application packets and handing the repair packets off to the Admission Controller as well.

The encoding process begins when a UDP packet with a service port falling within one of the supported service port range is intercepted by the raw socket interface described above. It is then read from the raw socket by the UDP proxy, and its payload is distributed across one or more chunks of a dynamically assigned maximum payload size. A two byte chunk reassembly trailer is then appended to the end of each chunk packet, followed by a four byte FEC control trailer. The IP header for each chunk retains the original source address, destination address, and source port of the original UDP packet; however, the destination port is that of the assigned gateway service port. The length fields within the IP and UDP headers of each chunk are updated to reflect the new lengths, and the IP and UDP checksums are recomputed. Finally, the chunks are enqueued for use by the Admission Controller.

If the payload of the original packet is sufficiently smaller than the assigned maximum payload size for a chunk, the UDP proxy will instead attempt to merge multiple original packets into a single chunk packet, which we refer to as a *blob*. Under these circumstances, the FEC Gateway will first append a two byte blob header to the chunk containing the length of the original payload, followed by the payload itself. Including the length information allows the receiving FEC gateway to pull apart chunks consisting of multiple original packets into original packets of the correct size. The blob length information is stored in Network Byte Order within the blob header, as shown in Figure 16.

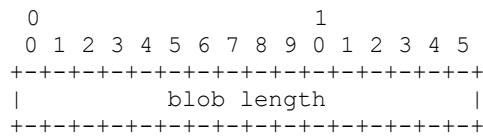


Figure 16. FEC blob header fields

If there is sufficient room within a chunk after appending the payload from an original packet for another payload of the same size, the UDP Proxy will *not* send the chunk, but instead will retain the chunk awaiting an additional packet. Once the chunk is filled (i.e., there is no longer room for an additional packet payload of the same size), the chunk trailer and control trailer are appended and the chunk is sent. If a partially filled chunk remains after some maximum hold time (described later in this document), the FEC gateway will append the trailers and send the (albeit smaller) chunk in the usual manner.

The FEC control trailer contains the information needed to reconstruct a group of encoded chunks from the mix of original and repair chunks received. The FEC control trailer consists of 1) a 1-bit field "A" indicating the packet type (that it is an original packet rather than a FEC repair packet), 2) a 1-bit field "B" indicating whether the decoder should preserve packet ordering, 3) a 1-bit reserved field "R", 4) a 5-bit slot identifier indicating the position of the chunk within the array of original chunks used for FEC repair chunk construction, and 5) a 24 bit FEC group identifier used to distinguish between encoded sets.

Note: In order to support handling of latency sensitive packets within GNAT, several fields have been added to the FEC trailer. Specifically, these fields include: 6) a time-to-go value (in units of microseconds) indicating how much time remains before the packet will be deemed unusable by the receiver, 7) a 4 bit source Bin ID (note: since this can be computed from the IP header, this is redundant information and can be eliminated in future releases), 8) a 20 bit packet ID that is used to validate references to this packet maintained in the BPF latency tracking structures, 9) an 8 bit unused field that acts as padding, 10) a 2 bit epoch field updated by each BPF that is used to verify matches in their latency tracking arrays, 11) an 18 bit record locator that is updated by each BPF to indicate where the packet information is stored in the latency tracking structure; and 12) a 12 bit unused field that acts as padding. This use of this latter set of fields (items 6-12) for low-latency processing is further described in the backpressure forwarding section, and is not further described here.

The layout of the FEC trailer used by the UDP Proxy is shown in Figure 17. Where fields span more than 8 bits, these fields are stored in Network Byte Order.

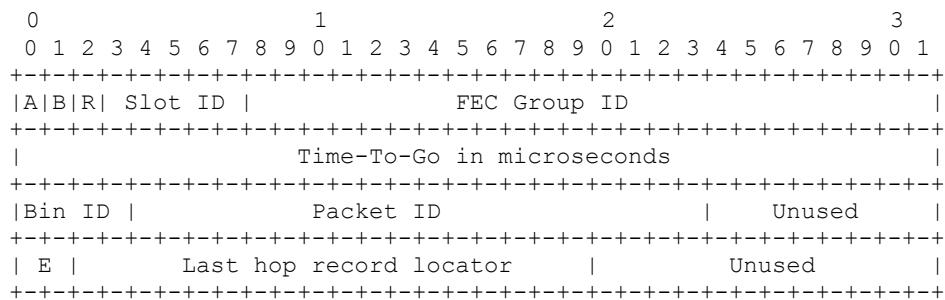


Figure 17. FEC control trailer fields

The FEC chunk trailer contains the information needed to reassemble a packet from its original or reconstructed chunks. The FEC chunk trailer consists of 1) a 1-bit field *isBlob* "C" indicating whether this chunk contains multiple original packet payloads (i.e., it is the result of a merge operation – *isBlob* = 1) OR contains either a single original packet or the result of a split operation (i.e., *isBlob* = 0); 2) a 5-bit field *Packet ID* indicating the order that the original packet was received in relative to the current encoding set (i.e., the encoding group) if the chunk contains either a single original packet or the result of a split operation (i.e., *isBlob* = 0), OR the order that the *first* of the multiple original packets in the chunk was received in if the chunk is the result of a merge operation (i.e., *isBlob*=1); 3) a 5-bit field *Chunk ID* indicating the order of the chunk relative to other chunks if the chunk is the result of a split operation, OR 0 if the chunk contains one or more original packets in their entirety; and 4) a 5-bit field indicating the number of chunks into which the packet was broken if the chunk is the result of a split operation or contains a single original packet (i.e. *isBlob*=0) OR the number of original packets in the chunk if the packet is the result of a merge operation (i.e., *isBlob*=1). The layout of the FEC chunk trailer is shown in Figure 18.

0		1	
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5		
+-----+-----+-----+-----+			
C Pkt ID Chk ID nChunks			
+-----+-----+-----+-----+			

Figure 18. FEC chunk trailer fields

A copy of the chunk is saved in an FEC construction cache, and a check is made against the current FEC settings to see if a sufficient number of chunks are available within the cache to perform the FEC encoding function (or alternately if a maximum hold time has expired, which forces the generation of a possibly reduced number of FEC packets to accommodate encoding of intermittent data sources, such as back-and-forth voice conversations). If the required number is available, the specified number of FEC packets are constructed and sent out the raw socket. The encoding construction cache is then flushed, and the group number is incremented (and rolled over as needed to fit within the group ID range allowed in the FEC control trailer.) The packet sequence number within the group is also reset to 0.

If the required number is not yet available, but the maximum hold timer has expired, the UDP Proxy adjusts the encoding parameters based on the number of original chunks in the cache. The adjusted ratio attempts to preserve the ratio of protected chunks to encoded chunks, rounding up the number of FEC chunks to the next largest integer.

The FEC encoding process is performed over the payloads of the cached chunks. To accommodate chunks of varying payload length, the encoding process generates FEC chunks with payload lengths equal to the longest payload length of all original chunks in the cache. Within the FEC computation, any chunks with payload lengths shorter than this maximum length are treated as if they really are this maximum length with the “missing” data being all zeros.

To assist in decoding operations, a four byte FEC repair trailer, followed by the prescribed four byte FEC control trailer are appended to the end of each repair chunk produced by the encoding process. The repair chunk consists of two eight bit fields indicating the FEC encoding rate (specifically, the number of original chunks and the number of FEC repair chunks for this group) and a 16 bit field containing an FEC value computed from the payload lengths of the original chunks. The additional 16-bit field is needed to reconstruct the payload length of any missing chunks in the same way that the FEC repair chunk contents are used to reconstruct the contents of any missing chunks – and is only needed to handle the general case where original chunks are allowed to vary in size. The FEC length is of course stored in Network Byte Order. The layout of the FEC repair trailer is shown in Figure 19.

0		1		2		3	
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1			
+-----+-----+-----+-----+-----+-----+-----+-----+							
base rate FEC rate			FEC Length				
+-----+-----+-----+-----+-----+-----+-----+-----+							

Figure 19. FEC repair trailer fields

Paralleling the control trailer appended to each original chunk, the FEC control trailer appended to each repair chunk carries 1) a 1-bit type field set to indicate that it is a repair chunk (vs an

original chunk); 2) a 1-bit in-order flag indicating whether the decoder should preserve packet ordering; 3) a 1-bit reserved field; 4) the a 5-bit slot ID indicating the position of the chunk within the array of repair chunks generated by FEC repair chunk construction; 5) an 8-bit field containing the FEC group number used to distinguish between encoded sets, and 6) the original 16 bit UDP destination port in Network Byte Order. Its layout is identical to that described in Figure 16.

To facilitate chunk processing by the decoder, the repair trailer is appended to the FEC repair chunk before the control trailer is appended. This ensures that the FEC control chunk is always the last four bytes in the packet regardless of whether it is an original chunk or a repair chunk, thereby simplifying the handling of any packets received by the decoder.

Note that since only repair chunks carry the encoding rate information, the actual encoding rate used for any given group can be delayed until sufficient original chunks are available to perform the encoding. This facilitates dynamically setting the FEC encoding rate by an outside control application in a way that is most responsive to the current network conditions. For the special case when the rate is dynamically reduced so that fewer chunks are required than are already available in the construction cache, the encoder will attempt to use an encoding rate that preserves the encoding rate ratio, similar to the approach used for adjusting the encoding rate in response to a maximum hold time expiration. For example, consider the situation when the UDP Proxy is set to use a (5,10) code and four chunks of the requisite five chunks are in the construction cache. Next, consider that before the fifth chunk becomes available, the encoding rate is dynamically modified to use a (3,9) code. When the fifth chunk becomes available, and the encoder discovers that too many packets are already available, the encoder will instead encode using a rate (5,15) code.

Also note that a variation of this approach would be to search across all construction caches potentially affected by a change in codes, to construct and immediately send any FEC packets that are now permissible by the change. In the above example, this would then result in the immediate enqueueing of FEC packets using a (4,12) code to the Admission Controller.

The actual calculation of FEC repair chunks uses one of three different algorithms, depending on the specified rate. Specifically, rate (1,N) codes simply use chunk repetition to generate (N-1) copies of the chunk. Alternately, rate (N, N+1) codes calculate the single repair chunk as an exclusive OR across the N original chunk payloads. Finally, all remaining cases are handled using the Vandermonde matrix FEC encoding technique created by Luigi Rizzo (luigi@iet.unipi.it).

4.3.1.4 Admission Control

The admission control submodule is responsible for transferring packets from the UDP Proxy to the packet queue maintained by the local backpressure forwarder. The rate at which packets from any given flow are transferred to the backpressure forwarder, if at all, is determined by the utility function and the current queue length in the backpressure forwarder for the given packet destination. Generally, the longer the queue, the higher the cost of sending packets and thus fewer packets will be sent into the network. If the queue is large enough, packets may not be admitted into the network.

4.3.1.4.1 Utility function Mapping

Each application flow is assigned an appropriate utility function, $U(r)$, by the Admission Planner based on the application type, as well as a set of parameters that control the shape of the utility function and define the corresponding relative flow priority. In general, the UDP Proxy primarily supports inelastic (latency sensitive, loss-tolerant) traffic, with elastic (latency-insensitive, loss-intolerant) traffic primarily handled by the TCP proxy.

4.3.1.4.2 Admissions of packets

The utility $U(r)$ gained from transferring a packet to the local backpressure forwarder (i.e., admitting the packet to the network) is a function of the rate at which packets are admitted into the network. The optimal rate for admitting packets is determined by minimizing the Lagrangian $U(r) - \lambda r$ which occurs when $U'(r) - \lambda = 0$. For backpressure forwarding, the cost of admitting a packet is proportional to the queue length i.e. $\lambda = Q/K$ for some normalizing constant K . Packets are not admitted into the network if the net utility is less than zero.

When a packet is admitted, a recheck timer is set for (b/r) seconds in the future for the next packet admission, where r is the current optimal rate at which the packet was admitted in units of bits per second, and b is the length of the packet in bits. Packets that are admitted are placed into the backpressure queue for the corresponding destination. Note that if the recheck timer is very small (as can be the case for high rate deliveries) it is both difficult for the operating system to generate accurate timing and therefore rate control, and overall performance can be degraded due to excessive call overhead. Hence if the calculated value of the recheck timer is below some threshold value t , then the admission controller will instead admit multiple packets, such that the total number of bits $\sum_{i=1,N} b_i$ across the set of N admitted packets divided by the computed rate is just greater than the threshold: i.e., $\sum_{i=1,N} b_i > t > \sum_{i=1,N-1} b_i$.

4.3.1.4.3 Hysteresis

If the backpressure queue lengths are such that the net utility is close to or just below zero, the admission controller can oscillate between sending and not sending. When the queue occupancy is low, it begins sending; however, this then increases the queue length enough that the net utility is less than zero and the flow stalls, which then allows the queue to begin to empty. Once the queue length goes below some threshold value, the admission controller decides to send again which make the queue go above the threshold and it stops admitting packets. This oscillation can result in the average admission rate being somewhere between zero and the minimum send rate. For applications with inelastic traffic such as VoIP or streaming video, there is no value in sending at (short-term) average rates less than some minimum rate. In such cases it is better to turn off the flow than to waste network resources by sending packets at an average rate that yields no utility.

We deal with this by monitoring the history of packet admissions. We define intervals (small periods of time) over which we find the average send rate and measure the average queue length. We use these averages to determine the average utility from sending and the average cost of sending, and therefore the average net utility over the interval. If the average net utility is less

than some small *delta* for *th* consecutive intervals, we then decide to turn off the flow by stopping admission of packets from that flow.

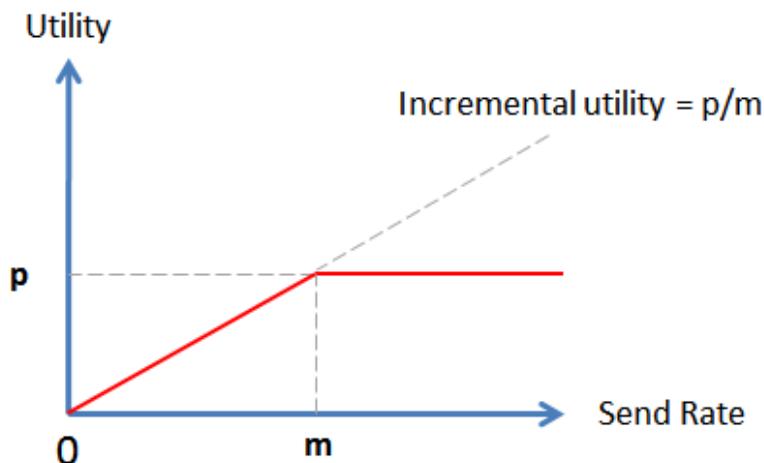
Once a flow is turned off at the admission control level, it can either stay off or be turned back on after some time. This would depend on the application. A VoIP conversation, for example, should stay off once it is terminated while a streaming video may still be useful if it comes back after some time. For applications that are tolerant of disruptions, we use a retry interval to schedule a time at which we attempt to turn back on the flow once it is turned off.

4.3.1.4.4 Trapezoidal Utility (TRAP)

Inelastic flows in general have the property that sending at less than some minimum rate m yields zero utility, and sending at rates greater than m yields no additional utility since inelastic applications in general will not generate traffic above that rate. This effectively defines a step function, where the utility is zero for all rates $0 \leq x < m$ and a constant utility p for rates $x \geq m$. For reasons that will be discussed shortly, we modify this step function model to have a trapezoidal form where the utility of admitting packets at rate x is defined as:

$$U(x) = \begin{cases} \frac{p}{m} & \text{if } x < m \\ p & \text{if } x \geq m \end{cases}$$

where p is the priority of the flow. This is illustrated in the figure below.



The incremental utility (i.e., the slope of the utility function) for rates $0 \leq x \leq m$ is given by $\Delta U = \frac{p}{m}$ and the incremental cost of admitting a packet is given by $c = \frac{Q}{k}$. The trapezoidal utility function admits packets whenever the incremental utility is greater than the incremental cost; i.e. whenever

$$\frac{p}{m} > \frac{Q}{k}$$

With a trapezoidal utility function, whenever the above relationship is satisfied it is satisfied for all rates x ; hence we have a choice in terms of what rate to choose whenever a packet is admitted. Since we want to maximize utility, a simple rule would be to choose $x=m$; since values less than m yield less than the maximum utility p , and values greater than m yield no additional utility. However, there are two downsides to this particular rule.

First, choosing $x=m$ immediately upon the start of the new flow induces a step function into the dynamical system which is GNAT. In general there will be other ongoing flows that will necessarily need to react to the new flow in order to establish a new equilibrium point, and in particular some of the other flows will themselves be inelastic and have trapezoidal utility functions.

When the system does not have sufficient capacity to accommodate all inelastic flows, there will be a period of time where the different trapezoidal flows are competing for the available capacity, yielding oscillatory behavior in the manner described in the hysteresis section above. Since these oscillations can and in general will temporarily disrupt any and all other ongoing flows, this is a situation we choose to avoid. Ideally any new flow will displace any ongoing lower priority flows without otherwise interrupting ongoing higher priority flows. We do this by changing the admission rate in smaller steps to affect a smoother transition between equilibrium points.

Second, not moving immediately to a rate of m implies that packets will be building up for that flow in the UDP proxy input buffer; thereby creating a persistent increase in end-to-end latency. We compensate for this by defining a maximum surge rate $b \geq m$ at which the trapezoidal utility can admit packets. This also better accommodates Variable Bit Rate (VBR) sources which may have averages above the nominal average rate for time periods longer than the nominal management intervals.

With these considerations in mind, the trapezoidal utility function behaves as follows:

First, divide the surge rate b into $N+1$ equally spaced rates between 0 and b

Next, assume that the trapezoidal utility has previously decided to transmit packets at some given rate x (initially $x=0$).

If the above inequality is true it increments the previous transmission rate by a fixed amount, else it decrements the previous rate by the same fixed amount. Hence the rate moves along a linear trajectory, mirroring the leading edge of the trapezoid. Of course, the minimum rate is 0, and the maximum rate is b .

More precisely, the trapezoidal utility function admits packets into the network at one of $N+1$ distinct rates $x = i \frac{b}{N} \geq i \frac{m}{N}$ where N is the number of steps and $0 \geq i \geq N$ is the step index.

Once a rate is set, the trapezoidal utility function continues to use that rate for a fixed period of time, called the *StepInterval*, at which point a new rate is recomputed. If the net incremental utility is positive, the trapezoidal advances to the next higher rate (effectively incrementing i); if the net incremental utility is negative it retracts to the next lower rate (effectively decrementing i).

We note here that the step utility function is a special case of the trapezoidal utility function, which can be obtained by setting $N=1$.

Hysteresis is incorporated by computing the average send rate over configurable time intervals. If the average send rate is less than $m\delta$ for th consecutive intervals (i.e. the average utility is 0), we turn the flow off for a longer period of time. The flow will attempt to restart after the off period. If the flow should remain off once it is turned off, a very long restart period can be selected

4.3.1.4.5 Simplified TRAPezoidal (STRAP) Utility Function

The TRAP utility function described above requires setting a raft of parameters that may be difficult or even impossible to determine. The goal for the Simplified TRAPezoidal (STRAP) utility function is developing a utility function and supporting processing algorithm that:

- Minimizes number of parameters required (no need for configured m, b , intervals etc)
- Maintains the gradual ramp up/down characteristics of the trapezoidal utility function
- Respects priorities (lower utility per bit triaged before higher utility per bit).

We define the STRAP utility function $U(x)$, as follows:

$$\begin{aligned} U(x) &= 0 \quad \text{if } x < s \\ U(x) &= p \quad \text{if } x \geq s(1 - \theta\delta) \end{aligned}$$

Here x is the current admission rate, s is the current rate at which traffic is being sourced, and δ is the loss threshold for the flow, and θ is the inertia of the flow.

Therefore, with the STRAP utility, a flow only gets utility if it delivers at least $(1 - \theta\delta)$ of the traffic being sourced, otherwise it gets 0 utility.

The TRAP utility function described above requires setting a raft of parameters that may be difficult or even impossible to determine. The goal for the Simplified TRAPezoidal (STRAP) utility function is developing a utility function and supporting processing algorithm that:

- Minimizes number of parameters required (no need for preconfigured m, b , intervals etc.)
- Maintains the gradual ramp up/down characteristics of the trapezoidal utility function
- Respects priorities (lower utility per bit triaged before higher utility per bit).
- Leads to stability (If there are identical flows and only a subset can fit, the set of flows that are on remains constant).
- Prioritization and Stability works in a distributed setting, without explicit coordination.

The table below compares TRAP and STRAP parameters:

Parameter	TRAP Utility Function	STRAP Utility Function
p (priority)	Configuration Required	Configuration Required
m (average source rate)	Configuration Required	Computed on the fly
b (maximum admission rate)	Configuration Required	Not Used
step interval	Configuration Required	Configurable, otherwise use default
averaging interval	Configuration Required	Configurable, otherwise use default
restart interval	Configuration Required	Configurable, otherwise use randomized default (~6s)
number of steps	Configuration Required	Configurable, otherwise use default (8 steps)
δ (maximum loss rate)	Configuration Required	Self-adapting, based on flow history

Priority

The priority of an inelastic flow indicates the relative priorities of flows. The decision to admit a flow or not is dependent on the queue size, the rate of the flow and the priority according to the equation below, which is identical for TRAP utility.

$$\frac{p}{m} > \frac{Q}{k}$$

Average source rate (m)

With the STRAP utility, the average source rate (m) is computed as a moving average and is as variable as the input source. This is more practical as it does not require a priori knowledge of the application rates and this approach better handles Variable Bit Rate (VBR) flows. If there are periods of low rate, the queue threshold for admission becomes higher as the incremental utility per bit is higher. The average source rate is used when deciding to step up/down. In the inequality above, it uses the source rate as the value of m , and steps up one step (to a maximum of step n (default 7)) if the inequality holds, otherwise it steps down one step (to a minimum 0). This makes the STRAP utility much less reliant on knowing the average or bounded rate for the flow and more adaptive to changes the traffic generated by the application.

Admission rate

The STRAP utility attempts to admit all packets received, every service interval, without relying on an explicit admission rate. Instead, it admits a fraction of the backlog based on the current step, according to the equation below.

$$\text{bytes_admitted} = \frac{\text{current_step}}{\text{total_number_of_steps}} * \text{current_backlog}$$

Inertia

Existing flows should be given preference over new flows of the same priority, if they both cannot fit. The notion on inertia achieve this by allowing ongoing flows to withstand more push-back than new flows by dynamically adjusting delta based on the duration of the flow and the stability of the queues in the system. When this is coupled with the ramp up of STRAP utility, the system is stable and inelastic flows do not thrash even when the system is oversubscribed with flows of identical priorities starting and/or restarting.

The inertia of a flow is initially set to 0. This means that any step-down (which means there are packets not being immediately admitted), will cause the flow to be triaged. Each averaging interval, if the queues to the destination of a flow are not increasing, then the inertia is additively increased. The increment is set to currently set to 0.05, and the maximum value for inertia is 1. When a flow has maximum inertia, then it is allowed to lose up to δ before it is triaged. Thus flows that have been active, while the network has been stable for longer periods of time will be more difficult to triage than flows that have just started up. Inertia dampens the system, and care

must be taken that too much inertia is not applied to a flow as this decreases the responsiveness of the system, though making it more stable.

Triage

Ideally, the STRAP utility should be operating on the highest step, which means all packets in the backlog should be admitted each cycle. If the STRAP utility is not on the highest step, then the backlog will start growing and packets will experience delays in the proxy queues. The current step is evaluated at fixed intervals and the current step can be increased (up to a maximum step), decreased (down to the 0th step) or remain the same. There are two factors that affect the current step in the STRAP utility:

- The current queue to the destination
- The current average loss rate experienced by the flow

If the current queue is such that $\frac{p}{m} > \frac{Q}{k}$ then the "cost" of admitting packets is less than the increased utility, so we can increase the step and send at a faster rate. If the loss rate (as reported by the destination in the RRM) is greater than δ then we will not be getting any utility from the flow and the rate should be decreased by decreasing the step. High loss rate can be caused by remote bottlenecks which can result in packets not being delivered in time. In these cases, the queue may not build up at the source to cause a step down, the loss rate triggers the step down as it represents a loss in utility.

The current step is used to update the average step every admission cycle. If the average step crosses the threshold (determined by the loss threshold and the inertia), according to the equation below, then the flow is triaged.

$$\text{average_step} = (1 - \delta\theta) * \text{total_number_of_steps}$$

To provide behavior similar to the TRAP utility, STRAP computes the following quantities:

- An instantaneous value of the admission rate is calculated each time packets are admitted using the formula: $\text{admission_rate} = \text{current_step total_step} \times \text{backlog_elapsed_time}$
 - A fraction of the backlog is sent during an interval.
 - This fraction is a function of the 'step' we are on.
 - The interval used in the calculation is the time since the last admission rate calculation where there were packets in the backlog.
 - E.g. if there are X bits of data in the backlog at the time we calculate the sent rate, and it has been t seconds since the admission rate was last calculated, then the admission rate should be X/t (if we are on the highest step). A burst of packets will then be admitted. After t seconds, the admission rate would have to be recomputed and the size of the backlog would again be used to calculate the admission rate.
- An average value of the ingress rate, m , is maintained and is reported to AMP.
 - The instantaneous value in admission rates can have large variations with VBR sources and this would pose a problem with supervisory control, resulting in flows being possibly turned ON and OFF frequently.

- The bits received from the m is summed over an $m_averaging_interval$ (set to 0.1 seconds).
- At the end of the $m_averaging_interval$ we calculate the average rate for that interval.
 - This averaging smooths out bursts from the applications.
- We maintain the EWMA of the average rate.
- There is a default number of steps (set to 8), though the value remains configurable.
 - Rules for stepping remains the same, step down if: $Q/k > p/m$, step up otherwise, where m is the average ingress rate.
 - Additive increase, multiplicative decrease to step up/down
 - step1 -> step2 -> step3 >...> step7 -> step8 -> step4 -> step2 -> step3 -> step1
 - For flows with a deadline, the interval for stepping is twice the deadline. This allows for fresh loss-rate information to propagate from the destination to the source.
- We measure the average backlog over intervals and triage a flow if the average backlog crosses a threshold.
 - This interval is similar to the averaging interval (used in the current TRAP utility function).
 - An interval is preferable to an instantaneous backlog threshold, as it avoids punishing short-lived bad behavior.
 - Each time the admission rate is calculated, the current penalty for the interval is updated in proportion to the size of the backlog that remained since the last admission rate calculation
 - If there is backlog rollover, then we must not be on the highest step, and this should be penalized.
 - Parallel to the current approach, there is no penalty when stepping up (treat the backlog as 0).
 - When a flow starts, it will start at the bottom step, and there will be a backlog for a few steps as we are only admitting a fraction of the packets in the backlog.
 - If the last step was a step down, the penalty is incremented using the following formula: $penalty = penalty + backlog_rollover \times elapsed_time$
 - The penalty is reset at the start of each averaging interval
- The threshold for triage, delta, has a default value (set to 0.9), but is configurable.
 - $threshold = avg_ingress_rate \times (1-\delta) \times interval$
 - Note: A flow is triaged if $penalty > threshold$
- Restart interval is a function of a base interval (has default value, configurable) and priority.
 - Higher priority flow restarts faster (which is better than having a lower priority flow restart only to have it triaged when a higher priority flow restarts soon after).
 - The following formula is used:
 - $base_restart_interval$ is set to 6 seconds.
 - As priority gets higher, the restart interval will approach half of the $base_restart_interval$
 - As priority get smaller, the restart interval will approach the $base_restart_interval$
 - The base restart interval is optionally configurable
 - The interval for stepping is set to 0.01 seconds, this is optionally configurable.

An interesting property is that the threshold for stepping down changes. If the source is sending at a lower rate than the average for a little while, then the utility per bit (p/m) goes up faster than if we're using an average and we are less likely to triage the flow in this case.

4.3.1.4.6 Logarithmic Utility

For debugging and simplified system testing, the UDP proxy also supports the use of logarithmic utility functions. Logarithmic utility functions more naturally support elastic application flows, and are described in the TCP proxy discussion.

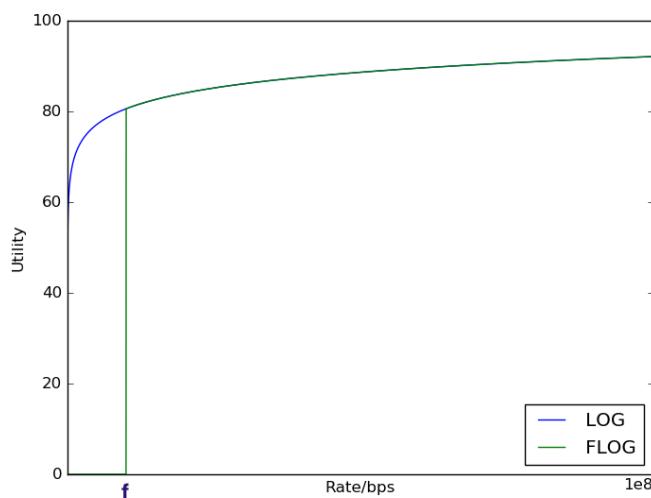
4.3.1.4.7 Floored Logarithmic Utility

The floored Log Utility (FLOG) was created to capture the utility for adaptive flows with some minimum requirement(s), e.g. RDP or Dynamic Adaptive Streaming over HTTP (DASH). The FLOG utility function is identical to the LOG utility as long as the rate is above some minimum threshold f , otherwise it is zero. DASH video can adapt down to a rate needed to support the minimum frame rate and it will play reasonably well. However, if the available rate is less than this, there will be a lot pauses for buffering while the stream catches up and this is not a good operating point for most users.

More formally, the utility of a flow with FLOG utility or priority p , operating at rate r is given by:

$$U(r, p) = \begin{cases} p \log(r + 1) & \text{if } r > f \\ 0 & \text{otherwise} \end{cases}$$

This is shown in the plot below which compares the LOG and the FLOG utility, for equal priority:



The value of f is important and we implement a few methods for setting or dynamically determining the minimum rate (floor) for an adaptive flow.

4.3.1.4.7.1 Configure static minimum rate

The simplest approach is allowing the user to specify a minimum rate, f , in the utility function definition. We monitor the average admission rates of small time intervals. If this rate is less than f for n consecutive intervals, the flow is triaged. This requires the user knowing about the rate requirements of the flow or using some generally accepted low value.

4.3.1.4.7.2 Dynamic floor estimation

Another simple approach is to monitor the backlog to determine if the flow is properly being serviced. We must be careful as GNAT will adapt and these flows will also adapt and we don't want to triage the flow before it, and the system, is done adapting. We monitor the queuing delay of packets waiting to be admitted (i.e. in the backlog). If this delay is larger than some configured threshold, then there will be at least this much end-to-end delay and we consider this to be an indication that the flow is not currently being properly serviced. However, if

- The admission rate is increasing (the system is adapting in favor of the application) or
- The backlog is decreasing (the application is adapting to the network)

we do not accrue penalty. If the backlog latency is greater than the threshold and neither of the above is true for n consecutive intervals, the flow is triaged.

4.3.1.5 Handing Admitted Packets Off to the Local Backpressure Forwarder

Once the admission controller decides to admit a packet into the backpressure forwarded network, It notifies the BPF with a pointer to the packet in shared memory. At this point, the BPF now "owns" the packet; however, the UDP proxy maintains a "soft" copy of the packet until FEC generation is complete. Whichever of the two processes is last to "delete" the packet will actually remove the packet from shared memory.

Coordination of packet deletion is handled by the common approach of maintaining a reference counter, which is incremented whenever a "copy" is made, and decremented whenever the "copy" is deleted. When the reference counter reaches zero, the packet can be safely removed.

4.3.2 Receiver Side Processing

4.3.2.1 Hand off of Packets from the Local Backpressure Forwarder

As described in the backpressure forwarder detailed design, upon receipt of a packet from the Wide Area Network (WAN), the backpressure forwarder makes a decision as to whether the packet is to be forwarded to another backpressure forwarder, consumed (if it is a LowQueueMsg or LatencyReportMsg), or delivered locally. If the packet is to be delivered locally, and it is a UDP packet, the packet is then passed to the UDP proxy for decoding, reordering and delivery. The actual transfer of packets to the local backpressure forwarder is (currently) performed using

Inter Process Communications, leveraging the IPC class described in the common class documentation.

The UDP proxy maintains a DecodingState object for each active flow. The mapping of packet to DecodingState is based on the 4-tuple of IP addresses and ports. Within each DecodingState is collection of FEC groups which are not yet expired (as described below). The mapping of packets to FEC group is based on the groupID of the packet, which is stored in the FEC trailer.

4.3.2.2 FEC Decoding Process

The FEC decoding process begins when a chunk contained in a UDP packet is received from the backpressure forwarder. If the trailer type indicates that it is an original chunk (vs. a repair chunk), it is inserted into a reconstruction cache as an original chunk.

A check is then made to determine if sufficient chunks are available to reassemble one or more of the original packets (the actual number being determined by the *isBlob* field in the chunk header). If so, the original packet(s) *may* be reassembled and forwarded out the VIF at this time. Whether or not packets are forwarded depends on whether there are any packets with lower sequence numbers in this FEC group that have not yet been reassembled and transmitted, and whether the specific service requires in-order packet delivery. If there are no such missing packets, or the service is known to not penalize out-of-order delivery, the packet(s) is (are) immediately forwarded. Otherwise the packet(s) is (are) held up awaiting the remainder of the FEC reconstruction process.

If the trailer type instead indicates that the chunk is a repair chunk, the control trailer is removed, the length is adjusted, and the chunk is inserted into a reconstruction cache as a repair chunk.

Once either the repair or original chunk has been inserted into the cache, a check is made to see if sufficient chunks have been received to enable reconstruction of any missing chunks. If so, the missing original chunks are regenerated, and all as yet untransmitted packets are reassembled from their constituent chunks and sent along with any previously unreleased packets in the proper order. All chunks in the reconstruction caches are then discarded, freeing up local memory.

The proxy keeps track of the expiration time of each packet. This expiration time is the minimum of the deadline specified in the service definition, the configured maximum hold time in the packet and the expiration time of the sequentially next packet that has been received. This definition ensures that packets, and FEC group expire sequentially in order. The proxy keeps track of the next group to be sent that has not yet expired. If reordering is required, packets are released in order until there is a missing packet and will advance again either when the missing packet comes in, or the next packet expires. If all the packets in a FEC group has been released and/or the group has expired, the FEC group is destroyed and the memory is freed. If packet come in late for a group that has already expired, the packet can be released if it is an original packet, otherwise it is destroyed.

Given the current set of program performance metrics where no credit is given for late arriving or otherwise out-of-order packets, the ability to preserve packet ordering in the manner described affords most of the gain achievable with an FEC based approach.

4.3.2.3 Packet Reordering

The UDP proxy can reorder packets, if configured to do so. FEC packets are aggregated into FecState objects with one such object per FEC group. We use the FEC GroupID to determine the correct sequence in which to release FEC groups and packetIDs to determine the sequence in which to release packets within a FEC group. We keep track of the current groupID that needs to be sent and use this to identify late packets. This groupID starts from a random integer and can wrap around.

When a packet arrives at the proxy from the BPF it is given an expiration time of $\min(\text{MaxHoldTime}, \text{TTG}, \text{NextPacketExpiration})$. This results in packets expiring in order based on groupIDs and packet IDs. To turn off reordering, set the MaxHoldTime in the UDP proxy to 0. We check for expired packets in a decoding state whenever a packet arrives for that decoding state and release any such packets in the correct order. When we receive a packet for the FEC group that is currently being released, we check if it is the next packet in this group that must be released and if so, we release this packet and any subsequent packets until there is a gap. There is a single timer that is set, and updated, to reflect the next time a packet will expire across all FEC groups. This handles the case of long inter arrival times and the termination of a flow.

Packets in a FEC group are released when any of the following conditions are met:

- The packet is the first packet in FEC group X and FEC group X-1 has been sent.
- The packet is the n^{th} packet and the $(n - 1)^{\text{th}}$ packet has already been sent in this FEC group.
- The packet is an original packet that belongs to a group that has already been sent.
- The packet has been in the UDP proxy for MaxHoldTime seconds.
- A subsequent packet (that has arrived at the proxy) has been in the UDP proxy for MaxHoldTime.
- The time-to-go on the packets reaches 0

We use the following series of examples to illustrate the behavior. We use the following notation – X:Y – to indicate that the packet has a FEC groupID of X and a packet ID of Y. Assume we are using a 6/8 code where X:6 and X:7 are the repair packets.

1. Packets arrive in the following order: 5:0, 5:1, 5:2, 5:3, 5:4, 5:5, 5:6, 5:7, 6:0, 6:1. We will release 5:0-5:5 in the order in which they arrive, discard 5:6 and 5:7, and release 6:0, 6:1
2. Packets arrive in the following order: 5:0, 5:1, 5:2, 5:3, 5:4, 5:5, 6:0, 6:1. We will release 5:0-5:5 in the order in which they arrive, and then wait until 6:0 expired. The FEC rate is only stored in repair packets, and we do not know the FEC rate until we receive one of these. In this example, we will not know that 5:5 is the last packet in this FEC group so we will wait for 5:6 until we either receive a repair packet or packet 6:0 expires.

3. Packets arrive in the following order: 5:1, 5:3, 5:2, 6:0, 5:0 with long expiration times. Packets 5:1 - 6:0 will be held in their respective FEC groups until 5:0 arrives then we release in order – 5:0, 5:1, 5:2, 5:3 – and wait until 5:4 comes in
4. Packets arrive in the following order: 5:1, 5:3, 5:2, 6:0, 5:0, 5:5, 5:6, 5:7 with long expiration times. We will release packets 5:0-5:3 after 5:0 comes in, as above and then hold until 5:6 comes in. At this point we can reconstruct 5:4 and send packets 5:4-5:6 and 6:0. We will delete FEC group 5 at this point and discard 5:7 when it comes in.
5. Packets arrive in the following order: 5:1, 5:3, 5:2, 6:0, 5:0 with MaxHoldTIme << TTG in each packet. The expiration time of 5:1 and 5:3 will be the time they were received plus MaxHoldTime. The expiration time for 5:2 will be set to the expiration time of 5:3, since this will be earlier than the receive time plus the MaxHoldTime.
6. Packets arrive in the following order: 5:1, 5:3, 5:2, 6:3, 5:5 5:4, 5:6 with MaxHoldTIme << TTG in each packet. The expiration time of 5:5 will be set to the expiration time of 6:3 for the same reason as above. If 6:3 expired before 5:4 arrives, then 5:4 will be considered a late packet and will be sent immediately. 5:6 will also be considered late, but it is a repair packet so it will be discarded.

Limitations:

- If the source is generating traffic at a greater rate than the UDP proxy is admitting packets for the flow, the backlog buffer in the UDP proxy may eventually fill. At this point, we will be dropping packets that cannot fit in the buffer, after these packets have been FEC encoded. This results in gaps in the FEC group that are admitted and only parts of a FEC group may actually be admitted. The gaps will cause packets to be held in the reorder buffer for min(MaxHoldTime, TTG) since the next expected group will never arrive since it was dropped.

4.3.2.4 Sending Decoded and Reordered Packets Along to Applications

Initially the UDP proxy used a virtual interface device (i.e., a Linux tun device) to capture packets from, and release packets to, application hosts. Use of tun devices also required disabling reverse path filtering in order to avoid having the Linux kernel discard packets appearing to come from an interface that was inconsistent with the routing tables for a given packet source address. As we now use raw sockets instead, this is no longer an issue.

4.3.2.5 Loss reporting at the destination and processing at the source

The destination UDP Proxy monitors loss rates for each flow and notifies the source proxy periodically of the current average loss rate and will also proactively generates a report if the loss rate for a flow exceeds a configured threshold (delta from the STRAP utility function). We declare a packet as lost if we are unable to reconstruct that packet before the deadline for reconstructing a packet with a higher sequence number. A packet must be released to the application when its TTG is zero, and this is use as the deadline. Consequently, we can only measure loss in flows where there is a latency constraint and reordering. For e.g. If a source receives 6 packets from the application: A, B, C, D, E and F, and the destination reconstructs and releases A, B, C and then receives some more packets and is able to reconstruct D and F, but not E, it will release D as soon as it is constructed but will hold on to F until either it can reconstruct and release E or until F's TTG reaches 0. If the TTG of F reaches 0 and the destination is unable to reconstruct E,

then E is declared loss and F is released to the application. Even if E arrives later, it is still declared as lost since it would have a negative TTG.

To allow for the computation of loss rates, the source UDP proxy includes a sequence number and the total bytes sent thus far in each original packet in the FEC trailer of each data packet. The destination UDP proxy keeps track of how many original packets were reconstructed and how many bytes were released to the application. The destination proxy can determine how many packets and bytes are missing and therefore compute a loss rate. The destination proxy keeps an Exponentially Weighted Moving Average (EWMA) of the loss rate, in terms of bytes, and triggers a Release Record Message (RRM) if the current average loss rate exceeds a threshold. These RRMs are triggered immediately when the threshold is crossed and periodically from then on as long as the loss rate remains above the threshold. This is done on a per-flow basis. A proxy can send multiple RRMs in a short time if they are for different flows. RRMs use the control packet queue in the BPF and takes priority over data traffic. When an RRM is received at the source proxy, it updates state for the flows specified in the RRM.

4.3.3 Internal State Management

The UDP Proxy maintains separate encoding state for each outbound flow, as defined by the 4-tuple of source address, destination address, source port, and destination port. The encoding state for a given 4-tuple consists of the current FEC encoding rate, the number of original packets seen so far matching this context, the packets themselves, and the current group ID. (Note: the sequence ID used for encoding an original packet is identically the number of original packets seen so far.) The encoding state also includes the admission control state for the flow, including the choice of utility functions, utility function parameters, and the encoded packet queues managed by the admission controller.

The UDP Proxy also maintains separate decoding state for each inbound flow, again as defined by the 4-tuple of source address, destination address, source port, and destination port. The decoding state for a given 4-tuple consists of the number of FEC encoded chunks (both original and repair) received so far, the respective sequence numbers of those chunks, and the chunks themselves. The decoding state also retains information about whether an original chunk has or has not been used in the reassembly and forwarding of an original packet, as needed to support in-order delivery requirements.

In addition, the UDP Proxy maintains a set of default encoder and admission control settings for each service type, as defined by a service port range. As per the *iptables* example in the first section where a service port range of 5060 to 5061 inclusively was used, either the source or the destination port within a UDP packet may actually be the service port for a given session. This will depend on the specific application, as well as the direction of information flow: typically information flowing to a server will have the service port as the destination port, and information flowing from a server will have the service port as source port within the packet.

The encoder settings for a given service type include the service port range, a flag specifying whether or not packet ordering should be preserved, a pair of values specifying the FEC coding rate, and a timeout value specifying the length of time an inactive encoding or decoding state for

4-tuples of this service type will be kept around (after which they are eligible for automatic removal).

Encoding and decoding state objects will automatically be created upon first receipt of an original packet by the encoder, or upon first receipt of an FEC encoded original or repair packet by the decoder. Both encoding and decoding state objects are maintained using a std::map of four tuples.

4.3.4 Interprocess Communications

Exchanges of information between GNAT components use a variety of interprocess communication methods. Transfers of packets (including application, FEC repair, and latency response message packets), and queue depth information and between the UDP proxy and the Backpressure Forwarder leverages shared memory constructs as described in the [Shared Memory IPC Section](#). Notification of packet and queue depth availability is performed using bi-directional named pipes, also as described in the Shared Memory IPC Section.

Transfer of utility function definitions and performance statistics between the UDP proxy and the Admission Planner is achieved via TCP-based network connection implemented using the RemoteControl class.

4.3.5 UDP Proxy Configuration

The UDP Proxy uses a property table-based approach for configuring various run-time parameters controlling its operation. Parameters can be set using command line arguments, from a configuration file, or both. Settings contained within the file have precedence over command line settings for the same parameter. If a parameter is not set, either via the command line or in the configuration file, a default value is used. A full list and explanation of config parameters can be found here: <https://waban.bbn.com/confluence/display/GNAT/GNAT+Configuration+Guide#GNATConfigurationGuide-udpproxyconfig>

The UDP Proxy executable *udpProxy* is told to use the configuration file *udpProxy.cfg* using the following *-c* command line switch as follows:

```
udp_proxy -c udp_proxy.cfg
```

Note that lines beginning with the character # within the configuration file are treated as comment fields and are ignored.

Finally, note that the command line switches can be described using the *help* command switch *-h* as follows:

```
udp_proxy -h
```

5 TCP Proxy

5.1 Summary of Module Goals and Objectives

The three main goals in creating a TCP Proxy are: 1) providing an Admission Control capability for TCP-based applications to interface with the backpressure forwarded network, 2) realizing a high-performance, reliable and adaptive transport capability over multipath routed networks with potentially highly varying latency and loss, and 3) providing these capabilities without the applications having knowledge that the gateway is operating on their behalf or in fact, is even present in the packet delivery path.

Topologically, the TCP Proxy sits between two communicating hosts as shown on the top of Figure 20 below. From the perspective of applications, however, GNAT nodes appear only as intermediate routers in that no modifications to packet content or application processing are required. This application transparency is depicted on the bottom of Figure 20.

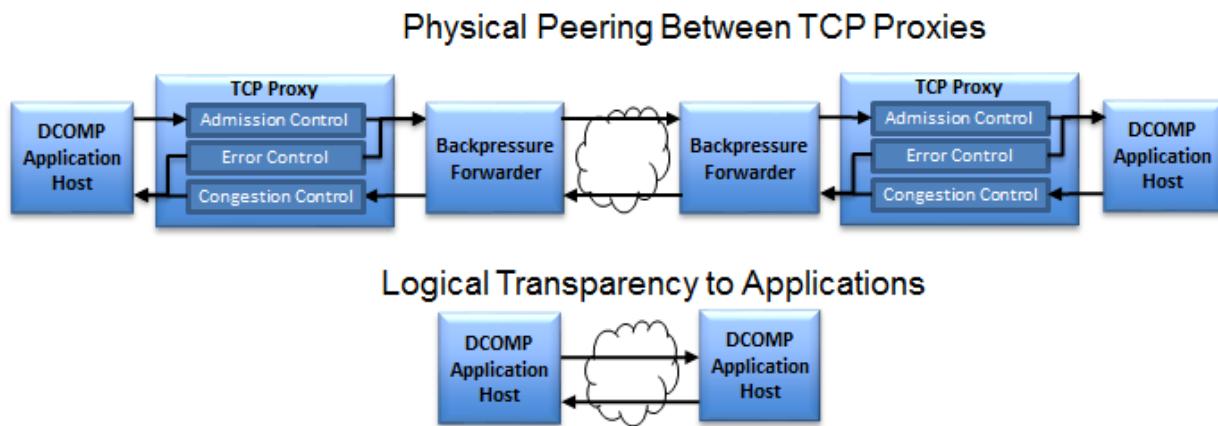


Figure 20: Transparent TCP Proxy relationship to network topology interconnecting two applications hosts (top) and as seen by communicating applications on those hosts (bottom)

Our design draws from our work extending and enhancing the Space Communications Protocol Standard (SCPS) TCP proxy first developed in the mid 1990's. Our implementation mirrors SCPS' 'TCP termination' mechanisms including per-application buffering, upstream flow control to sending applications, and standard New Reno congestion control on the LAN-facing side (i.e., between proxies and application hosts). On the WAN-facing side GNAT's TCP proxy leverages Selective ACK (SACK) signaling for efficient error control/retransmission, and incorporates backpressure-appropriate admission control algorithms in lieu of congestion control or congestion avoidance algorithms.

5.2 High Level Design

5.2.1 Adjacent Modules and Components

As shown in Figure 21 below, the TCP Proxy receives data packets from the TCP-based Applications and proxied data packets (from proxies in remote enclaves) from the local Backpressure Forwarder. It receives control information from: 1) the Admission Planner in the form of utility functions and parameter settings; 2) the local Backpressure Forwarder in the form of requests for packets (as Backpressure Forwarder Packet Queues empty); and 3) from proxies in remote enclaves typically embedded within the proxied packet headers (e.g., as ACKs and/or SACKs (selective acknowledgement lists)) and connection state information such as SYN/FIN/RST typical of TCP operations).

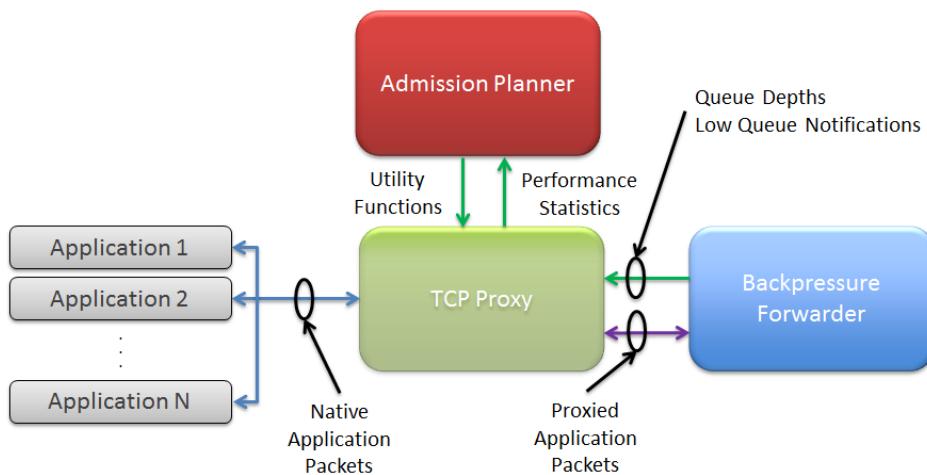


Figure 21. Relationships between the TCP Proxy, external applications, and supporting GNAT components

5.2.2 Input Sources and Information Needed

5.2.2.1 Control Plane Inputs

The Table below identifies the Control Plane inputs and their sources for the TCP Proxy.

Input Source	Input	Description
Backpressure Forwarder	Queue Depths	The depths of the Backpressure Forwarder Queues.
Admission Planner	Flow Utility Function	The Utility Function associated with a TCP Flow. This will be utilized to optimize the movement of packets to the Backpressure Forwarder Packet Queues.

5.2.2.2 Data Plane Inputs

The Table below identifies the Data Plane inputs and their sources for the TCP Proxy.

Input Source	Input	Description
Applications	Application Packets	The TCP packets that are received from the local Applications that are destined for remote Applications.
Backpressure	Proxied Packets	The packets that are received from the peer TCP Proxy that are

Input Source	Input	Description
Forwarder		destined for the local Applications.

5.2.3 Output Destinations and Information Provided

5.2.3.1 Control Plane Outputs

The Table below identifies the Control Plane outputs and their destinations for the TCP Proxy.

Output Destination	Output	Description
Admission Planner	Flow Status	Progress indicator for each flow, including completion state and achieved utility.

5.2.3.2 Data Plane Outputs

Output Destination	Output	Description
Applications	Application Packets	The packets that are received from the peer TCP Proxy that are to be delivered to the local Applications.
Backpressure Forwarder	Proxied Packets	The TCP packets that are received from the local Applications that are destined for the peer TCP Proxy.

5.2.4 Events

TCP Proxy Startup - When first launched, the TCP Proxy reads a local file to retrieve any configuration information. It then retrieves any command line arguments that, if present, override corresponding configuration information from the file. Finally it loads the set of configured parameters into its configured state, overriding any default values, including key parameters such as Maximum Segment Size (MSS), error signaling (e.g., SACK), retransmission settings (e.g., RTO estimation parameters), and so forth. Finally it configures the Linux forwarding engine to begin redirecting all received TCP packets to the TCP Proxy.

Establishing a New TCP flow - The establishment of a new TCP flow in the proxy is triggered by the receipt of a TCP SYN packet. The SYN packet may originate from a local application, indicating that the connection is being initiated locally, or it can be received from the Backpressure Forwarder, indicating that a peer TCP Proxy is originating the connection. The end-to-end MSS utilized for the flow is negotiated during the connection establishment handshake.

Assigning a Utility Function to an Active TCP Flow - Each TCP flow is assigned a Utility Function that controls the admission of packets into the backpressure forwarder packet queues. In general, the Admission Planner is responsible for assigning the utility function for each flow, however, given that the TCP Proxy will setup new flow state upon the arrival of a SYN packet, it is quite conceivable that the TCP Proxy will need to begin managing a new TCP flow before the Admission Planner is able to respond. Hence the TCP Proxy will initially assign a default utility function to the flow based on the configured settings, with the utility function subsequently updated by the Admission Planner.

Servicing an Active TCP Flow - Once a TCP flow has been established, the TCP Proxy must process the packets that are associated with the flow. A TCP flow is supported by two interfaces

that have a peer relationship with one another. One of these interfaces, the LAN Interface, is responsible for terminating the TCP flow with the local application and the other interface, the Overlay Network Interface, is responsible for reliably delivering the Application packets to the peer TCP Proxy servicing the destination Application over the overlay network. Figure 22 below illustrates the movement of Application packets through the TCP Proxy.

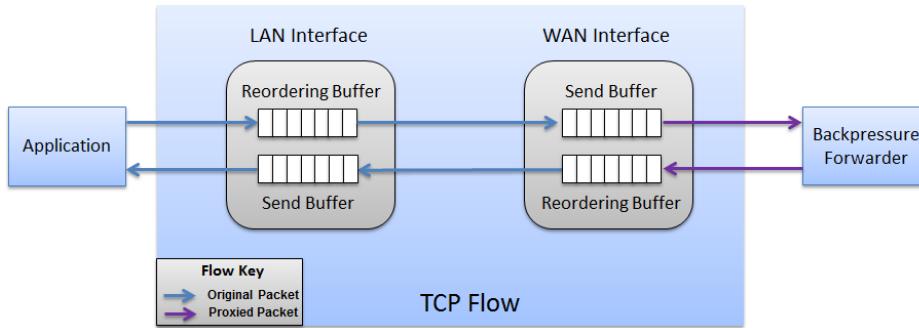


Figure 22. Movement of a TCP Flow's packets through the TCP Proxy.

TCP packets generated by local applications entering the GNAT node are redirected to the proxy using iptables rules (see below). Once a packet is received by the proxy, the flow state associated with the 4-tuple, consisting of source address, source port, destination address, and destination port, is located. If the received packet is in order and the flow state has room in its Send Buffer, the packet is placed into the Send Buffer and an appropriate TCP Ack is generated and sent back to the Application. If the received packet is out of order, the packet is placed into the Reordering Buffer and an appropriate TCP Ack is generated and sent back to the Application. Once the Send Buffer is full, a TCP window length of 0 is sent to the Application, indicating that no additional packets should be transmitted by the Application at this time.

Packets buffered in the LAN Interface Reordering Buffer are moved to the Send Buffer as "holes" in the TCP sequence number stream are eliminated. The Admission Control portion of the proxy is responsible for sending the packets in the Send Buffer to the Backpressure Forwarder, via shared memory, where the backpressure forwarding packet queues are located. The movement of packets from the Send Buffer to the backpressure queues is controlled by the Utility Function associated with the TCP Flow. Packets are not removed from the Send Buffer until their receipt has been acknowledged by the TCP Proxy at the destination.

As proxied packets are received by the TCP Proxy from the Backpressure Forwarder, via shared memory, they are placed in a Reordering Buffer if they are out of order or they are placed into the Release Buffer if they are in order and an appropriate TCP Ack is generated and sent to the source TCP Proxy. If the Release Buffer is full, a TCP window length of 0 is sent to the source TCP proxy.

Once the Overlay Network Interface Reordering Buffer accumulates packets that are free of "holes" in the TCP sequence number stream, the packets are moved to the Release Buffer for delivery to the Application. They are removed from the Release Buffer when an acknowledgement is received from the Application indicating they have been successfully received.

The TCP Proxy's zero-copy architecture moves packets received on one interface directly to the peer interface without repackaging: packet payloads are not split across multiple packets or merged with payloads from other packets. Only packet headers are updated to include the appropriate TCP ACK number, TCP options, and TCP window size. Eliminating deep copies of the packets as they move from one interface to the other increases the TCP Proxy's efficiency. Additionally, this behavior preserves the TCP port numbers from the source Application to the destination Application, enabling the proxy to provide transparent behavior from the Applications point of view.

Terminating an Active TCP Flow - The termination of a TCP flow occurs when either: 1) the Application closes the TCP connection or 2) the Admission Planner determines that there is no utility in further supporting the flow.

If the proxy receives a FIN packet from the Application, the proxy terminates the local TCP connection to the Application by adhering to the TCP connection termination protocol and sends a FIN packet to the peer proxy. If the proxy receives a FIN packet from its peer proxy, the proxy terminates the connection to its peer by adhering to the TCP connection termination protocol and sends a FIN packet to the destination Application.

A TCP flow may also be terminated following direction from the Admission Planner that supporting the flow no longer provides sufficient utility. This can be accomplished by transmitting a TCP reset (RST) to the Application and to the peer proxy. If a proxy receives a TCP Reset from its peer, it terminates the TCP connection to the peer and transmits a reset to the destination Application.

5.2.5 Performance Monitoring Statistics

Statistics provided to the User Interface include:

Summary statistics: Number of currently active application flows; cumulative average utility achieved; cumulative aggregate utility achieved

Connection statistics (indexed by four-tuple): utility function priority; cumulative number of proxied data packets sent; cumulative number of proxied data bytes sent; recent send rate in bits per second; recent send rate in packets per second; cumulative number of proxied data packets received; cumulative number of proxied data bytes received; recent receive rate in bits per second; recent receive rate in packets per second; recent average utility achieved.

5.3 Detailed Design

5.3.1 TCP Proxy Overview

Our TCP Proxy design draws from our work for AFRL extending and enhancing the Space Communications Protocol Standard (SCPS) TCP proxy first developed in the mid 1990's. Although our initial work developing a TCP proxy for GNAT leveraged a significant fraction of our enhanced SCPS code base, after a long series of simplifications and code optimizations, little remains of the original code.

5.3.2 Getting Application Packets Into the TCP Proxy

Transparent intercept of application packets by the TCP Proxy is accomplished using a raw socket configured with Berkeley Packet Filters to intercept those packet types which can be handled by the TCP Proxy, along with associated *iptables* rules configured to drop the intercepted packets. The use of *iptables* is necessary since packets obtained via the raw socket interface are in fact copies of packets, and without the drop rules the packets will also be delivered to the kernel.

The following packet selection logic is embodied in the Berkeley Packet Filter used by the TCP proxy. For the purpose of this discussion, assume that the local interface IP Address is 10.1.3.1.

- Any TCP packets that are not being sent to the local interface and not marked with a TOS value of 4 (which we use to bypass IRON/GNAT for demos) and not included in user-identified bypass list or VXLAN packets carrying TCP traffic (UDP packets with a destination port of 8472, encapsulating TCP packets)

(tcp and ip[1] != 0x4 and not dst 10.1.3.1 %s) or (udp dst port 8472 and udp[39]==6)

The above Berkeley Packet Filter specification includes a %s item, which represents a user-configured bypass filter string (described in more detail in Section 4.3.2.1 below) that contains the information for the user-specified 5-tuples representing the packets for which the user does not want the TCP Proxy to process.

The corresponding *iptables* rules for dropping these packets (on a host with a LAN-facing eno2 interface and IP Address of 10.1.3.1) are given by:

```
iptables -A PREROUTING -t mangle -i eno2 -p tcp ! -d 10.1.3.1 -j DROP
```

```
iptables -A PREROUTING -t mangle -i eno2 -p udp --dport 8472 -m u32 --u32 "56 & 0xFF = 0x6" -j DROP
```

5.3.2.1 TCP Proxy Bypass Filter Specification

As described above, all packets with a TOS value of 4 bypass the TCP Proxy. For applications and application hosts that do not easily support setting TOS values for transmitted packets, the TCP Proxy bypass filters provide for an additional mechanism for user-defined fine-tuning of the

TCP Proxy's raw socket Berkeley Packet Filter to control the packets that are received and processed by the proxy. The bypass filters are represented as 5-tuples that have the following format:

protocol;saddr:[sport | sport_low,sport_high]->daddr:[dport | dport_low,dport_high]

where

- **protocol** identifies the protocol of the packet and **must** be either `tcp` or `udp`.
- **saddr** identifies source IP Address. A value of `*` may be used here to refer to any source address.
- **sport** identifies source port. A value of `*` may be used here to refer to any source port.
- **sport_low,sport_high** identifies a range of source ports.
- **daddr** identifies the destination IP Address. A value of `*` may be used here to refer to any destination address.
- **dport** identifies the destination port. A value of `*` may be used here to refer to any destination port.
- **port_low,dport_high** identifies a range of destination ports.

As an example, consider the desire to bypass all TCP packets with destination ports in the range 30900-30909. The following user-configured bypass specification achieves this:

`tcp;*:*->*:30900,30909`

At run-time, the above bypass specification is converted into the following Berkeley Packet Filter string

and not (dst portrange 30900-30909)

which is then substituted for the `%s` item in the TCP Proxy Berkeley Packet Filter described above.

5.3.3 Error Control

5.3.3.1 Error Control Signaling Between Proxies

Reliability within TCP is implemented by its retransmission algorithm. Standard TCP New Reno detects and retransmits lost packets using an algorithm based on cumulative Acknowledgement that is simple and requires very small amounts of control information (4 Bytes in the TCP header). This algorithm is perfectly acceptable for LAN communications (e.g., between an application and the local proxy) where network speeds are very high and loss probabilities are low. However, its performance degrades significantly in WAN environments even with small amounts of loss; simple cumulative acknowledgement schemes can be inefficient, and signaling delays over longer haul WANs further exacerbate the problem.

To improve signaling efficiency and reduce signaling delay, the GNAT TCP proxy uses the Selective ACKnowledgement (SACK) retransmission protocol described in IETF RFC 2018.

5.3.3.2 Error Control Parameter Estimation

A key issue for Error Control within a multipath-forwarded environment is estimating how long to wait before deciding that a packet has actually been lost (vs. just substantially delayed) in order to initiate retransmission, i.e., what should the retransmit timer be set to in order to minimize delay in error recovery without generating substantial numbers of duplicate packets at the receiver?

Currently the TCP proxy tracks the worst Round Trip Time (RTT) observed between proxies and uses that value as the timer for retransmitting any packets that have been reported as holes by the receiving TCP Proxy. Conceptually, waiting longer than is minimally necessary to retransmit packets does not affect network efficiency, and only delays the delivery of packets to the end application. The potential down side for waiting longer than is minimally necessary is that the sending proxy may be "zero windowed" by the receiving proxy when the receiving proxy's reordering buffer becomes full. In this situation, the sending proxy may only send packets that are reported as missing by the receiving proxy, and that the network may be significantly under driven particularly when there are only a small number of flows. Once the missing packets are reported as being received by the receiving proxy and the receive window is reopened, the sending proxy can resume sending new packets. The effects of under driving the network due to receiver window limitations can be ameliorated by increasing the size of the receiver's reordering buffer.

5.3.4 Admission Control

The admission control submodule is responsible for moving packets from the Overlay Network Interface Send Buffer to the respective Backpressure Forwarding Packet Queue.

5.3.4.1 Utility function Mapping

Each application will be assigned a utility function, $U(r)$ by the Admission Planner. These input utility functions must then be mapped to "standard" utility functions (e.g. logarithmic, see below) with a set of parameters such that the mapped utility function will be a close approximation of the input utility function.

5.3.4.2 Admissions of packets

The utility $U(r)$ gained from transferring a packet to the local backpressure forwarder (i.e., admitting the packet to the network) is a function of the rate at which packets are admitted into the network. The optimal rate for admitting packets is determined by minimizing the Lagrangian $U(r) - \lambda r$ which occurs when $U'(r) - \lambda = 0$. For backpressure forwarding, the cost of admitting a packet is proportional to the queue length Q i.e. $\lambda = Q/K$ for some normalizing constant K . Packets are not admitted into the network if the net utility is less than zero.

When a packet is admitted, a recheck timer is set for (b/r) seconds in the future for the next packet admission, where r is the current optimal rate at which the packet was admitted in units of bits per second, and b is the length of the packet in bits. Packets that are admitted are placed into the backpressure queue for the corresponding destination. Note that if the recheck timer is very

small (as can be the case for high rate deliveries) it is both difficult for the operating system to generate accurate timing and therefore rate control, and overall performance can be degraded due to excessive call overhead. Hence if the calculated value of the recheck timer is below some threshold value t , then the admission controller will instead admit multiple packets, such that the total number of bits $\sum_{i=1,N} b_i$ across the set of N admitted packets divided by the computed rate is just greater than the threshold: i.e., $\sum_{i=1,N} b_i > t > \sum_{i=1,N-1} b_i$.

5.3.4.3 Elastic Traffic: Logarithmic Utility

We use a logarithmic utility function as the basis for computing the utility for elastic traffic, given by

$$U_{\log}(r) = p \log(ar + 1)$$

where a is a shape parameter and p is a scaling parameter that can be interpreted as the relative priority of the flow. Computing the first derivative $U'_{\log}(r)$ with respect to r yields

$$U'_{\log}(r) = \frac{pa}{ar + 1}$$

Then, computing the value of $r = r_{opt}$ that optimizes the Lagrangian equation $U_{\log}(r) - \lambda r$ yields

$$U'_{\log}(r_{opt}) - \lambda = 0$$

Defining λ in terms of the queue depth Q and a normalization constant K yields

$$\lambda = \frac{Q}{k}$$

such that the optimal rate r_{opt} is given by

$$r_{opt} = \frac{pak - Q}{aQ}$$

For the special case when the queue depth Q is zero, this yields an infinite value for the rate r_{opt} . Even though a ‘send as fast as possible’ condition is quite acceptable from an admission control perspective, checks are necessary to ensure that the computation of r_{opt} doesn’t result in a divide-by-zero condition. Hence this utility function (currently) also requires specifying a parameter m that defines the maximum rate in units of bits per second.

5.3.4.4 Elastic Traffic with Flow Completion Deadlines: Self-adapting Logarithmic Utility

This utility function is used to control the admission rate for applications flows that have a pre-defined delivery deadline that must either be met (in the context of other competing application flows) or should be terminated as soon as possible to avoid expending network resources unnecessarily. The key difference between this utility function and other utility functions is that elastic

traffic with completion deadlines effectively achieves no utility unless the entire transfer is completed. This type of utility function really has no intrinsic per-packet utility in the sense that the logarithmic utility function does, and so must be recast in order to work in a backpressure forwarded environment.

5.3.5 Flow Termination State Transitions

As described in Section 4.2.4, the proxy uses a hop-by-hop reliability model. When data packets are received by the proxy, they are placed into the proxy's internal send buffer and an ACK is generated by the proxy and sent back to the source. The delivery of packets to the receiving application, and receiving ACKs from the receiving application is decoupled and is handled separately.

In contrast, connection termination within the proxy uses an end-to-end model. When a FIN packet is received by the proxy, it is placed into the proxy's internal send buffer. The corresponding ACK for the received FIN is not sent back to the local application until the FIN is received by the remote application, the remote application acknowledges it, and the local proxy has received that ACK.

To help understand the nuances in play here, we note that there are effectively two coupled state machines within an instance of the TCP proxy, one managing transactions on the LAN facing interface, and the other managing transactions on the WAN facing interface. For normal data packet exchanges these can operate nearly independently; however, with an end-to-end termination model, these now need to be coordinated to have a 'correct' behavior from the perspective of the applications. This dual-but-coordinated state machine model means that the "standard" TCP state transitions within the proxies, as described in Stevens' TCP/IP Illustrated, Volume 1, need to be appropriately modified. The modified flow termination state transitions for the proxy's internal interfaces are described below.

5.3.5.1 3-Way Flow Termination State Transitions

Figure 23 depicts a typical 3-way flow termination handshake from the perspective of the TCP proxy nearest the application initiating the termination processing.

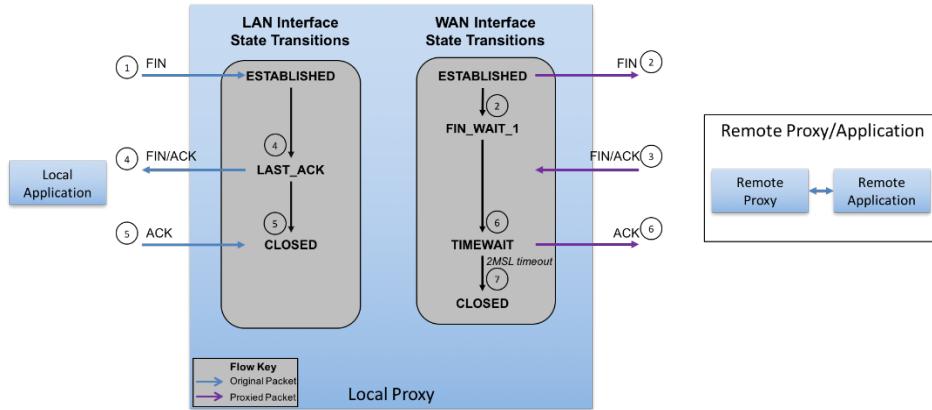


Figure 23. 3-Way TCP Flow Termination State Transitions

The following series of events occur during the flow termination sequence depicted in Figure 23:

1. The LAN Interface receives a FIN packet from the local application. An ACK for the FIN is not sent back to the local application at this time, and the LAN side interface remains in the ESTABLISHED state.
2. The WAN Interface sends the local application FIN to the remote application and transitions to the FIN_WAIT_1 state.
3. The WAN Interface receives a FIN/ACK from the remote application. This acknowledges receipt of the local application FIN by the remote application and contains the remote application FIN. An ACK for the FIN received from the remote application is not sent back towards the remote application at this time.
4. The LAN Interface sends the FIN/ACK to the local application. Since the LAN Interface has now received a FIN from the local application and sent an ACK for the FIN to the local application, it transitions to the LAST_ACK state.
5. The LAN Interface receives an ACK from the local application for the remote application FIN and transitions to the CLOSED state.
6. The WAN Interface sends the ACK for the remote application FIN. Since the WAN Interface has now received a FIN from the remote application and sent an ACK to the remote application, it transitions to the TIMEWAIT state. The 2MSL (Maximum Segment Lifetime) timer is started.
7. The 2MSL timeout occurs and the WAN Interface transitions to the CLOSED state.

Once both the LAN Interface and the WAN Interface have transitioned to the CLOSED state, the flow is no longer active and is removed from the proxy's internal state.

5.3.5.2 4-Way Flow Termination State Transitions

Figure 24 depicts a 4-way flow termination handshake from the perspective of the TCP proxy nearest the application initiating the termination processing. In this example, the local application performs a close that is acknowledged by the remote application, however the remote application is not yet ready to issue its own FIN. At some later time, the remote application then issues its own FIN, and both the LAN and the WAN interfaces can then proceed towards a CLOSED state.

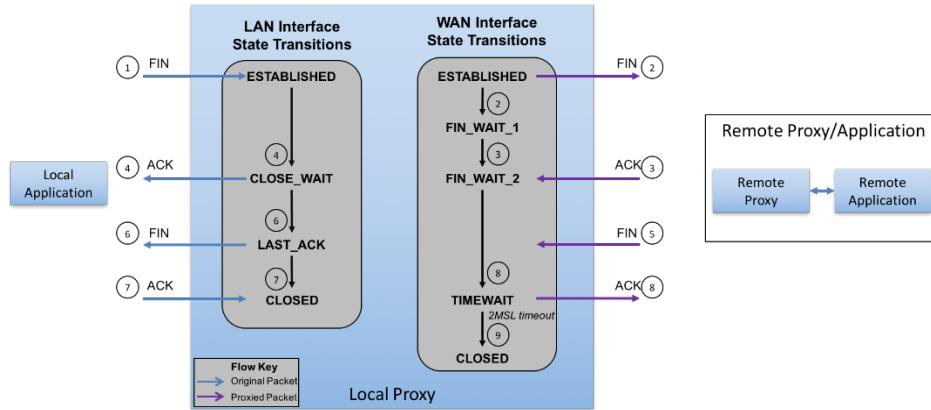


Figure 24. 4-Way TCP Flow Termination State Transitions

The following series of events occur during the flow termination sequence depicted in Figure 24:

1. The LAN Interface receives a FIN packet from the local application. An ACK for the FIN is not sent back to the local application at this time, and so remains in the ESTABLISHED state.
2. The WAN Interface sends the local application FIN to the remote application and transitions to the FIN_WAIT_1 state.
3. The WAN Interface receives an ACK from the remote application that acknowledges the local application FIN and transitions to the FIN_WAIT_2 state.
4. The LAN Interface sends the ACK for the local application's FIN to the local application. Since the LAN Interface has now received a FIN from the local application and has sent an ACK for the FIN to the local application, it transitions to the CLOSE_WAIT state.
5. The WAN Interface receives a FIN from the remote application. An ACK for the FIN is not sent back to the remote application at this time, and so remains in the FIN_WAIT_2 state.
6. The LAN Interface sends the FIN from the remote application to the local application and transitions to the LAST_ACK state.
7. The LAN Interface receives the ACK from the local application for the remote application FIN and transitions to the CLOSED state.
8. The WAN Interface sends the ACK for the remote application FIN to the remote application. Since the WAN Interface has now received a FIN from the remote application and sent an ACK to the remote application, it transitions to the TIMEWAIT state. The 2MSL (Maximum Segment Lifetime) timer is started.
9. The 2MSL timeout occurs and the WAN Interface transitions to the CLOSED state.

Once both the LAN Interface and the WAN Interface have transitioned to the CLOSED state, the flow is no longer active and is removed from the proxy's state.

5.3.5.3 Simultaneous Termination State Transitions

Figure 25 depicts the sequence of events that occur during a TCP simultaneous termination from the perspective of the TCP proxy nearest the application issuing the first FIN. Simultaneous termination occurs when both the local and remote applications perform a close at the very nearly the same time, such that FINs from both the local and remote applications essentially pass each other in the network.

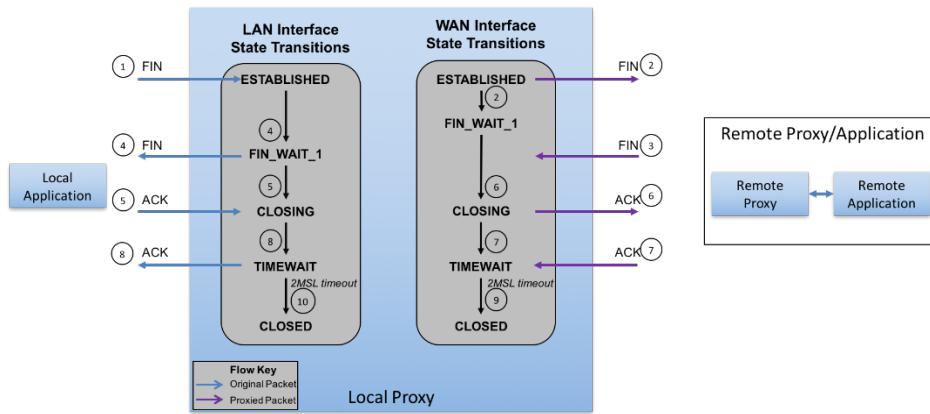


Figure 25. TCP Simultaneous Termination State Transitions

The following series of events occur during the flow termination sequence depicted in Figure 25:

1. The LAN Interface receives a FIN packet from the local application. An ACK for the FIN is not sent back to the local application at this time, so it remains in the ESTABLISHED state.
2. The WAN Interface sends the local application FIN to the remote application and transitions to the FIN_WAIT_1 state.
3. The WAN Interface receives a FIN packet from the remote application. An ACK for the FIN from the remote application is not sent back to the remote application at this time, so the WAN interface remains in the FIN_WAIT_1 state.
4. The LAN Interface sends the remote application FIN to the local application and transitions to the FIN_WAIT_1 state.
5. The LAN Interface receives the ACK from the local application for the remote application FIN and transitions to the CLOSING state.
6. The WAN Interface sends the ACK for the remote application FIN. Since the WAN Interface has now received a FIN from the remote application and sent an ACK for the FIN to the remote application, it transitions to the CLOSING state.
7. The WAN Interface receives the ACK from the remote application for the local application FIN and transitions to the TIMEWAIT state. The 2MSL timer is started.
8. The LAN Interface sends the ACK for the local application FIN to the local application. Since the LAN Interface has not received a FIN from the local application and sent an

ACK for the FIN to the local application, it transitions to the TIMEWAIT state. The 2MSL timer is started.

9. The 2MSL timeout occurs and the WAN Interface transitions to the CLOSED state.
10. The 2MSL timeout occurs and the LAN Interface transitions to the CLOSED state.

Once both the LAN Interface and the WAN Interface have transitioned to the CLOSED state, the flow is no longer active and is removed from the proxy's state.

5.3.6 Self-Adapting Buffer Sizes

The TCP Proxy must buffer received data so that missing packets in the data stream can be recovered (sometimes described as being repaired). In order to maintain transfer rates, the buffers should be sized so as to allow for the continued transmission of new data while retransmissions are occurring to recover any missing packets. Statically configuring the proxy buffer sizes to work for a wide range of network conditions (send rates, RTTs, and number of flows) can result in the proxy requiring huge amounts of memory that can quickly impede scalability. In order to reduce the memory footprint of the proxy and dynamically adapt to the possible wide range of network conditions, the proxy employs a self-adapting buffer strategy that is a function of the flow transfer rate and the measured RTT of the flow. The flow transfer rate is in turn a function of the number of competing flows, the flows' utility functions (in practice, the relative priorities), and the current queue backlog in the Backpressure Forwarder. Only the WAN-facing buffers are self-adapting. The current assumption is that the LAN side networking conditions will lead to very few losses which in turn will require very little buffering; hence the LAN-facing buffers are statically configured. The self-adapting buffer sizing operation, for WAN-facing buffers, is summarized as follows:

- Each flow has an initial WAN-facing send buffer size of 20 KB.
- Each flow has a maximum WAN-facing send buffer size of 3 MB. This is required to correctly convey the window scaling factor in the original SYN packet during the initial TCP handshake for the flow. If the initial buffer size is used to negotiate the window scaling factor, the scaling factor is not large enough to convey larger receive windows as the buffer size increases.
- The WAN-facing buffer size is updated once per RTT for each flow.
 - Buffer size increases, up to the maximum size of 3 MB, are not subject to any constraints.
 - Buffer size decreases are subject to the constraint that once a receive window is advertised, the advertised window will never decrease in size. The buffer size will only decrease when enough packets have been drained from the buffer such that the new upper edge of the advertised window moves to the right.

5.3.7 Controlling the Advertised Window Size

As is often the case, the inbound rate on the LAN-facing interface of the TCP Proxy greatly exceeds that of the outbound rate on the WAN-facing interface. This results in the TCP Proxy inbound buffers filling up and the TCP Proxy transmission of a zero window packet to the local application. As buffer space becomes available, the TCP Proxy "opens" the advertised window

indicating that the local application is permitted to resume transmitting packets. The TCP Proxy controls the size of the advertised window to the local application to maximize efficiency by minimizing the overhead associated with small packets.

The TCP Proxy ensures that any advertised window to the local application is a multiple of the Maximum Segment Size (MSS). This ensures that packets received from the local application contain the maximum amount of data which reduces the overhead associated with the packet headers. The TCP Proxy sets the MSS on the LAN-facing interface to 1320 bytes, 1280 bytes for data and 40 bytes for headers (IP and TCP headers). The MSS must account for any TCP options that are included in the packets. The TCP Proxy disables SACK on the LAN-facing interface to simplify the MSS size computation as the need to repair on the LAN-facing interface is deemed to be small and should be quick if the need does arise. If the LAN-facing interface supports the TCP Timestamp option then an additional 12 bytes are added to the MSS. This MSS and a maximum window scaling factor of 8 ensures that any advertised window to the local application is a multiple of the MSS.

6 Shared Memory IPC

6.1 Overview

6.1.1 Rationale

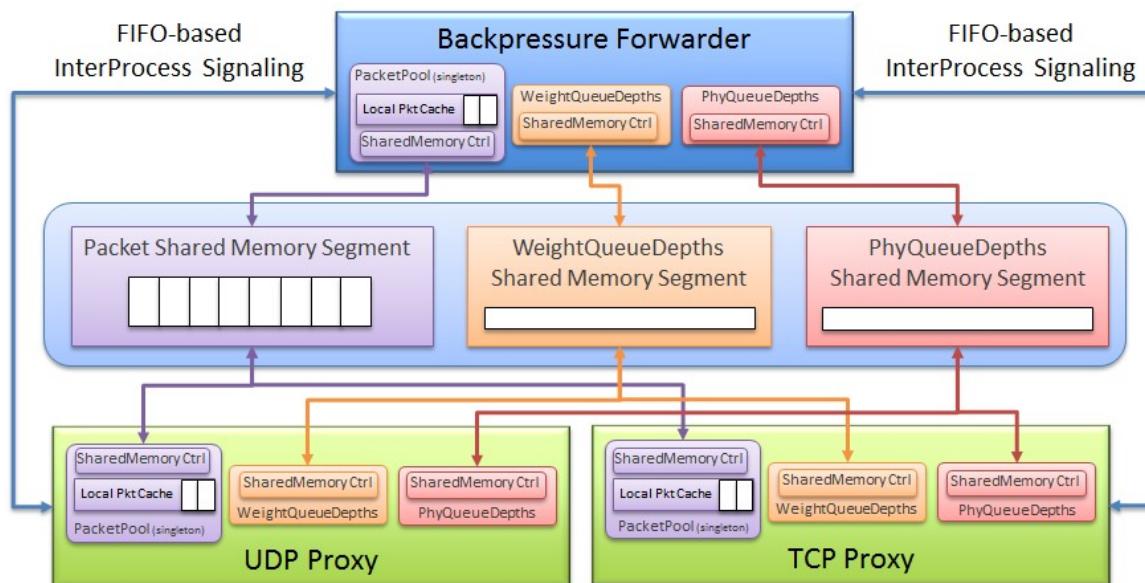
Each of the four major GNAT components, which include the AMP, the BPF, the UDP proxy and the TCP proxy, runs as a separate process. Since these processes are currently configured to all run on the same host, interprocess communications between the BPF and the two proxies is implemented using shared memory as a way of minimizing delay and processing overhead. This effectively enables a "zero-copy" approach for efficiently moving packets, queue depth information and other control information between components.

6.1.2 Architecture

There are multiple different shared memory segments used within GNAT. Each segment is identified by a name, has a given fixed size, and is embodied in a Shared Memory object to facilitate access to and use of the shared memory segment. Currently all shared memory segments are created and initialized by the BPF, and the proxies attach to them.

For example, Packet objects used by the different components are managed by a PacketPool object residing in each component, which in turn contains a SharedMemory object that arbitrates access to a common shared memory segment used for packets. Since all packets across the suite of GNAT components are allocated from a single shared memory segment, passing of packets from one component to another is as simple as handing of a reference (i.e., a pointer) from one component to another: copying packets is not needed.

While access to each shared memory segment is protected using locking mechanism (implemented using a semaphore), the shared memory object does not natively provide a convenient signaling mechanism, for instance to notify the backpressure forwarder that a new packet was sent from one of the proxies. This inter-process signaling is implemented using a named pipe (FIFO) that can be used in the same way that file-descriptors are typically used in select calls.



6.2 Shared Objects

6.2.1 Queue Depths

The BPF provides queue depth information to the UDP and TCP proxies via shared memory. The BPF provides two types of queue depth information. The first (weights) is used to compute queue differentials and admit packets. The second is the physical queue depth, which is used to detect when a queue is empty and prompt the admission of additional packets. Typically, weighted queue depths are equal to the physical queue depths for non-HeavyBall-type algorithms.

The BPF module creates all shared memory objects at start up, and the proxies attach to them. If for any reason a shared memory segment has not yet been created when a proxy attempts to attach to it, the proxy periodically retries attaching until the shared memory segment is available.

The queue depth objects are updated following these events:

- The BPF receives and processes one packet,
- The BPF transmits packets until it has nothing to do,
- The BPF transmits at least 5,000B.

Each of the proxies can be configured to access the queue depths objects using one of two different methods. The first method (the default method) maintains a local copy of the queue depths information and uses a timer to periodically update the local copies. The second method allows directly accessing the queue depths objects in shared memory, although this increases contention.

6.2.2 Packets

The PacketPool within the BPF creates and initializes a shared memory segment from which all packets used across the suite of GNAT components are pre-allocated. PacketPool objects (implemented as singletons within each GNAT component) coordinates access to packets from the common shared memory segment. In order to reduce the frequency with which packets are pulled from or returned to the shared memory segment, the local PacketPool within a GNAT component requests and caches multiple packets at a time so that handing out and recycling of packets is largely a local operation. Whenever the PacketPool packet cache is empty (i.e., the PacketPool has handed out its entire reserve of packets), the PacketPool object fetches additional packets from the shared memory segment. Alternately, whenever the PacketPool's packet cache fills up (i.e., the associated GNAT component has recycled many of its packets), packets are transferred back to the shared memory segment.

6.2.2.1 Locking Mechanism

Since different GNAT components may try to access the shared memory objects simultaneously, lightweight locks (implemented as semaphores) are provided. Before accessing a given shared memory segment, the caller must first capture the lock (and may spend some time waiting for it to become available). Upon acquiring a lock, the amount of time holding the lock must be kept to a minimum so as not to induce access contention. The lock must be released after the tasks have been performed.

The queue depth objects are read and copied with one `memcpy` command between calls to lock and unlock the semaphore.

The semaphore is deleted as part of the destructor for the shared memory object. However, it is

important to note that the OS maintains the shared memory segment in case the creating process disappears in order to avoid bad memory access. It does so until no process uses the shared memory.

6.3 Signaling Mechanism

The Inter-Process Signaling FIFO (fifo.h) supplements the shared memory implementation by providing an easy way for processes to exchange short messages.

The BPF establishes a unidirectional signaling channel with each proxy, and vice versa. Before sending a short message via this inter-process signaling FIFO, it checks whether the FIFO is already opened. If not, it tries to open it using a unique path name.

The receiver uses the FIFO's file descriptor to cache arriving messages in its select loop. The receiver may read one or more messages at a time.

The FIFO lets the BPF and proxies exchange packets by sending their index in the shared memory. This allows each receiving process to look up the packet at the specified index in their own memory space. (Note: each memory segment will in general have a different address in different processes; address translation between components is managed by the individual Shared Memory objects)

6.4 Events

Shared Memory Creation: The shared memory object may be called to create a segment and semaphore to provide protected access to various shared objects. If the object with the same identifying tuple (i.e., key, id) has already been created, the method returns.

Shared Memory Attach: The shared memory object may be called to attach to a segment and create a semaphore to provide protected access to various shared objects. If the object with the same identifying tuple has already been attached, the method returns.

Lock: The shared memory object must lock the semaphore before accessing the shared object. If unavailable, the call blocks until the semaphore is released.

Receive: The FIFO signals a new packet was received. The message includes an index to allow the receiving method to fetch the proper packet.

7 Capacity Adaptive Tunnels (CATs)

7.1 Summary of Module Goals and Objectives

The main goals of the CATs are:

- Controlling transmission rates to allow backpressure forwarding to work in an overlay environment. The CATs estimate and adapt to the underlying black network conditions, which may include shared bottlenecks, cross traffic, Denial of Service attacks, and other network events that can disrupt traffic flows.
- Eliminating the need for explicit coordination or joint estimation between IRON nodes. The CATs adapt to the black network conditions based upon their own feedback mechanisms and present a tunnel interface to the Backpressure Forwarding component.
- Providing flow-type adjustable reliability for improved performance and efficiency. For example, some applications might perform better with ARQ-based reliability, while others might do better with FEC-based reliability.

7.2 High Level Design

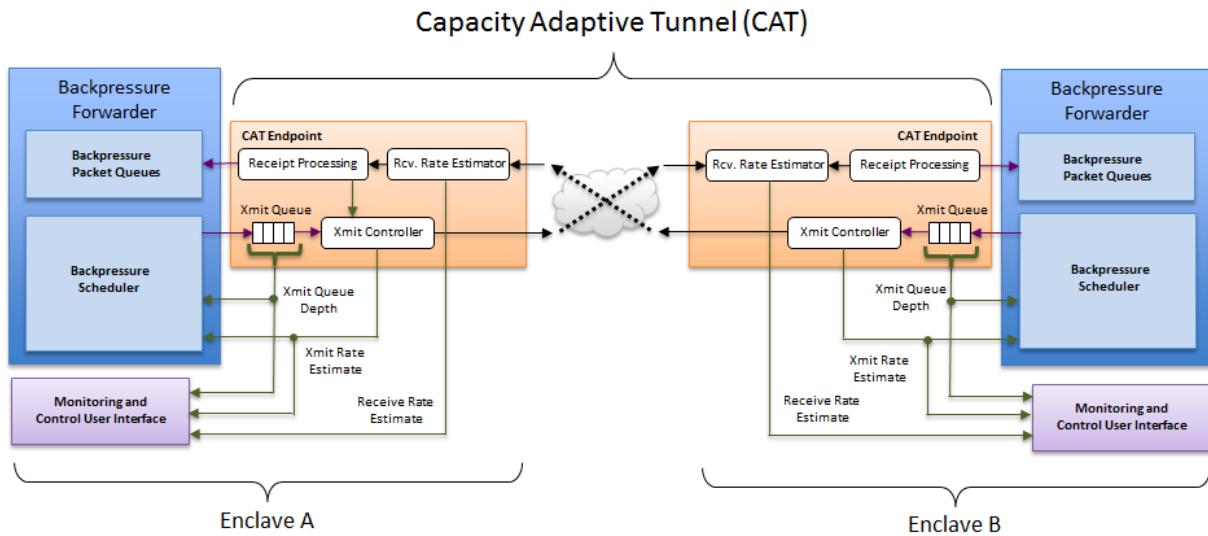
Capacity Adaptive Tunnels (CATs) provide the Path Controller functionality within the IRON architecture. CATs monitor the available capacity between adjacent enclaves and use this information to control the transmission rates in order to prevent packet loss from negatively impacting the backpressure forwarding. A pair of Path Controllers on adjacent enclaves collectively form a CAT between the enclaves. Each enclave may have multiple CAT endpoints, with each CAT leading to an adjacent enclave.

As a first approximation, a CAT can be thought of as a TCP connection between Backpressure Forwarders where the payload consists of application and control packets. However, for IRON the desirable properties for CATs are fairly different from TCP, and CATs should not be thought of simply as tunnels implemented using TCP. For example, TCP provides in-order, reliable delivery whereas a CAT does not provide in-order delivery and for some stream types may provide only semi-reliable delivery; for other stream types, it may provide only best effort services.

7.2.1 Adjacent Modules and Components

The role of a CAT is to move packets between Backpressure Forwarders located in two different enclaves separated by a network. A CAT consists of two endpoints, one in each enclave. Each CAT endpoint receives both data and control packets from the local Backpressure Forwarder, as well as from a remote Backpressure Forwarder via the corresponding CAT endpoint at the other end of the tunnel. CAT endpoints also exchange state information, generally embedded within the tunnel packet headers, that is used for both rate estimation and error control.

The CAT architecture is shown in the following figure. Note that black lines represent tunneled packets, the purple lines represent proxied packets, and the green lines represent information flows.



Each IRON enclave is assigned a unique *bin identifier* (BID), which is a small integer number that is used to quickly identify the Backpressure Forwarder assigned to that particular enclave. In terms of CATs, each local CAT endpoint is associated with one local BID, for the local Backpressure Forwarder, and one remote BID, for an adjacent IRON enclave's Backpressure Forwarder. Since Backpressure Forwarders are not assigned identifying network addresses, each CAT endpoint configuration must specify the local and remote network addresses to be used in setting up the CAT. This is necessary since a Backpressure Forwarder might have multiple network interfaces available and must use different network interfaces for its CAT endpoints.

7.2.2 Input Sources and Information Needed

7.2.2.1 Control Plane

Source	Input	Description
Remote CAT endpoint	Packet receipt information (e.g., ACKs, lists of missing and received packets, measurement data, etc.)	Feedback to the local IRON node CAT endpoint. This information is used to control the local CAT endpoint transmission rate and is included in-band.

7.2.2.2 Data Plane

Source	Input	Description
Local Backpressure Forwarder	Proxied application packets	These are packets that are destined to an adjacent IRON enclave. Each packet is directed to the appropriate CAT based on the remote BID of the tunnel to use as decided by the Backpressure Scheduler.
Remote Backpressure Forwarder via the remote CAT endpoint	Proxied application packets	These are packets that are arriving from an adjacent IRON enclave. Each local CAT endpoint is bound to a unique four tuple (source and destination addresses and ports) so that application packets are received from the matching CAT endpoint on a remote node.

Source	Input	Description
Local Backpressure Forwarder	Link state advertisements	These packets are either generated or forwarded by the local Backpressure Forwarder. They contain overlay network topology and latency information for a single Backpressure Forwarder. They are used by the remote Backpressure Forwarder in making path selection decisions for packets from latency-sensitive applications by each of the Backpressure Forwarders along the path. Each link state advertisement is directed to the appropriate CAT based on the remote BID of the tunnel to use as decided by the Backpressure Forwarder.
Remote Backpressure Forwarder via the remote CAT endpoint	Link state advertisements	These packets are either generated or forwarded by the remote Backpressure Forwarder. They contain overlay network topology and latency information for a single Backpressure Forwarder. They are used by the local Backpressure Forwarder in making path selection decisions for packets from latency-sensitive applications by each of the Backpressure Forwarders along the path. Each local CAT endpoint is bound to a unique four tuple (source and destination addresses and ports) so that link state advertisements are received from the matching CAT endpoint.
Local Backpressure Forwarder	Queue length advertisement messages	These are packets generated by the local Backpressure Forwarder and sent to the remote Backpressure Forwarder for use in making forwarding decisions. Each queue length advertisement is directed to the appropriate CAT based on the remote BID of the tunnel to use as decided by the Backpressure Scheduler.
Remote Backpressure Forwarder via the remote CAT endpoint	Queue length advertisement messages	These are packets generated by the remote Backpressure Forwarder being sent to the local Backpressure Forwarder for use in making forwarding decisions. Each local CAT endpoint is bound to a unique four tuple (source and destination addresses and ports) so that queue length advertisement messages are received from the matching CAT endpoint.
Local Backpressure Forwarder	Group advertisement messages	These are packets generated by the local Backpressure Forwarder and sent to the remote Backpressure Forwarder for use in controlling multicast group memberships. Each group advertisement message is directed to the appropriate CAT based on the remote BID of the tunnel to use as decided by the Backpressure Forwarder.
Remote Backpressure Forwarder via the remote CAT endpoint	Group advertisement messages	These are packets generated by the remote Backpressure Forwarder and sent to the local Backpressure Forwarder for use in controlling multicast group memberships. Each local CAT endpoint is bound to a unique four tuple (source and destination addresses and ports) so that group advertisement messages are received from the matching CAT endpoint.
Local Backpressure Forwarder	Receiver report messages	These are packets generated by the local Backpressure Forwarder and sent to the remote Backpressure Forwarder for transferring packet transmit and receipt information between UDP proxies. Each receiver report message is directed to the appropriate CAT based on the remote BID of the tunnel to use as decided by the Backpressure Forwarder.
Remote Backpressure Forwarder via the remote CAT	Receiver report messages	These are packets generated by the remote Backpressure Forwarder and sent to the local Backpressure Forwarder for transferring packet transmit and receipt information between UDP proxies. Each local CAT endpoint is bound to a unique four tuple

Source	Input	Description
endpoint		(source and destination addresses and ports) so that receiver report messages are received from the matching CAT endpoint.

7.2.3 Output Destinations and Information Provided

7.2.3.1 Control Plane

Destination	Output	Description
Local Backpressure Forwarder	Transmit queue length in bytes	This information is used to inform the local Backpressure Scheduler when there is room in the CAT transmit buffer to accept more packets.
Local Backpressure Forwarder	Tunnel available channel and transport capacity estimates over the CAT	This information is passed to the local Backpressure Scheduler to help it control the rate at which queue length advertisement messages are sent. This allows limiting the amount of control overhead used for backpressure forwarding. As an example, the TCP Vegas available capacity estimation algorithm computes a target steady-state capacity estimate.
Local Backpressure Forwarder	Tunnel packet delivery delay estimates	This information is used by the local Low-Latency Backpressure Scheduler to determine the time to reach a destination and make low-latency routing decisions.
Remote CAT endpoint	Packet receipt information (e.g., ACKs, lists of missing and received packets, measurement data, etc.)	This is sent to the paired CAT endpoint on a remote IRON node and is used to control its transmission rate and reliability behaviors. It is included in-band, as headers in the encapsulating tunnel packets.

7.2.3.2 Data Plane

Destination	Output	Description
Local Backpressure Forwarder	Proxied application packets from the remote Backpressure Forwarder, via the remote CAT endpoint	These are packets from a remote IRON enclave that are to be handled by the local Backpressure Forwarder. They include packets that will be forwarded to other adjacent IRON enclaves and packets destined to the local IRON enclave.
Remote Backpressure Forwarder via the remote CAT endpoint	Proxied application packets	These are packets from the local IRON enclave that are being forwarded to the remote Backpressure Forwarder. They include packets that will be forwarded to other IRON enclaves and packets destined to the remote IRON enclave.
Local Backpressure Forwarder	Link state advertisements	These packets are either generated or forwarded by the remote Backpressure Forwarder. They contain overlay network topology and latency information for a single Backpressure Forwarder. They are used by the local Backpressure Forwarder in making path selection decisions for packets from latency-sensitive applications by each of the Backpressure Forwarders along the path.
Remote Backpressure Forwarder via the remote CAT endpoint	Link state advertisements	These packets are either generated or forwarded by the local Backpressure Forwarder. They contain overlay network topology and latency information for a single Backpressure Forwarder. They are used by the remote Backpressure Forwarder in making path selection decisions for packets from latency-sensitive applications by each of the Backpressure Forwarders along the path.
Local Backpressure	Queue length	These are packets generated by the remote Backpressure

Destination	Output	Description
Forwarder	advertisement messages	Forwarder being sent to the local Backpressure Forwarder for use in making forwarding decisions.
Remote Backpressure Forwarder via the remote CAT endpoint	Queue length advertisement messages	These are packets generated by the local Backpressure Forwarder being sent to the remote Backpressure Forwarder for use in making forwarding decisions.
Local Backpressure Forwarder	Group advertisement messages	These are packets generated by the remote Backpressure Forwarder being sent to the local Backpressure Forwarder for use in controlling multicast group memberships.
Remote Backpressure Forwarder via the remote CAT endpoint	Group advertisement messages	These are packets generated by the local Backpressure Forwarder being sent to the remote Backpressure Forwarder for use in controlling multicast group memberships.
Local Backpressure Forwarder	Receiver report messages	These are packets generated by the remote Backpressure Forwarder being sent to the local Backpressure Forwarder for transferring packet transmit and receipt information between UDP proxies.
Remote Backpressure Forwarder via the remote CAT endpoint	Receiver report messages	These are packets generated by the local Backpressure Forwarder being sent to the remote Backpressure Forwarder for transferring packet transmit and receipt information between UDP proxies.

7.2.4 Events

CAT Startup: Upon startup, the Path Controller (CAT) instances are created by the local Backpressure Forwarder. Note that each CAT instance will normally lead to a different adjacent IRON enclave, but there may be multiple CATs leading to the same adjacent IRON enclave. The CATs get their configuration information from the Backpressure Forwarder, which originally gets the configuration information from the Monitoring and Control User Interface. The Backpressure Forwarder configuration information includes details on the number of CATs that are to be created. The configuration information for each CAT includes any required connectivity details, an optional label string, and other optional settings. The connectivity information can simply specify the associated CAT's destination address for non-connection-oriented CATs, or can specify detailed connection information for connection-oriented CATs. The optional CAT label string is used for differentiating the various CATs when the CAT statistics are reported to the Monitoring and Control User Interface. Other optional CAT settings can include aggressiveness settings (e.g., the transmit rate control algorithms and parameters to use) and reliability settings (e.g., the level of reliability required and how that reliability is accomplished).

If a connection-oriented channel is utilized for a CAT, then the two ends of the tunnel must coordinate which one will perform the "listen" operation (i.e. act as the "server" endpoint) and which will perform the "connect" operation (i.e., act as the "client" endpoint). This information might be specified directly in the configuration information for each CAT. However, in order to simplify configuration, this might be accomplished by comparing the local and remote IP addresses for each pair of CATs that are creating a single tunnel between enclaves, with the end having the

"lower" numerical address always performing one of the operations and the other end performing the other operation.

If a TCP connection is used to implement a CAT, then various TCP options may be set in order to achieve the desired behavior. For example, Nagle's algorithm (which is used to aggregate small data transfers into a single larger packet for efficiency) may need to be enabled or disabled, depending on the desired tunnel behavior.

Backpressure Packet Transmission: The Backpressure Scheduler releases proxied data packets and backpressure control packets to a CAT instance for transfer to the CAT's adjacent IRON enclave. Data packets are directed to CATs based on the BID of the backpressure packet queue that the packet was dequeued from. Backpressure control packets containing link state advertisements (LSAs) are flooded based on where the packet was generated and received from. Backpressure control packets containing queue length advertisement message (QLAM) or receiver report message (RRM) information are directed to the CATs based on the BID associated with the event. Backpressure control packets containing multicast group advertisement message (GRAM) information are directed to the CATs based on the BIDs that have not received the packet yet. This data plane input causes the CAT to queue the packet in its transmit queue. It then sends a control plane output consisting of the current transmit queue depth in bytes back to the Backpressure Scheduler.

CAT Packet Transmission: When the CAT's transmit rate control algorithm allows a packet to be sent to the adjacent IRON enclave, it dequeues a packet from its transmit queue, adds any necessary local packet receipt information (a control plane output) and tunneling information, updates any packet information that must be updated just before transmission, and sends it out the INE interface to the other end of the tunnel. It then sends a control plane output consisting of the current transmit queue depth in bytes to the Backpressure Scheduler. When the CAT transmit controller has enough data being transmitted and congestion control information gathered, the CAT instance will convert the information into a black network channel/transport capacity estimate and send a control plane output of this rate estimate to the Backpressure Scheduler.

CAT Packet Reception: This data plane input occurs when a packet sent by the adjacent IRON enclave arrives at a CAT instance from the INE interface. The packet is directed to the correct CAT based on the destination IP address and destination TCP/UDP port number. First, any tunneling information is removed from the packet. If the packet contains any remote packet receipt information from the paired IRON node CAT instance, this is removed from the packet and used in the local transmit rate control algorithm. The resulting packet is then queued in a receive queue and passed as a data plane output to the backpressure packet queues when the tunneling receive rules dictate.

When a QLAM packet is received by a CAT and passed to the Backpressure Scheduler, the Backpressure Scheduler extracts the source's BID from the packet and creates an association between the receiving CAT and the extracted BID. The BID can then be mapped back to the CAT, creating a mapping for sending packets to the correct tunnel.

7.2.5 Configurable Parameters

The CAT configuration information is included in the Backpressure Forwarder configuration information, and includes the following parameters.

- The number of CATs to be created. The Backpressure Forwarder uses this information to create its CAT endpoint instances. This information comes from the Monitoring and Control User Interface at configuration time.
- A set of connectivity details for each CAT. The connectivity information can simply specify the associated CAT endpoint's destination address for non-connection-oriented CATs, or can specify detailed connection information for connection-oriented CATs. This information comes from the Monitoring and Control User Interface at configuration time.
- An optional label string for each CAT. This label is used when the CAT statistics are reported to the Monitoring and Control User Interface, and is useful for differentiating between parallel tunnels between two IRON enclaves. This information comes from the Monitoring and Control User Interface at configuration time.
- A set of optional aggressiveness settings for the CAT. This can include the transmit rate control algorithm to be used, and any associated parameters to use with the algorithm. This information comes from the Monitoring and Control User Interface at configuration time.
- A set of optional reliability settings for the CAT. This can include the level of reliability required (e.g., full reliability, partial reliability, or no reliability) and how that reliability is to be accomplished (e.g., using ARQ or FEC), along with any associated parameters to be used. This information comes from the Monitoring and Control User Interface at configuration time.

7.2.6 Performance Monitoring Statistics

Each CAT instance provides the following statistics for use by the Monitoring and Control User Interface, as well as any other statistics gathering tool.

Black Network Channel Capacity Estimate: This is a computed statistic for each CAT instance. It is based on the number and size of all packets transmitted to the adjacent IRON enclave over an interval of time when the transmissions are being rate limited. It provides an indication of the black network conditions to the adjacent IRON enclave.

Black Network Transport Capacity Estimate: This is a computed statistic for each CAT instance. It is based on the number and size of the data provided by the Backpressure Forwarder for transmission to the adjacent IRON enclave over an interval of time when the transmissions are being rate limited. It provides a measure of the amount of tunneled data that can be sent to the adjacent IRON node by the local Backpressure Forwarder.

Packet Delivery Delay Estimate: This is a computed statistic for each CAT instance. It is based on the round-trip and one-way time measurements to/from the adjacent IRON enclave, and is computed as a mean and a variance of the local to remote one-way delay estimates. It provides a

measure of how long a latency-sensitive data packet would take to be transferred to the adjacent IRON enclave, and how that delay might vary.

7.3 Detailed Design

It is possible for a CAT to utilize a wide range of tunneling protocols. The primary CAT developed for the IRON program leverages a UDP-based transport protocol called SLIQ for exchanging datagrams between IRON enclaves. The SLIQ packets pass from the local IRON node, through the local Inline Network Encryptor (INE), over the black network, through the remote INE, and to the remote IRON node. SLIQ currently supports two different congestion control algorithms to control transmission rates in order to share the available network capacity fairly. These algorithms are CUBIC and Copa, with three versions of Copa available (a Copa Beta 1 version, a Copa Beta 2 version, and the final Copa version). A SLIQ CAT was created to implement the necessary Path Controller interface for the backpressure forwarder, and uses the SLIQ protocol along with one or more of its congestion control algorithms to implement the tunnel over the black network.

This section provides a detailed design of the SLIQ protocol, the SLIQ congestion control algorithms that have been implemented to date, and the resulting SLIQ CAT.

7.3.1 SLIQ Protocol

SLIQ (Simple Lightweight IPv4 QUIC) is a UDP-based transport protocol that operates entirely in user space. It was originally based on Google's QUIC (Quick UDP Internet Connection) protocol, although SLIQ has evolved and is now significantly different than QUIC. SLIQ is implemented in C++ as a library that is linked into the application software. As the name suggests, SLIQ currently supports IPv4 only (not IPv6).

The SLIQ protocol has the following features:

- Provides a connection-oriented datagram transport protocol.
- Reliability is provided by selective acknowledgement (selective ACK) packets, data packet retransmissions, and Forward Error Correction (FEC) data packets.
- Supports multiple concurrent data transfers as separate bi-directional streams within a single connection.
- Supports prioritization of the streams within each connection.
- Implements a connection-level pluggable congestion control framework.
- Supports optional send pacing for each congestion control algorithm.
- Implements stream-level flow control using a fixed receive window size (32,768 packets) for simplicity.
- Supports stream-level datagram reliability modes including fully reliable (using retransmissions), semi-reliable (using a limited number of retransmissions or FEC packets), and best-effort (no retransmissions or FEC packets).
- Supports stream-level in-order and out-of-order datagram delivery modes.
- Supports stream-level datagram delivery deadlines using Latency-Constrained Adaptive Error Control (AEC).

- Detects link outages (complete loss of connectivity) and actively monitors for when the outage ends for fast recovery.
- Provides connection-level channel capacity estimates and transport capacity estimates to the application.
- Allows configuring each stream's transmit queue maximum size, dequeuing policy, and drop policy.

At startup, SLIQ first creates a connection in one of two possible ways. The first is the server listening method, similar to how TCP creates a connection. With this method, the SLIQ server endpoint starts up first, listening on a well-known port number. Then, SLIQ client endpoints connect to the SLIQ server using a three-way packet exchange, establishing separate, independent SLIQ connections. Alternatively, the server and client endpoints can be set up using a direct connection method, where each endpoint configures its end of the connection in one step, followed by the three-way packet exchange. This method eliminates the need for the listening step, simplifying the server application. Within each of the SLIQ connections that are established, the two endpoints can create one or more bi-directional streams for actually transferring datagrams of application data. Each stream has a priority (0-7, with 0 being the highest priority), a reliability mode (reliable, semi-reliable, or best effort), and a delivery mode (in-order datagram delivery, or out-of-order datagram delivery). Congestion control is applied to each direction of each connection to make sure that the available network capacity is not exceeded, and is shared with other network traffic (both SLIQ and non-SLIQ traffic).

SLIQ is intended for use as a tunneling protocol, and operates by encapsulating and forwarding each payload it is provided atomically; SLIQ does not break larger payloads into smaller payloads, nor does it merge multiple payloads together prior to transport. As such, SLIQ does not provide datagram fragmentation and reassembly. If the payload sizes cause SLIQ packets to exceed the MTU of the underlying IP network, then IP fragmentation will occur. While not optimal from a loss perspective (since if at least one IP fragment is lost, then the entire SLIQ packet is lost), the protocol will still function when this occurs.

The SLIQ library is implemented to use packets from the Backpressure Forwarder's Packet Pool, which is implemented using shared memory. Received datagram payloads are placed in these Packet objects to be delivered to the Backpressure Forwarder via the SLIQ CAT. Datagram payloads that are sent by the Backpressure Forwarder are passed to SLIQ, through the SLIQ CAT, in these same Packet objects. The SLIQ library returns Packet objects to the common Packet Pool when they are no longer needed. The use of this common Packet Pool allows packets to be passed between computer processes without having to copy the data over and over, significantly improving performance.

Note that in the following discussions regarding the SLIQ protocol, the "application" is the SLIQ CAT.

7.3.1.1 Packet Header Formats

SLIQ currently defines 11 different header types for use in SLIQ packets. The first byte of each header has a one-byte Header Type field that identifies the type of SLIQ header. This supports

efficient header identification and extensibility, potentially allowing up to 255 different headers types to be defined. However, since the first byte of each packet is also used to distinguish between SLIQ CAT, Backpressure Forwarder, and IPv4 headers as well, the SLIQ Header Type values have been carefully selected to avoid conflicts and allow for quick SLIQ header rule verification. Thus, all SLIQ connection maintenance, headers must use Header Type values between 0 and 15, all SLIQ data transfer headers that can be concatenated together must use Header Type values between 32 and 39, and all SLIQ stand-alone headers must use Header Types values between 40 and 47. The current set of SLIQ headers easily fits within these Header Type ranges.

Note that in each of the SLIQ headers, all multi-byte fields are transmitted in network byte order (big-endian), and all unused fields in SLIQ headers must be set to zero when transmitted.

Each SLIQ header type is described in the following subsections.

7.3.1.1.1 Connection Handshake Header

The connection handshake header is used to establish a connection between a client endpoint and a server endpoint, and is sent back and forth between the two endpoints with different message tags for each stage of the establishment process. This header must be sent by itself, with no other SLIQ headers or data following it. The client first sends a "CH" tag (short for "Client Hello") to the server, the server responds to the client with an "SH" tag (short for "Server Hello"), and the client then responds to the server with a "CC" tag (short for "Client Confirm"). The "RJ" tag (short for "Reject") is sent by either endpoint if an error occurs during the connection establishment process.

The source of the header inserts its own locally-generated timestamp, in microseconds, in the Packet Timestamp field when the packet is sent. Since a timestamp value of zero is to be interpreted as the absence of a timestamp, a timestamp value of one should be sent if the current time generates a timestamp value that is zero. If the header is sent in response to a previously received header, then the received Packet Timestamp field is placed unaltered into the Echo Timestamp field in the response header. Otherwise, the Echo Timestamp field is set to zero.

The client endpoint selects the congestion control algorithms that are to be used on both endpoints of the connection, and this header includes all of the congestion control algorithm settings in order to transfer them to the server endpoint. Each congestion control setting takes 8 bytes within the header.

The client selects a unique 32-bit client identifier at random and places it in the Unique Client ID field in the "CH" tagged connection handshake header that is sent. This identifier is also placed in the Unique Client ID field in each of the following connection handshake headers that are exchanged for the connection. This field allows SLIQ to distinguish between connection handshake headers for different connections that might become intermingled.

Once the connection is successfully established using connection handshake headers, SLIQ identifies the connection using the UDP/IP five-tuple (protocol UDP, local endpoint IP address, local endpoint UDP port number, remote endpoint IP address, and remote UDP port number) for each packet received.

The connection handshake header format is shown below.

```
.0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+
|      Type      | # of CC Alg   |           Message Tag          |
+-----+-----+-----+
|           Packet Timestamp          |
+-----+-----+-----+
|           Echo Timestamp          |
+-----+-----+-----+
| CC Alg Type #1| Unused |D|P|           Unused          |
+-----+-----+-----+
|           CC Alg Parameters #1          |
+-----+-----+-----+
~           ~           ~
+-----+-----+-----+
| CC Alg Type #N| Unused |D|P|           Unused          |
+-----+-----+-----+
|           CC Alg Parameters #N          |
+-----+-----+-----+
|           Unique Client ID          |
+-----+-----+-----+
```

The connection handshake header fields are:

- **Type** (1 byte) - Header type, set to 0x00
- **Number of Congestion Control Algorithms** (1 byte) - The number of congestion control algorithm parameter sets contained in this header, must be greater than or equal to one
- **Message Tag** (2 bytes) - The handshake message tag, consisting of two UTF-8 characters ("CH" = Client Hello, "SH" = Server Hello, "CC" = Client Confirm, "RJ" = Reject)
- **Packet Timestamp** (4 bytes) - The source timestamp in microseconds, valid timestamps are non-zero
- **Echo Timestamp** (4 bytes) - The timestamp in microseconds being returned to its source, copied from the previously received handshake message Packet Timestamp, valid timestamps are non-zero

For each congestion control parameter set included (as determined by the Number of Congestion Control Algorithms field), the required fields are:

- **Congestion Control Algorithm Type** (1 byte) - Congestion control type (0 = None, 1 = Google's CUBIC, 2 = Google's Reno, 3 = Linux Kernel's CUBIC, 4 = Copa Beta 1 with Constant Delta, 5 = Copa Beta 1 with Maximum Throughput Policy Controller, 6 = Copa Beta 2, 7 = Copa)
- **Flags** (1 byte) - Bit field of flags
 - **U** (6 bits) - Unused, set to zero
 - **D** (1 bit) - Deterministic Congestion Control Flag, Copa Beta 1 only, otherwise set to zero
 - **P** (1 bit) - Send Pacing Flag, Google's CUBIC/Reno only, otherwise set to zero
- **Unused** (2 bytes) - Unused, set to zero

- **Congestion Control Algorithm Parameters** (4 bytes) - Congestion control-specific parameters

Note that at least one congestion control algorithm must be specified for each connection. The first algorithm listed in the header is assigned a SLIQ Congestion Control Identifier (CCID) of zero, then next one, etc. CCIDs are specified in data headers, congestion control synchronization headers, and congestion control packet train headers.

Following all of the congestion control parameter sets, the required fields are:

- **Unique Client ID** (4 bytes) - A unique identifier for the connection establishment attempt, selected by the client, included in each handshake message exchanged for the connection

7.3.1.1.2 Reset Connection Header

The reset connection header is used to immediately close a connection. This header must be sent by itself, with no other SLIQ headers or data following it. It may be sent by either endpoint, and when received, the connection is immediately closed, even if data is lost. The header is only sent when an unrecoverable error occurs and the connection can no longer continue. For normal connection tear down, the close connection header should be used instead of this header.

The reset connection header format is shown below.

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
-----	-----	-----	-----
Type	Flags (Unused)	Error Code	
-----	-----	-----	-----

The reset connection header fields are:

- **Type** (1 byte) - Header type, set to 0x01
- **Flags** (1 byte) - Bit field of flags, all unused, set to zero
- **Error Code** (2 bytes) - The reason for the reset (0 = no error, 1 = receive close error, 2 = socket write error, 3 = internal error)

7.3.1.1.3 Close Connection Header

The close connection header is used to gracefully close an existing connection. This header must be sent by itself, with no other SLIQ headers or data following it. It may be sent by either endpoint, and it is sent back with the ACK flag set to acknowledge the close.

The close connection header format is shown below.

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
-----	-----	-----	-----
Type	Unused	A	Reason Code
-----	-----	-----	-----

The close connection header fields are:

- **Type** (1 byte) - Header type, set to 0x02
- **Flags** (1 byte) - Bit field of flags
 - **Unused** (7 bits) - Unused, set to zero
 - **A** (1 bit) - ACK Flag
- **Reason Code** (2 bytes) - The reason for the close (0 = normal close, 1 = flow control sent too much data)

7.3.1.1.4 Create Stream Header

The create stream header is used to establish a new stream within an existing connection. This header must be sent by itself, with no other SLIQ headers or data following it. It may be sent by either endpoint, and is sent back with the ACK flag set to acknowledge the stream creation. If the stream cannot be created by the endpoint, then there is no create stream header sent back to the originator.

Streams are bi-directional data flows within a connection. Each stream is identified by a unique Stream ID, which is an integer between 1 and 32, inclusive. In order to avoid stream creation conflicts, streams initiated by the client endpoint must use odd Stream ID numbers, and streams initiated by the server endpoint must use even Stream ID numbers.

The create stream header includes all of the error control algorithm settings to be used by both endpoints for the stream. ARQ is short for *automatic repeat request*, which uses acknowledgement (ACK) packets and timeouts along with data packet retransmissions to recover lost data packets. FEC is short for *forward error correction*, which uses redundant error-correcting packets sent along with the original data packets to detect and correct a limited number of lost data packets at the receiving endpoint. The best effort reliability mode does not attempt to perform any error control. The semi-reliable modes, semi-reliable ARQ and semi-reliable ARQ+FEC, attempt to provide reliable delivery of all data using ARQ and possibly FEC, but only up to a maximum number of data packet retransmissions. The semi-reliable ARQ mode will only use ARQ for recovering lost packets within the specified number of data packet retransmissions. The semi-reliable ARQ+FEC mode will dynamically use ARQ and/or FEC as needed to attempt to meet a specified datagram delivery deadline for the stream. The selection of which error control methods to use in this mode is decided by the Latency-Constrained Adaptive Error Control (AEC) algorithm. The reliable ARQ reliability mode provides full reliability of all data, regardless of the number of data packet retransmissions required.

The delivery mode setting controls the order of the received data delivered to the receiving endpoint. Setting the delivery mode to ordered delivery mode guarantees that the data is delivered in the correct order to the receiving application, which also requires a guarantee that all of the data will be delivered no matter how long that might take, similar to TCP. Setting the delivery mode to unordered delivery mode makes no guarantees about data delivery order to the receiving application. Note that the ordered delivery mode is only possible if the reliability mode is set to reliable ARQ.

In general, applications that require all data to be transferred without concern over the data delivery delay should choose reliable ARQ and ordered delivery. Applications that require some fraction of the data to be delivered within a specific data delivery delay should choose one of the semi-reliable modes and unordered delivery. Applications that do not need to worry about lost data at all should choose best effort and unordered delivery.

All of the parameters necessary for creating the stream by the receiving endpoint are included in this header. The create stream header format is shown below.

```
.0          1          2          3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+
|      Type   | Unused  |T|A| Stream ID | Priority   |
+-----+-----+-----+-----+
|          Initial Window Size           |
+-----+-----+-----+-----+
|          Initial Packet Sequence Number |
+-----+-----+-----+-----+
| Del | Rel | Rexmit Limit | FEC Target Delivery Rnds/Time |
+-----+-----+-----+-----+
| FEC Target Pkt Recv Probability| Unused        |
+-----+-----+-----+-----+
```

The create stream header fields are:

- **Type** (1 byte) - Header type, set to 0x03
- **Flags** (1 byte) - Bit field of flags
 - **U** (6 bits) - Unused, set to zero
 - **T** (1 bit) - Delivery Time Flag, semi-reliable ARQ+FEC mode only, otherwise set to zero
 - **A** (1 bit) - ACK Flag
- **Stream ID** (1 byte) - Stream identifier, 1-32, odd for client-initiated streams, even for server-initiated streams
- **Priority** (1 byte) - Stream priority, 0-7 (0 is highest priority, 7 is lowest priority)
- **Initial Window Size** (4 bytes) - Initial stream window size in packets, must be set to 32768
- **Initial Packet Sequence Number** (4 bytes) - Initial stream data packet sequence number
- **Delivery Mode** (4 bits) - Delivery mode for the stream (0 = unordered delivery mode, 1 = ordered delivery mode)
- **Reliability Mode** (4 bits) - Reliability mode for the stream (0 = best effort, 1 = semi-reliable ARQ, 2 = semi-reliable ARQ+FEC, 4 = reliable ARQ)
- **Semi-Reliable Packet Delivery Retransmission Limit** (1 byte) - Stream data packet delivery retransmission limit, semi-reliable ARQ mode or semi-reliable ARQ+FEC mode only
- **FEC Target Delivery Rounds or Time** (2 bytes) - Target packet delivery rounds if Delivery Time Flag is set to zero, target packet delivery time in milliseconds if Delivery Time Flag is set to one, semi-reliable ARQ+FEC mode only
- **FEC Target Packet Receive Probability** (2 bytes) - Target packet receive probability times 10000, 1-9990, semi-reliable ARQ+FEC mode only
- **Unused** (2 bytes) - Unused, set to zero

7.3.1.1.5 Reset Stream Header

The reset stream header is used to immediately close a stream within a connection. This header must be sent by itself, with no other SLIQ headers or data following it. It may be sent by either endpoint, and when received, the stream is immediately closed, even if data is lost. The header is only sent when an unrecoverable error occurs and the stream can no longer continue. Receipt of this header does not cause the connection to be closed. For normal stream tear down, the data header FIN flag should be used, as this allows all of the stream's data to be delivered.

The reset stream header format is shown below.

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1
-----	-----	-----	-----
Type	Flags (Unused)	Stream ID	Error Code
-----	-----	-----	-----
Final Packet Sequence Number			
-----	-----	-----	-----

The reset stream header fields are:

- **Type** (1 byte) - Header type, set to 0x04
- **Flags** (1 byte) - Bit field of flags, all unused, set to zero
- **Stream ID** (1 byte) - Stream identifier, 1-32, odd for client-initiated streams, even for server-initiated streams
- **Error Code** (1 byte) - The reason for the reset (0 = no error, 1 = socket partial write error, 2 = socket write error, 3 = flow control error, 4 = transmit queue error)
- **Final Packet Sequence Number** (4 bytes) - Final stream data packet sequence number

7.3.1.1.6 Data Header

The data header is used to transfer payload data between endpoints for a stream within a connection. The combination of a data header and payload data is referred to as a data packet. The payload data is placed after the header as-is from the application, and it is neither fragmented nor combined with any other payload data by SLIQ. Each data packet is assigned the next sequence number for the stream, and the data packet's retransmission count is included each time it is sent. The original transmission of a data packet has the retransmission count set to zero.

The source of the header inserts its own locally-generated timestamp, in microseconds, in the Packet Timestamp field when the packet is sent. Since a timestamp value of zero is to be interpreted as the absence of a timestamp, a timestamp value of one should be sent if the current time generates a timestamp value that is zero. The Packet Timestamp Delta field is set to the timestamp difference value at the moment the most recent Packet Timestamp field (from any SLIQ header) was received from the remote endpoint. The timestamp difference value is computed as the 32-bit unsigned subtraction: the local timestamp minus the remote timestamp. Since a timestamp difference value of zero is to be interpreted as the absence of a timestamp difference value, a timestamp difference value of one should be sent if the timestamp difference value computation result is zero.

If the *M* flag is set, then this data header includes the optional Move Forward Packet Sequence Number field, which is 4 bytes long, immediately following the Packet Timestamp Delta field. Otherwise, the Move Forward Packet Sequence Number field is not included in the header.

If the data header's *E* flag is set, then it includes the optional FEC fields (the *T* flag field through the Encoded Packet Length field), which can be either 4 bytes or 6 bytes long, immediately following the optional Move Forward Packet Sequence Number field. Otherwise, these fields are not included. When the *E* flag is set, the Encoded Packet Length field, which is 2 bytes long, is only included if the *L* flag is set. The FEC Group Index field marks the data packet's position within the FEC group, which is always set to 0 to (K-1) for the K source data packets and K to (N-1) for the (N-K) encoded data packets. The FEC Group Identifier field is used to identify each FEC group, with the N packets for an FEC group being assigned the same FEC Group Identifier value. The FEC Group Identifier field is incremented for each new FEC group, and wraps back to zero once the maximum 16-bit unsigned integer value is reached.

For latency-sensitive data, the data header can contain one or more Time-to-Go (TTG) values. The TTG values are set at the source to be equal to the delivery deadline for the payload data. As the payload data traverses each SLIQ connection, its TTG value is decremented by how much time has elapsed in moving the data closer to the destination. Once the TTG value reaches zero, it stays at zero and signifies that the payload data has missed its delivery deadline. The Number of Time-to-Go (TTG) Values field specifies how many TTG values are added at the end of the data header. Each TTG value is 2 bytes long, and its order within the array of TTG values is used to determine which TTG value it is to be interpreted as. All non-FEC data headers and FEC source packet data headers for latency-sensitive payload data contain a single TTG value, that of the payload data. All FEC encoded packet data headers for latency-sensitive payload data contain K TTG values, one for each of the source packet's payload data.

If the data header's *P* flag is set, then it is a persist data packet that causes the receiving endpoint to immediately generate an ACK packet in return. Persist data packets never contain any payload, and the packet sequence number is ignored by the recipient.

If a data packet is the last one that will ever be sent for the stream before the stream is to be closed, then the data header's *F* flag is set to indicate that this is the final data packet (also called a FIN packet) to be sent for the stream. Data packets with the *F* flag set may or may not have payload data present.

The data header format is shown below. Note that the fields marked with an asterisk, as well as the FEC Packet Type field (listed as *T* below), are optional fields.

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+++++	+++++	+++++	+++++
Type U L E M U P F Stream ID Number of TTG			
+++++	+++++	+++++	+++++
CC ID Rexmit Count Payload Length in Bytes			
+++++	+++++	+++++	+++++
Packet Sequence Number			
+++++	+++++	+++++	+++++
Packet Timestamp			

```

+-----+
|          Packet Timestamp Delta          |
+-----+
|          Move Forward Packet Sequence Number*   |
+-----+
|T|U| Index* |NumSrc*|Round* |           Group ID*   |
+-----+
|   Encoded Packet Length*    |       Time-To-Go #1*   |
+-----+
|       Time-To-Go #2*    |       Time-To-Go #3*   |
+-----+
~           ~
~           ~
+-----+
|       Time-To-Go #N*    |           Payload   |
+-----+
|           ~
|           ~
|           ~
|           |
+-----+

```

The data header fields are:

- **Type** (1 byte) - Header type, set to 0x20
- **Flags** (1 byte) - Bit field of flags
 - **U** (1 bit) - Unused, set to zero
 - **L** (1 bit) - Encoded Packet Length Field Present Flag, only used if the **E** Flag is one, otherwise set to zero
 - **E** (1 bit) - FEC Fields Present Flag
 - **M** (1 bit) - Move Forward Present Flag
 - **U** (2 bits) - Unused, set to zero
 - **P** (1 bit) - Persist Flag
 - **F** (1 bit) - FIN Flag
- **Stream ID** (1 byte) - Stream identifier, 1-32, odd for client-initiated streams, even for server-initiated streams
- **Number of TTG Values** (1 byte) - The number of TTG values included in this header, 0-30
- **Congestion Control Identifier** (1 byte) - The index assigned to the congestion control algorithm within the connection that allowed the data to be sent
- **Retransmission Count** (1 byte) - Retransmission count, 0-255 (0 is the original transmission, 1-255 are for retransmissions)
- **Payload Length in Bytes** (2 bytes) - The size of the payload data following this header in bytes
- **Packet Sequence Number** (4 bytes) - Stream data packet sequence number
- **Packet Timestamp** (4 bytes) - The packet send time in microseconds as an unsigned 32-bit integer
- **Packet Timestamp Delta** (4 bytes) - The timestamp difference between the local and remote endpoints in microseconds as an unsigned 32-bit integer

If the **M** Flag is set, then the following move forward information is included next:

- **Move Forward Packet Sequence Number** (4 bytes) - The new next expected data packet sequence number that the receiver must use

If the *E* Flag is set, then the following FEC information is included next:

- **FEC Packet Type** (1 bit) - The type of FEC packet (0 = source data packet, 1 = encoded data packet)
- **Unused** (1 bit) - Unused, set to zero
- **FEC Group Index** (6 bits) - The FEC group index within this FEC group, 0-30
- **Number of FEC Source Packets** (4 bits) - The number of source packets in this FEC group, set to 0 in source data packets, set to 1-31 in encoded data packets
- **Round Number** (4 bits) - The round number that initiated this data packet transmission, 0-15 (0 = retransmission due to RTO or outage, 1 = original transmission round, 2-14 = ARQ retransmission rounds within target delivery delay, 15 = out of rounds for target delivery delay)
- **FEC Group ID** (2 bytes) - The unique identifier assigned to this FEC group

If the *L* Flag is set, then the following FEC encoded packet information is included next:

- **FEC Encoded Packet Length** (2 bytes) - The encoded payload length, used to determining the length of decoded payloads

For each TTG value included (as determined by the Number of TTG Values field), the required fields are:

- **Time-To-Go Value** (2 bytes) - A time-to-go value associated with the payload data, if most significant bit is zero then $TTG = (15_bit_value / 32767.0)$ seconds, if most significant bit is one then $TTG = (1.0 + (15_bit_value / 1000.0))$ seconds

Following the data header is the payload:

- **Payload** (variable length) - Payload data as supplied by the application or encoded using FEC

7.3.1.1.7 ACK Header

The ACK (short for Acknowledgement) header is used to acknowledge the receipt of data packets for a stream within a connection. It may also contain some number of observed packet times for data packets that are being acknowledged (only once per data header received), and some number of selective ACK blocks. Note that all streams within a connection, regardless of their reliability modes (reliable, semi-reliable, or best effort), must generate ACK headers in order for the connection-level congestion control to operate correctly.

The source of the header inserts its own locally-generated timestamp, in microseconds, in the Packet Timestamp field when the packet is sent. Since a timestamp value of zero is to be interpreted as the absence of a timestamp, a timestamp value of one should be sent if the current time

generates a timestamp value that is zero. The Packet Timestamp Delta field is set to the timestamp difference value at the moment the most recent Packet Timestamp field (from any SLIQ header) was received from the remote endpoint. The timestamp difference value is computed as the 32-bit unsigned subtraction: the local timestamp minus the remote timestamp. Since a timestamp difference value of zero is to be interpreted as the absence of a timestamp difference value, a timestamp difference value of one should be sent if the timestamp difference value computation result is zero.

The observed packet times are included to allow the data packet originator to compute a round-trip time (RTT) when the ACK packet is received. Each observation time includes the data packet's sequence number and the data packet's received timestamp, which is increased by the amount of time between the data packet's receipt and the ACK packet's transmission. Upon receipt of the ACK packet, the RTT for a data packet included in the observed packet times is then the ACK packet's receive time minus the observed packet time for the data packet.

Selective ACK Blocks, or simply ACK Blocks, are used to convey detailed data packet acknowledgement information back to the sender. Each ACK Block consists of a 1-bit Type field and a 15-bit ACK Block Offset field. The ACK Block Offset field is an unsigned integer offset from the Next Expected Sequence Number contained in the ACK header. An ACK Block may consist of a single data packet being ACKed (one ACK Block Offset of Type 0 for the individual data packet) or multiple sequential data packets being ACKed (two sequential ACK Block Offsets of Type 1, the first for the first packet in the block, the second for the last packet in the block). If all packets have been received, then the ACK header Next Expected Sequence Number is set to the largest observed data packet sequence number plus one, and no ACK Block Offsets are included in the ACK header. If there are missing data packets, then the ACK header Next Expected Sequence Number is set to the sequence number of the first missing data packet, the first ACK Block must include the ACK for the latest data packet received, and one of the ACK Blocks must include the ACK for the largest observed data packet sequence number. Although the maximum number of ACK Blocks in an ACK header is 31, in order to limit the size of ACK headers while still conveying the most useful ACK Block information, the maximum number targeted by SLIQ is 10 ACK Blocks per ACK header (although 11 is allowed if needed for a Type 1 ACK Block at the end).

The ACK header format is shown below.

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Type Flags (Unused) Stream ID #OPT #ABO			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Next Expected Packet Sequence Number			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Packet Timestamp			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Packet Timestamp Delta			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Observed Packet Sequence Number #1			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Observed Packet Timestamp #1			
+-+-+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Observed Packet Sequence Number #2			

```
+++++-----+-----+-----+-----+-----+-----+-----+  
|           Observed Packet Timestamp #2           |  
+++++-----+-----+-----+-----+-----+-----+-----+  
~           ~  
~           ~  
+++++-----+-----+-----+-----+-----+-----+-----+  
|           Observed Packet Sequence Number #N      |  
+++++-----+-----+-----+-----+-----+-----+-----+  
|           Observed Packet Timestamp #N           |  
+++++-----+-----+-----+-----+-----+-----+-----+  
|T|   ACK Block Offset #1    |T|   ACK Block Offset #2    |  
+++++-----+-----+-----+-----+-----+-----+-----+  
~           ~  
~           ~  
+++++-----+-----+-----+-----+-----+-----+-----+  
|T|   ACK Block Offset #N      |  
+++++-----+-----+-----+-----+
```

The required ACK header fields are:

- **Type** (1 byte) - Header type, set to 0x21
- **Flags** (1 byte) - Bit field of flags, all unused, set to zero
- **Stream ID** (1 byte) - Stream identifier, 1-32, odd for client-initiated streams, even for server-initiated streams
- **Number of Observed Packet Times** (3 bits) - Number of observed packet times contained in this header, 0-7
- **Number of ACK Block Offsets** (5 bits) - Number of ACK block offsets contained in this header, 0-31
- **Next Expected Packet Sequence Number** (4 bytes) - The lowest data packet sequence number not received yet
- **Packet Timestamp** (4 bytes) - The packet send time in microseconds as an unsigned 32-bit integer
- **Packet Timestamp Delta** (4 bytes) - The timestamp difference between the local and remote endpoints in microseconds as an unsigned 32-bit integer

For each Observed Packet Time included (as determined by the Number of Observed Packet Times field), the required fields are:

- **Observed Packet Sequence Number** (4 bytes) - The received data packet stream sequence number
- **Observed Packet Timestamp** (4 bytes) - The received data packet timestamp in microseconds as an unsigned 32-bit integer, adjusted up by the time between the data packet's arrival time and this ACK header's send time

For each ACK Block Offset included (as determined by the Number of ACK Block Offsets field), the required fields are:

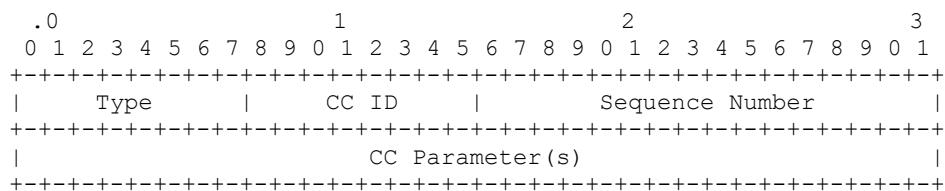
- **Type** (1 bit) - The type of ACK block offset (0 = single packet ACK block, 1 = ACK block start/end for two sequential ACK blocks)
- **ACK Block Offset** (15 bits) - An unsigned integer offset from the Next Expected Sequence Number

7.3.1.1.8 Congestion Control Synchronization Header

The congestion control synchronization header is used to pass connection-level congestion control-specific parameters between peers. A 16-bit sequence number is included to determine message order in case messages are lost or reordered within the network. It is up to the congestion control algorithm referenced by the congestion control identifier to interpret the 32-bits of parameters.

There are no reliability guarantees for this header. The congestion control algorithms are expected to simply utilize the headers that are received. Not all congestion control algorithms use this header.

The congestion control synchronization header format is shown below.



The congestion control synchronization header fields are:

- **Type** (1 byte) - Header type, set to 0x22
- **Congestion Control Identifier** (1 byte) - The index assigned to the congestion control algorithm within the connection to which the information is directed
- **Sequence Number** (2 bytes) - The sequence number for the message, monotonically increasing by one for each message sent
- **Congestion Control Parameter(s)** (2 bytes) - The parameter(s) to be interpreted by the congestion control algorithm

7.3.1.1.9 Received Packet Count Header

The received packet count header is used to pass information about the number of data packets received on the connection in order to accurately compute a connection packet error rate (PER) estimate at the sender. It includes information about the latest data packet received, and the total number of data packets received at that point. The Connection Received Data Packet Count is a total count since the start of the connection, and is allowed to roll over to zero when the count exceeds the maximum unsigned 32-bit value. The Stream ID, Packet Sequence Number, and Retransmission Count fields allow the sender to determine its total number of data packets sent count at the point in time when that data packet was transmitted (which must be stored by the sender, and is not included in this header), and the Connection Received Data Packet Count field supplies the total number of data packets received count at that same point of reference. The difference in these two counts between successive received packet count headers allows for an accurate computation of the one-way PER for the connection.

There are no reliability guarantees for this header. SLIQ simply utilizes the headers that it does receive to update its estimate of the connection PER.

The received packet count header format is shown below.

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1
+++++	+++++	+++++	+++++
Type	Flags (Unused)	Stream ID	Rexmit Count
+++++	+++++	+++++	+++++
	Packet Sequence Number		
+++++	+++++	+++++	+++++
Connection Received Data Packet Count			
+++++	+++++	+++++	+++++

The received packet count header fields are:

- **Type** (1 byte) - Header type, set to 0x23
- **Flags** (1 byte) - Bit field of flags, all unused, set to zero
- **Last Received Data Packet Stream ID** (1 byte) - The stream identifier of the last data packet received, 1-32
- **Last Received Data Packet Retransmission Count** (1 byte) - The retransmission count of the last data packet received, 0-255 (0 is the original transmission, 1-255 are for re-transmissions)
- **Last Received Data Packet Sequence Number** (4 bytes) - The stream data packet sequence number of the last data packet received
- **Connection Received Data Packet Count** (4 bytes) - The total number of data packets received since the start of the connection stored as an unsigned 32-bit number

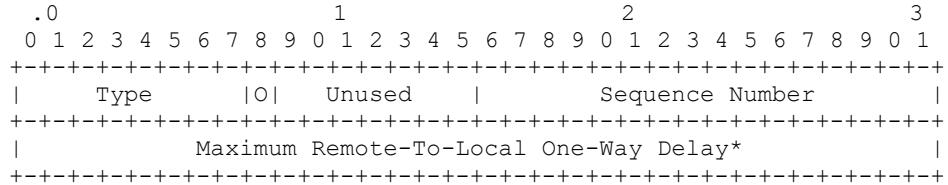
7.3.1.1.10 Connection Measurement Header

The connection measurement header is used to pass connection measurements between endpoints. This header is used when a measurement can only be computed at one endpoint but is needed at the other endpoint.

Currently, there is only one measurement included in this header, the maximum remote-to-local one-way delay. However, the header has been designed to be easily extended as needed to include other connection measurements in the future, with each measurement type optional. The Flags field is to have one bit for each supported measurement type, ordered from most significant bit to least significant bit. When the bit for the measurement is set in the Flags field, then the measurement is included in the header. When the bit for the measurement is not set in the Flags field, then the measurement is not included in the header. The order of the measurement types in the Flags field matches the order of the optional measurement fields later in the header. Each measurement field has its own field length, and may be either a fixed length or a variable length.

A 16-bit sequence number is included to determine message order in case messages are lost or reordered within the network. There are no reliability guarantees for this header.

The connection measurement header format is shown below. Note that the fields marked with an asterisk are optional fields.



The connection measurement header fields are:

- **Type** (1 byte) - Header type, set to 0x24
- **Flags** (1 byte) - Bit field of flags
 - **O** (1 bit) - Maximum Remote-To-Local One-Way Delay Present Flag
 - **U** (7 bits) - Unused, set to zero

If the *O* Flag is set, then the following measurement information is included next:

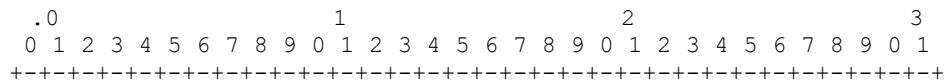
- **Maximum Remote-To-Local One-Way Delay** (4 bytes) - The maximum remote-to-local one-way delay estimate in microseconds

7.3.1.1.11 Congestion Control Packet Train Header

The congestion control packet train header is optionally used by congestion control algorithms to exchange packet trains. These packet trains are typically used during startup to characterize the channel, resulting in estimates of the channel capacity and RTT that can be used to initialize control parameters within the algorithm. Outside of the packet timestamp and timestamp delta fields, it is up to the congestion control algorithm referenced by the congestion control identifier to interpret how these packets are transmitted and how they are processed upon receipt. Not all congestion control algorithms use this header.

The source of the header inserts its own locally-generated timestamp, in microseconds, in the Packet Timestamp field when the packet is sent. Since a timestamp value of zero is to be interpreted as the absence of a timestamp, a timestamp value of one should be sent if the current time generates a timestamp value that is zero. The Packet Timestamp Delta field is set to the timestamp difference value at the moment the most recent Packet Timestamp field (from any SLIQ header) was received from the remote endpoint. The timestamp difference value is computed as the 32-bit unsigned subtraction: the local timestamp minus the remote timestamp. Since a timestamp difference value of zero is to be interpreted as the absence of a timestamp difference value, a timestamp difference value of one should be sent if the timestamp difference value computation result is zero.

The congestion control packet train header format is shown below.



Type	CC ID	PT Pkt Type	PT Seq Num
		Packet Pair Inter-Receive Time	
		Packet Timestamp	
		Packet Timestamp Delta	
		Payload	
~			~
~			~
~			~

The congestion control packet train header fields are:

- **Type** (1 byte) - Header type, set to 0x28
- **Congestion Control Identifier** (1 byte) - The index assigned to the congestion control algorithm within the connection to which the information is directed
- **Packet Train Packet Type** (1 byte) - The packet type assigned to the packet by the congestion control algorithm
- **Packet Train Sequence Number** (1 byte) - The sequence number assigned to the packet by the congestion control algorithm
- **Packet Pair Inter-Receive Time** (4 bytes) - The packet pair inter-receive time in microseconds, computed by the congestion control algorithm
- **Packet Timestamp** (4 bytes) - The packet send time in microseconds as an unsigned 32-bit integer
- **Packet Timestamp Delta** (4 bytes) - The timestamp difference between the local and remote endpoints in microseconds as an unsigned 32-bit integer
- **Payload** (variable) - The packet padding appended to achieve the desired packet size, sized by the congestion control algorithm

7.3.1.2 Header Concatenation Rules

In order to minimize the number of SLIQ packets sent, certain SLIQ headers are allowed to be concatenated into a single UDP/IP packet for transmission to the peer. The rules for header concatenation are:

- Only SLIQ headers for a single connection can be concatenated together into a single UDP/IP packet.
- Only the data, ACK, congestion control synchronization, received packet count, and connection measurement headers may be concatenated.
- The SLIQ headers are concatenated with no padding between the headers.
- SLIQ headers may be concatenated until the total length of the SLIQ headers and payload reaches 1472 bytes. Once this limit would be exceeded, another UDP/IP packet must be utilized for additional SLIQ headers.
- All ACK, congestion control synchronization, received packet count, and connection measurement headers must come before any data header.
- Only one data header may be included in each UDP/IP packet, and if included, it must occur last. This is to prevent any SLIQ headers from being placed after the payload.

The use of header concatenation has proven useful in reducing the number of UDP/IP packets sent, especially for SLIQ ACKs.

7.3.1.3 State Machines

The following subsections capture the SLIQ protocol in a series of state machine diagrams. In each diagram:

- Ellipses are states,
- Rectangles are processing steps,
- Diamonds are decision points, and
- Arrows are events.

The text for an event always appears on the right side of the arrow. For some diagrams, the event text contains processing steps between brackets (i.e., within "[" and "]" characters) after the event string in order to keep the diagrams as small as possible. Events without any event text label are followed immediately.

7.3.1.3.1 Connection Establishment

SLIQ connection establishment may occur in one of two ways. This first is the normal TCP-like client/server approach. With this approach, the server endpoint starts first and listens on a well-known UDP port number, while the client endpoint starts second and connects to the server. The second is the direct connection approach, which eliminates the need for the server endpoint to perform a listen on a well-known UDP port number. With this approach, the client and server are both initialized with the IP addresses and UDP port numbers of both endpoints. The benefit to the direct connection approach is that the client and server endpoints may use the same UDP port number which simplifies debugging, and the server is not open to unintended connections.

7.3.1.3.1.1 TCP-Like Client/Server Connection Establishment

The state machine for the TCP-like client/server connection establishment procedure is shown below.

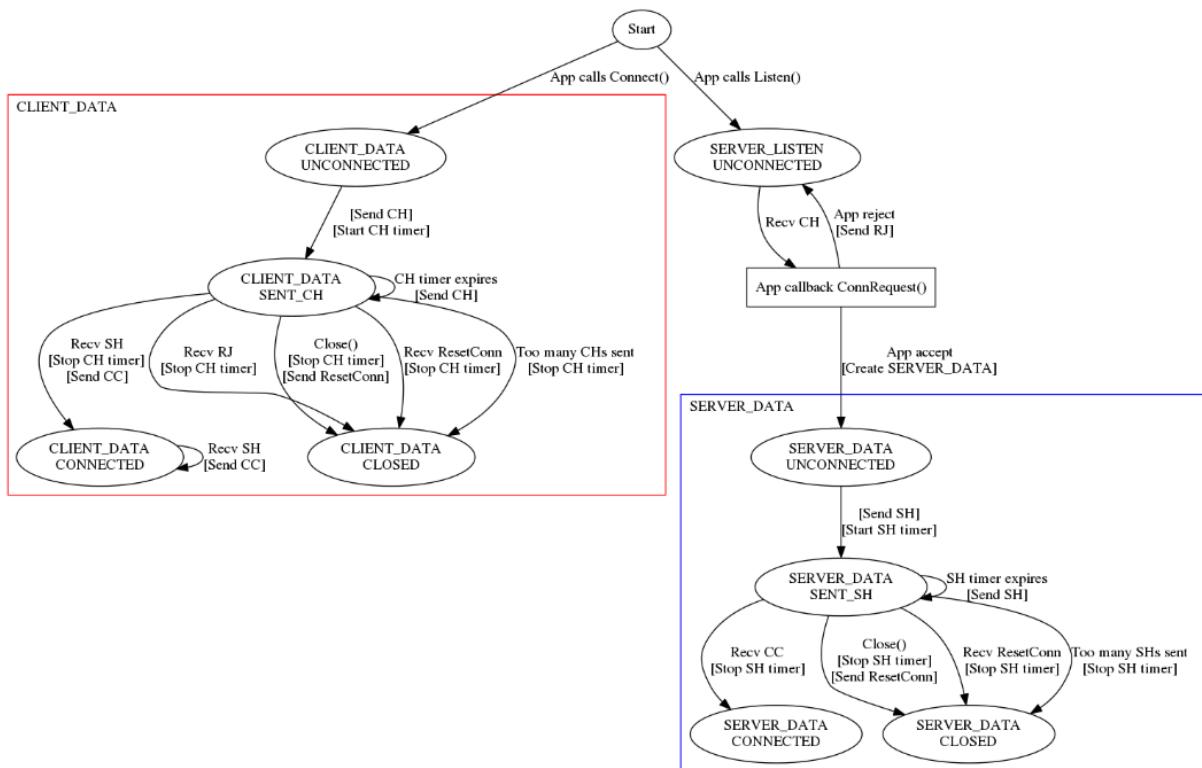
The server is on the right side of the figure. It first opens a server listen endpoint that waits for a connection request from the client. Once the request is received and the server application accepts it, a server data endpoint is created. The server data endpoint logic is within the blue box. The client is on the left side of the figure. It only uses a single client data endpoint, which is within the red box.

All packets sent and received are SLIQ connection handshake packets, with this packet's Message Tag field set to the specified strings. Valid tags include CH (client hello), SH (server hello), CC (client confirmation), and RJ (rejected by server application).

The UDP port numbers used for the connection change as follows. The server listen socket is bound to the well-known port number (which we'll call P_{sl} , short for "Port - server listen"), while the client data socket is bound to an ephemeral port number (which we'll call P_c , short for "Port -

client"). The CH packet is sent from Pc to Psl . When the server receives this packet on the server listen socket and the server application accepts the connection request, the server data socket is created and is bound to a new ephemeral port number (called Psd , short for "Port - server data"). The SH packet is then sent using the server data socket from Psd to Pc . When the client receives the SH packet, it stores the source port number Psd as the port number for the new connection. The client then sends the CC packet from Pc to Psd , and the server receives this on the server data socket in order to complete the connection setup. All other SLIQ packets are exchanged using the client and server data sockets using the Pc and Psd port numbers. The server listen socket may remain open in order to create additional connections.

The CH and SH timers currently use a 0.333 second duration. After 32 CH or SH packets are sent without a response, the connection establishment is considered to have failed, the connection is transitioned to the CLOSED state, and an error is returned to the application. Stream creation and data transfers within streams occur with the client side in the CLIENT_DATA CONNECTED state and the server side in the SERVER_DATA CONNECTED state.



7.3.1.3.1.2 Direction Connection Establishment

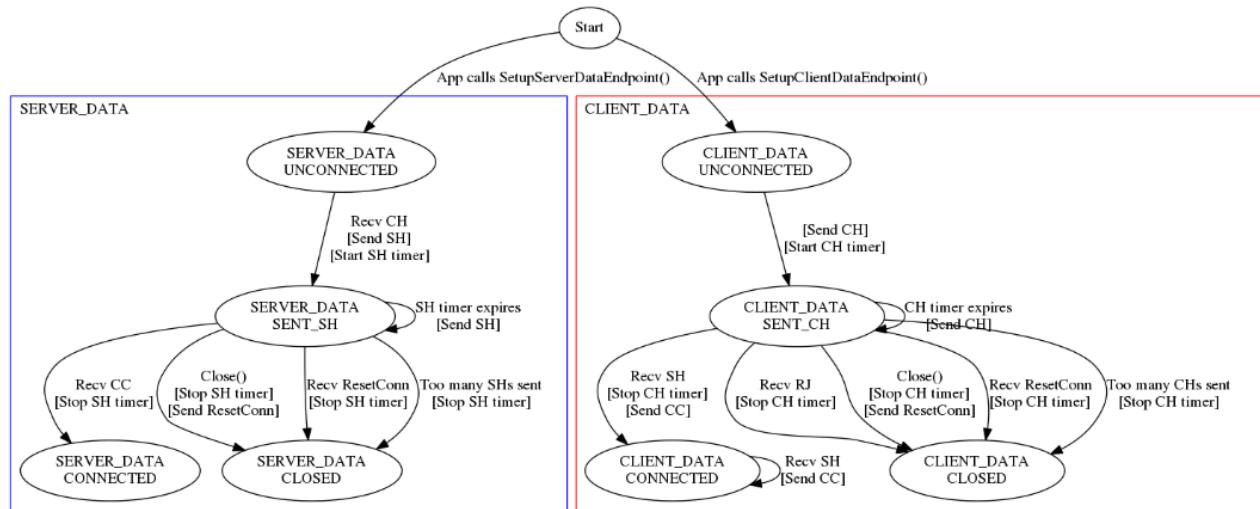
The state machine for the direct connection establishment procedure is shown below.

The server is on the left side of the figure. It opens a server data endpoint right away that waits for a connection request only from the client specified in the SetupServerDirectEndpoint() call. This logic is all within the server data endpoint logic in the blue box. The client is on the right side of the figure. It only uses a single client data endpoint, which is within the red box.

All packets sent and received are SLIQ connection handshake packets, with this packet's Message Tag field set to the specified strings. Valid tags include CH (client hello), SH (server hello), CC (client confirmation), and RJ (rejected).

The UDP port numbers used for the connection must be specified for both endpoints in the SetupServerDataEndpoint() and SetupClientDataEndpoint() calls. All SLIQ packets are exchanged using these UDP port numbers for the duration of the connection.

The CH and SH timers currently use a 0.333 second duration. After 32 CH or SH packets are sent without a response, the connection establishment is considered to have failed, the connection is transitioned to the CLOSED state, and an error is returned to the application. Stream creation and data transfers within streams occur with the client side in the CLIENT_DATA CONNECTED state and the server side in the SERVER_DATA CONNECTED state.



7.3.1.3.2 Connection Termination

SLIQ connection termination may be initiated by either end of the connection. The state machine for terminating the connection is shown below.

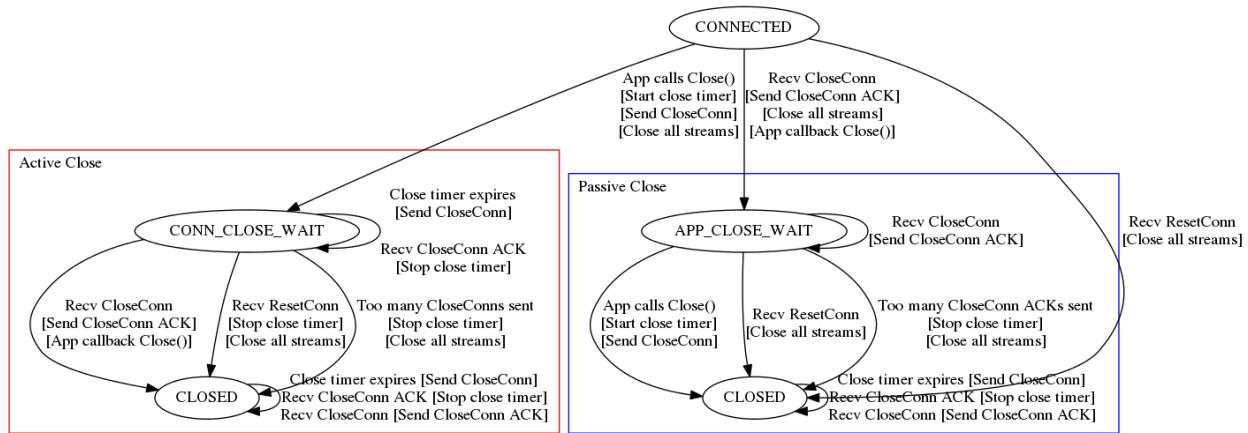
The end that initiates the connection termination performs the Active Close logic. The other end performs the Passive Close logic.

The SLIQ close connection and reset connection packets are used to implement the connection termination. The close connection packet contains an ACK flag that allows the packet to serve as an acknowledgement. After receiving a close connection packet with the ACK flag not set, the packet is sent back to the originator with the ACK flag set in order to confirm receipt.

Note that it is possible for both ends of the connection to initiate an Active Close at about the same time. In this case, both ends will use the Active Close logic, the close connection packets will cross in the network, both ends will send the necessary close connection ACK packets, and both ends will move to the CLOSED state.

If the application issues a close command when there are still streams that are open, all streams are terminated immediately. Similarly, upon receipt of a close connection message, any open streams are terminated immediately.

The Close Timer currently uses a 0.333 second duration. After 32 close connection or close connection ACK packets are sent without a response, the connection is forced into the CLOSED state.



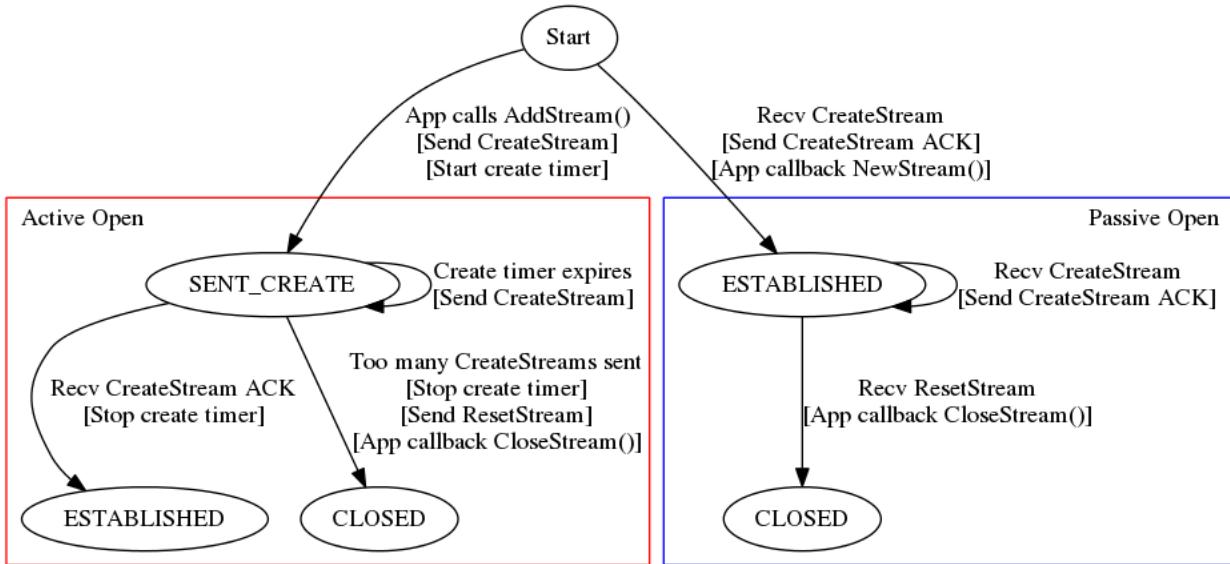
7.3.1.3.3 Stream Establishment

Once a connection has been established successfully, streams may be created. Either end may initiate the creation of a stream. The state machine for stream establishment is shown below.

The end that initiates the creation of the stream is the Active Open side. The other end is the Passive Open side. Each stream is identified by a unique integer number between 1 and 32 (inclusive). In order to prevent stream creation collisions, the SLIQ client can only create streams with odd numbers, and the SLIQ server can only create streams with even numbers.

The SLIQ create stream and reset stream packets are used to create streams. The create stream packet contains an ACK flag that allows the packet to serve as an acknowledgement. After receiving a create stream packet with the ACK flag not set, the packet is sent back to the originator with the ACK flag set in order to confirm receipt. Reset stream packets cause the stream to immediately become closed.

The Create Timers currently use a 0.333 second duration. After 32 create stream packets are sent without a response, the stream establishment is considered to have failed, the stream is transitioned to the CLOSED state, and an error is returned to the application. Stream data transfers may begin once both ends are in the ESTABLISHED state.



7.3.1.3.4 Stream Termination

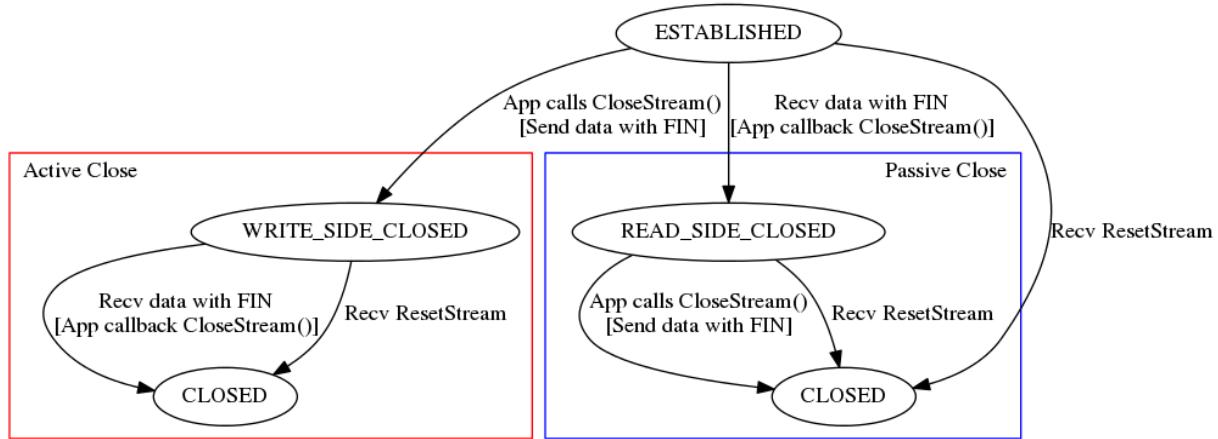
SLIQ stream termination may be initiated by either end of the stream. The state machine for terminating the stream is shown below.

The end that initiates the termination of the stream is the Active Close side. The other end is the Passive Close side. Either side may initiate the termination.

The SLIQ data and reset stream packets are used to terminate a stream. The data packet contains a FIN flag (labeled as the *F* flag in the data packet format) that allows marking that data packet sequence number as the final one in that direction. Reset stream packets cause the stream to immediately become closed, even if not all of the data has been received yet.

In order to guarantee that all of the data is transferred and delivered during termination, the applications should close all of the streams first. Once all of the streams are properly terminated, then the applications can close the connection (typically, the client initiates the connection close, but this is not required). If the application on either end initiates a connection close without first closing all of the streams, then it is possible that not all of the stream data will be delivered before the connection is closed.

In all of the possible reliability modes (reliable, semi-reliable, and best effort), data packets with the FIN flag must always be delivered and ACKed. SLIQ will attempt to deliver the data packet with the FIN flag set forever, retransmitting them as necessary, in an attempt to guarantee that all of the data it was responsible for delivering was transferred and that the applications know that the stream has been closed. If this takes too long, then it is up to the application to give up at some point and simply close the connection, which will immediately terminate all of the streams.



7.3.1.3.5 Data Transfer

Data is only transferred within an established stream, and each stream supports bi-directional data transfers. Data transfer involves a number of terms that are used in the following descriptions.

- **Sequence Number:** SLIQ data packets include a sequence number, which are used to determine the order of the data included in them. Sequence numbers are assigned at the stream level. SLIQ sequence numbers are monotonically increasing unsigned 32-bit integers that allow rollover. Each data transfer direction of a stream has its own sequence numbers, with the starting sequence numbers selected randomly.
- **Transmit Queue:** Each stream has a transmit queue that holds payload data from the application that has not been sent to the peer yet. Data is placed in the transmit queue when either there is already data in the transmit queue waiting to be sent, or the transmit queue is empty but flow control or congestion control determines that data cannot be sent immediately. Application payload data that can be sent immediately by SLIQ is not added to the transmit queue. The payload data in the transmit queue will be dequeued, placed in a data packet, and sent by SLIQ at a later time. Once a stream's transmit queue is full, any **Send()** call from the application on that stream will fail.
- **Send Window:** Each stream has a send window that holds data packets that have been sent to the peer in sequence number order (low to high). Once application payload data is placed in a data packet, assigned the next sequence number, and sent to the peer, the data packet is added to the high end of the send window as an unACKed packet. Once the peer acknowledges the data packet, it is marked as an ACKed packet in the send window. Data packets at the low end of the send window that have been ACKed are removed from the send window and released. The size of the send window (in packets) is controlled by the peer's available receive window size.
- **Receive Window:** Each stream has a receive window that holds data packets that have been received from the peer in sequence number order (low to high). Each data packet received from the peer is placed in the receive window in its correct place based on its sequence number. As payload data can be delivered to the application (which depends on the configured delivery mode for the stream), the data is removed from the data packet(s),

the data is delivered to the application, and the data packet(s) is/are removed from the receive window and released. The receipt of data packets is conveyed to the peer in ACK packets.

- **Flow Control:** Flow control refers to the sender honoring the receiver's advertised receive window size. Since both the send and receive windows have fixed sizes, and these two sizes are always the same, flow control is easy to implement. Flow control occurs at the stream level.
- **Congestion Control:** Congestion control refers to the sender controlling its send rate to avoid congestion in the network. Congestion control occurs at the connection level, and not at the level of individual streams. Each connection can use one or more congestion control algorithms in parallel.

When a SLIQ stream is set to semi-reliable ARQ+FEC mode, a target packet delivery delay (specified as either a time value in seconds or a number of transmission rounds) and a target packet receive probability (the probability that a datagram will be received by the target packet delivery delay) are specified. This mode uses the Latency-Constrained Adaptive Error Control (AEC) algorithm to decide which error control methods to use given the current network conditions in order to meet the specified datagram delivery deadline (see the FEC Data Send state machine section for details). If the current network packet error rate (PER) is large enough that the use of ARQ by itself is insufficient to meet the delivery deadline and packet receive probability, then either FEC is used along with ARQ or FEC is used by itself. The use of FEC involves the sending of redundant data along with the application data to be transferred, and using the received application and redundant data to regenerate the missing application data. There are a number of FEC terms and limits that are used in the following descriptions:

- **FEC Source Data Packet:** A data packet containing payload data from the sending application. The payload is from the "source", which is the application.
- **FEC Encoded Data Packet:** A data packet containing redundant payload data that is an encoding of one or more FEC source data packet payloads. The payload is "encoded" data that cannot be delivered directly to the application, but can be used with other received FEC source and/or encoded data packets to regenerate missing FEC source data packets.
- **FEC Group ID:** A unique integer identifier for a group of FEC source and encoded data packets, where the FEC encoded data packets in the group were generated from the FEC source data packets in the group. All of these packets will have the same unique FEC group ID when they are transmitted at the sending SLIQ endpoint. The unique FEC group ID is then used at the receiving SLIQ endpoint to make sure that the FEC source and encoded data packets that go together are processed together. Note that the SLIQ assigned FEC group IDs are incremented by one for each FEC group created, and are allowed to wrap around back to zero and be reused. Since the 16-bit FEC group ID field used by SLIQ supports 65,536 values, the SLIQ send and receive windows can only support 32,768 packets at a time, and there must be at least one data packet in each FEC group, there is no chance of an FEC group ID being reused within the send or receive windows.
- **FEC Group Index:** A zero-based integer index for FEC source and encoded data packets within the same FEC group that is used to order the packets at the receiving SLIQ endpoint. For each FEC group, the FEC source data packets are always transmitted before

the FEC encoded data packets, as the encoded packets cannot be created until all of the source packets are available. The FEC code used is an (n,k) erasure code, where n is the number of FEC source and encoded data packets in the FEC group, and k is the number of FEC source data packets in the FEC group. We introduce a new variable, m , that is the number of FEC encoded data packets, where $m=(n-k)$. Note that n must always be greater than or equal to k . For a given FEC group, the FEC source data packets will be assigned FEC group indices of 0 through $(k-1)$, and if there are FEC encoded data packets, then they will be assigned FEC group indices of k through $(n-1)$. The current FEC implementation in SLIQ limits k to values between 1 and 10 (inclusive), and n to values up to and including 31.

- **Round Number:** The transmission or retransmission round number for the packets in an FEC group. The original transmission of the FEC source and encoded data packets for an FEC group occurs in round 1. The ARQ retransmissions that are still within the target packet delivery deadline occur in rounds 2 through 14. Once the target packet delivery deadline is passed, the round is set to 15, which is to be interpreted as "out of rounds." A FEC packet retransmission due to an RTO or an outage condition will have the round set to 0, which is to be interpreted as "no round information."
- **Target Number of Rounds:** The maximum number of rounds that can be used for an FEC group and still meet the target packet delivery delay. Represented as a variable N in this document, where N must be between 1 and 14 (inclusive).
- **FEC Group End of Round (EOR):** Each round for an FEC group begins when the first FEC data packet is sent in the round, and ends when all of the FEC data packets for the round have been transmitted and enough time has elapsed that all of the ACK packets for the round's FEC data packets will have been received. The EOR is when this end time has been reached. When the EOR occurs, a determination can be made about what FEC data packets are still missing, and the next round for the FEC group (if there is one) can be planned.
- **Original FEC Encoded Data Packet:** An FEC encoded data packet that is generated in round 1 and has not been sent yet. Treated as if it were an original transmission of application data, as opposed to as if it were a retransmission of application data. Placed in a separate data packet queue called the original FEC encoded data packet queue until it is sent. Once sent, it is moved to the send window, and it becomes a normal FEC encoded data packet that can then be retransmitted later, when it will be treated as if it were a retransmission of application data.
- **Additional FEC Encoded Data Packet:** An FEC encoded data packet that is generated in round 2 or later and has not been sent yet. Treated as if it were a retransmission of application data, as opposed to as if it were an original transmission of application data. Placed in a separate data packet queue called the additional FEC encoded data packet queue until it is sent. Once sent, it is moved to the send window, and it becomes a normal FEC encoded data packet that can be retransmitted later, when it will also be treated as if it were a retransmission of application data.

The following series of state machines are all part of the data transfer process.

7.3.1.3.5.1 Application Data Send

The state machine shown below is the application data send state machine, which is used for moving data from the application into SLIQ for a specific stream. The SLIQ data packets are used for sending application payload data and FEC redundant payload data.

SLIQ first tries to create room in the transmit queue by attempting to send data (encapsulated as SLIQ data packets) from the transmit queue over the wire (see the call to Delayed Data Send in the diagram below). The data packet retransmit count field is set to zero for the initial data packet transmission. Retransmissions of the data packet have this field set to the correct retransmission number.

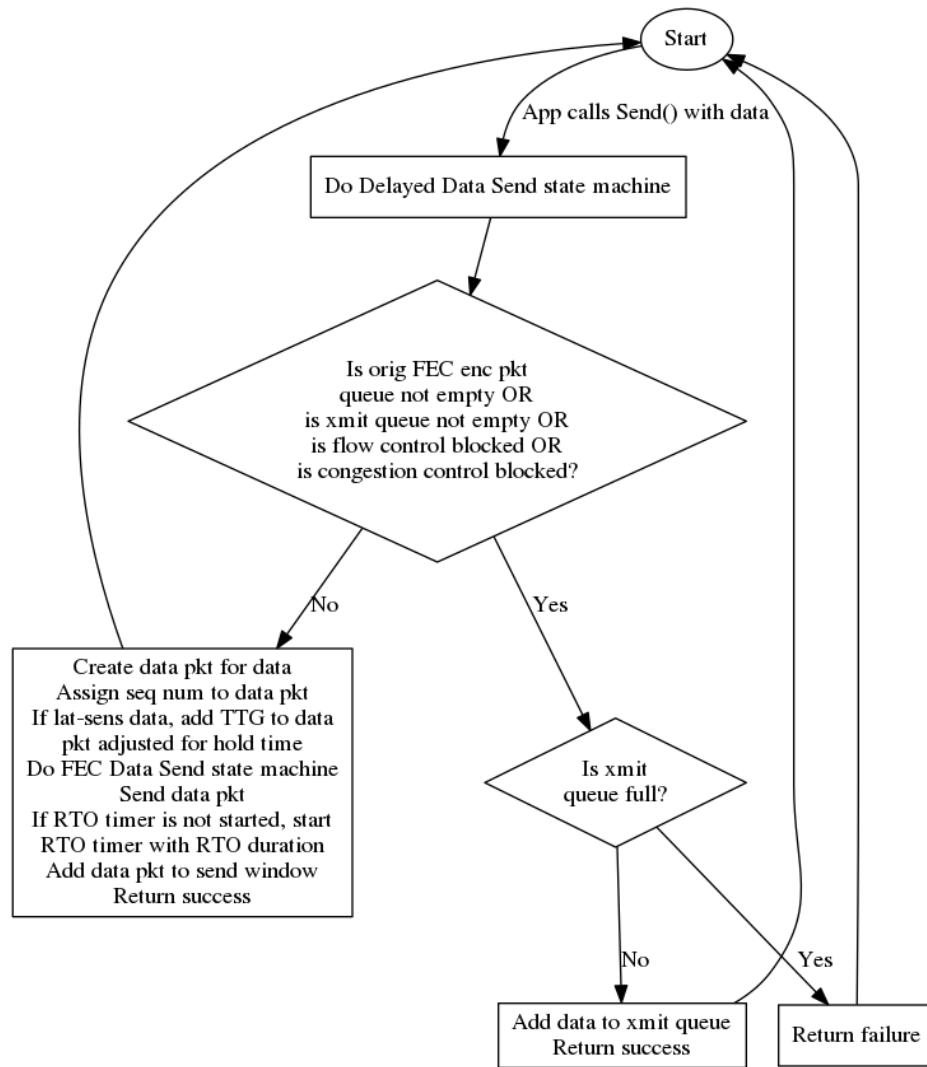
If the application data to be sent is latency-sensitive, then a single Time-to-Go (TTG) value is added as part of the data packet header. The TTG value is equal to the current amount of time left to deliver the packet, and is adjusted down by the hold time (the amount of time that the data has been queued before being transmitted). The TTG value will be adjusted down by the network delay when received by the receiving SLIQ endpoint. Once the TTG value reaches zero, it stays at zero and signifies that the data has missed its delivery deadline.

The retransmission timeout (RTO) timer is similar to the RTO timer used in TCP. It is used to detect when the receiver has not responded within the expected amount of time given the observed network delays. SLIQ also uses this timer to detect outages, as described later in the *Outage Mode* section. It is implemented at the connection level. The RTO timer duration defaults to 1 second. Once RTT estimates are available, it is set to the smoothed RTT plus 4 times the RTT mean deviation. The duration is always bounded to be between 200 milliseconds and 1 second.

In order to avoid setting and resetting an actual RTO timer frequently, it is implemented using a base RTO timer and an RTO time. The base RTO timer goes off every 100 milliseconds constantly. The RTO time holds the current RTO expiration time. This allows the RTO timer to be set by simply updating the RTO time. When the base RTO timer goes off and the current time is past the RTO time, then the RTO timer is considered to have expired and the RTO callback functionality is called. Note that this means that the actual RTO timer callback will frequently occur later than the RTO time using this approach, but this is an acceptable condition as the RTO timer is not normally needed during steady-state operation of SLIQ.

It is possible for the congestion control algorithm to request a more accurate RTO timer. This is needed when the congestion control algorithm relies on the RTO timer for steady state operation due to the current network conditions. An example of when this might be needed is when the congestion control algorithm implements a congestion window and is designed to operate with high levels on non-congestion packet loss. When the congestion window size is small, it becomes very likely that the necessary ACK packets do not arrive to maintain normal steady-state operation, and the RTO timer is needed to retransmit packets at certain times. Before the base RTO timer is set or the RTO time is updated, the congestion control algorithm is queried if fast RTOs are needed or not. If fast RTOs are required, then the base RTO timer is set with a duration of one half of the smoothed RTT (limited to be no less than 1 millisecond) and the RTO

time is set to the smoothed RTT plus 5 times the RTT mean deviation. If fast RTOs are not required, then the base RTO timer is set to go off in 100 milliseconds and the RTO time is set to the requested time period.



7.3.1.3.5.2 Delayed Data Send

The state machine shown below is the delayed data send state machine, which operates at the connection level. It is called by other state machines, or the SLIQ protocol, when queued data for a connection may be sent.

The state machine includes three nested loops. The outermost of the three loops controls two passes of the innermost two loops: one to send new FEC encoded data packets and retransmission data packets, and one to send new data packets. The middle loop iterates over all of the possible priority levels, from highest priority to lowest priority, allowing higher priority streams in the connection to send before lower priority streams in the connection. The innermost loop iterates over all of the streams in the connection with the middle loop's priority level in a round-

robin fashion, giving each stream at that priority level an opportunity to send a data packet in a fair manner.

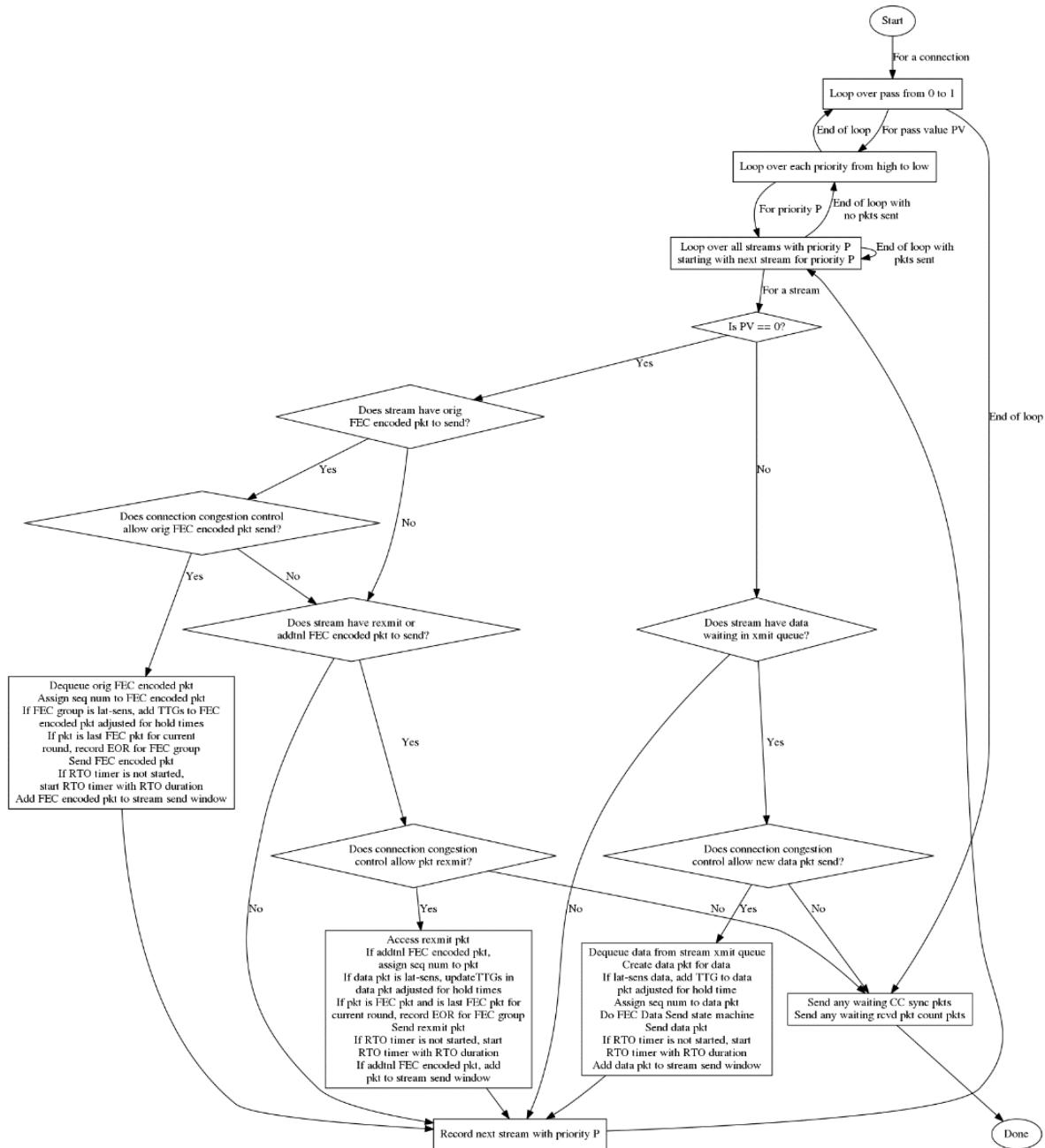
Additional details of these three loops are as follows.

- The outermost loop controls which pass is occurring within the two inner loops. The first pass attempts to send as many original FEC encoded data packets or retransmission data packets (including additional FEC encoded data packets) as possible for all of the streams in the connection. The second pass attempts to send as many new data packets as possible for all of the streams in the connection. The two passes correspond to pass values PV of 0 and 1, respectively.
- The second pass of the outermost loop only occurs if there are no more original FEC encoded data packets or retransmission data packets (including additional FEC encoded data packets) to send by all of the streams in the connection during the first pass.
- The middle loop walks over each priority level used by the streams within the connection, from highest priority to lowest priority. The priority for this loop execution is called P .
- Within this middle loop is an innermost loop that repeatedly walks over all of the streams with priority level P in the connection. This inner loop records the next stream to be processed after each iteration. The next time this loop is called for the connection and priority level P , it starts the loop with the recorded next stream.
- For each stream in the innermost loop, an attempt to send one data packet from the stream's original FEC encoded data packet queue, retransmission list, or transmit queue is made.
- After all of the streams with priority level P have attempted to send one data packet, a decision to repeat the innermost loop is made. If at least one of the streams sent a data packet, then the innermost loop walks through all of the streams again. Otherwise, the loop ends and control returns to the middle loop. This implements a round-robin scheduler at each priority level, with higher priority streams having precedence over lower priority streams.
- The middle loop completes when the lowest priority level has been processed.
- If the connection becomes congestion control blocked at any point, then control breaks out of all of the three nested loops.
- The final checks before the state machine processing ends are for congestion control synchronization and received packet count information that is waiting to be sent. If either of these types of information are waiting to be sent, then they are sent immediately.

For any latency-sensitive data packets that are sent, the data packet header TTG value(s) must be set before transmission. For latency-sensitive non-FEC data packets, the data packet header includes a single TTG value, which is adjusted down by the hold time (the amount of time spent queued locally). For latency-sensitive FEC source data packets, the data packet header includes a single TTG value, which is adjusted down by the hold time. For latency-sensitive FEC encoded data packets, the data packet header includes k TTG values, where k is the number of FEC source data packets in the FEC group, each adjusted down by the individual FEC source data packet hold times. These TTG values are placed in the data packet header in FEC group index order, from 0 to $(k-1)$. This provides a separate TTG value that can be used for each FEC source data

packet that is regenerated due to the receipt of this FEC encoded data packet. Note that once a TTG value reaches zero, it stays at zero and signifies that the data has missed its delivery deadline.

When fast RTOs are not required, the RTO timer duration defaults to 1 second. Once RTT estimates are available, it is set to the smoothed RTT plus 4 times the RTT mean deviation. The duration is always bounded to be between 200 milliseconds and 1 second. The settings when fast RTOs are required are detailed in the Application Data Send section above.



7.3.1.3.5.3 FEC Data Send

The state machine shown below is the FEC data packet send state machine, which operates at the stream level. It is called by other state machines just before an original data packet is sent, and implements the Latency-Constrained Active Error Control (AEC) algorithm. If FEC is needed to meet the specified datagram delivery deadline, then the algorithm adds the necessary FEC information to the data packet header and updates the FEC group state as required.

Note the following variables used in this state machine and the following descriptions:

- **k** - The number of FEC source data packets to send in the FEC group. Stays constant for all rounds. Must be between 1 and 10 (inclusive).
- **m** - The number of FEC encoded data packets to send in the FEC group. It is possible for this to increase as rounds go by.
- **N** - The target number of rounds for the FEC group. This is the maximum number of rounds that still allows the target packet delivery delay to be met.
- **sr** - The number of received (ACKed) FEC source data packets in the FEC group.
- **er** - The number of received (ACKed) FEC encoded data packets in the FEC group.
- **dyn_k** - The dynamically-selected *k* value to be used in future FEC groups for the stream. Tuned such that the maximum number of FEC data packets (*k+m*) is used in each FEC group, yet all of the FEC data packets can be transmitted before an ACK is received for the first FEC data packet.
- **tgt_N** - The target number of rounds for the stream as specified by the application. Only available when the target packet delivery delay is specified as a number of rounds.
- **arq_N** - The number of rounds required to meet the target packet receive probability for the stream using pure ARQ given the current PER estimate. Computed as the minimum *arq_N* value that satisfies the inequality: $PER^{arq_N} \leq (1 - P_{recv})$.
- **PER** - The current packet error rate (PER) estimate for the connection. Must be between 0.0 and 1.0 (inclusive).
- **MaxRTT** - The maximum round-trip-time estimate for the connection.
- **MaxLtrOwd** - The maximum local-to-remote one-way delay estimate for the connection.
- **MaxPst** - The maximum packet serialization time estimate for the connection. This is computed by taking the maximum FEC data packet size in bits and dividing it by the current connection channel capacity estimate in bits/second.

Each round for an FEC group begins when the first FEC data packet is sent in the round, and ends when all of the FEC data packets scheduled for the round have been transmitted and enough time has elapsed that all of the ACK packets for those FEC data packets will have been received. Note that the end of a round cannot depend on the receipt of a specific ACK packet, because either the last FEC data packet might be lost or the ACK packet for the last FEC data packet might be lost. Thus, the end of round is determined by storing the data packet timestamp recorded in the last FEC data packet sent in the round as the EOR timestamp, and comparing each received ACK packet's observed packet timestamp against this EOR timestamp. As soon as an ACK packet's observed packet timestamp is greater than or equal to the EOR timestamp, then the FEC group's EOR has been reached, and the FEC group's next round (if there is one) can begin. Once

all of the rounds for an FEC group have passed, the FEC group is considered to be "out of rounds." Once out of rounds, the target packet delivery delay can no longer be met, and pure ARQ is used to retransmit just the FEC source data packets (not the FEC encoded data packets) up to the maximum retransmission limit set for the stream.

The optimal value of k is stored in dyn_k . It is discovered at run-time by observing the k value used for each FEC group in the stream and when the ACK packets begin arriving for those groups. It is tuned such that the maximum number of FEC data packets ($k+m$) are sent before an ACK is received for the first FEC data packet. The dyn_k state is initialized to the maximum k value (10) and is updated using the following rules:

- When 16 FEC groups have used the current value of dyn_k for their k value, and each of those 16 FEC groups had their first ACK arrive after all of the FEC data packets for the group have been sent, then dyn_k is increased by one (limited to a maximum value of 10).
- When an FEC group has used the current value of dyn_k as its k value and has its first ACK arrive before all of the FEC data packets for the group have been sent, then dyn_k is immediately decreased by one (limited to a minimum value of 1).

The state machine utilizes two multidimensional tables, an FEC efficiency table and an FEC lookup table. These tables have the following layouts and uses:

- **The FEC Efficiency Table** - This table stores the expected packet efficiency at the receiver for an FEC group (computed as useful FEC packets received for the FEC group divided by all packets received for the FEC group) given the target packet receive probability (P_{recv}), the current PER (PER), the target number of rounds for the group (N), and the number of FEC source data packets in the group (k). Note that the FEC efficiency table is only defined for a limited set of Precv and PER values, and the actual values for these dimensions are mapped to the next higher value when there is not an exact match. The state machine uses the FEC efficiency table to iterate over all possible N and k values to find the FEC group settings that provide the highest overall efficiency.
- **The FEC Lookup Table** - This table stores the degrees of freedom (i.e., the number of FEC data packets) to send by an FEC group at each round. The table is split into two complete tables at the top level. One is the midgame FEC lookup table, which is used for all rounds except the last round. The other is the endgame FEC lookup table, which is used for just the last round. Each of these tables stores the number of FEC data packets to send given the target number of rounds for the group (N), the current PER (PER), the number of FEC source data packets in the group (k), the number of received FEC source data packets in the group (sr), and the number of received FEC encoded data packets in the group (er). Note that the FEC lookup table is only defined for a limited set of PER values, and the actual PER value is mapped to the next higher value when there is not an exact match.

The state machine requires determining if different error control alternatives are able to meet the target packet delivery delay. These decisions are made as follows:

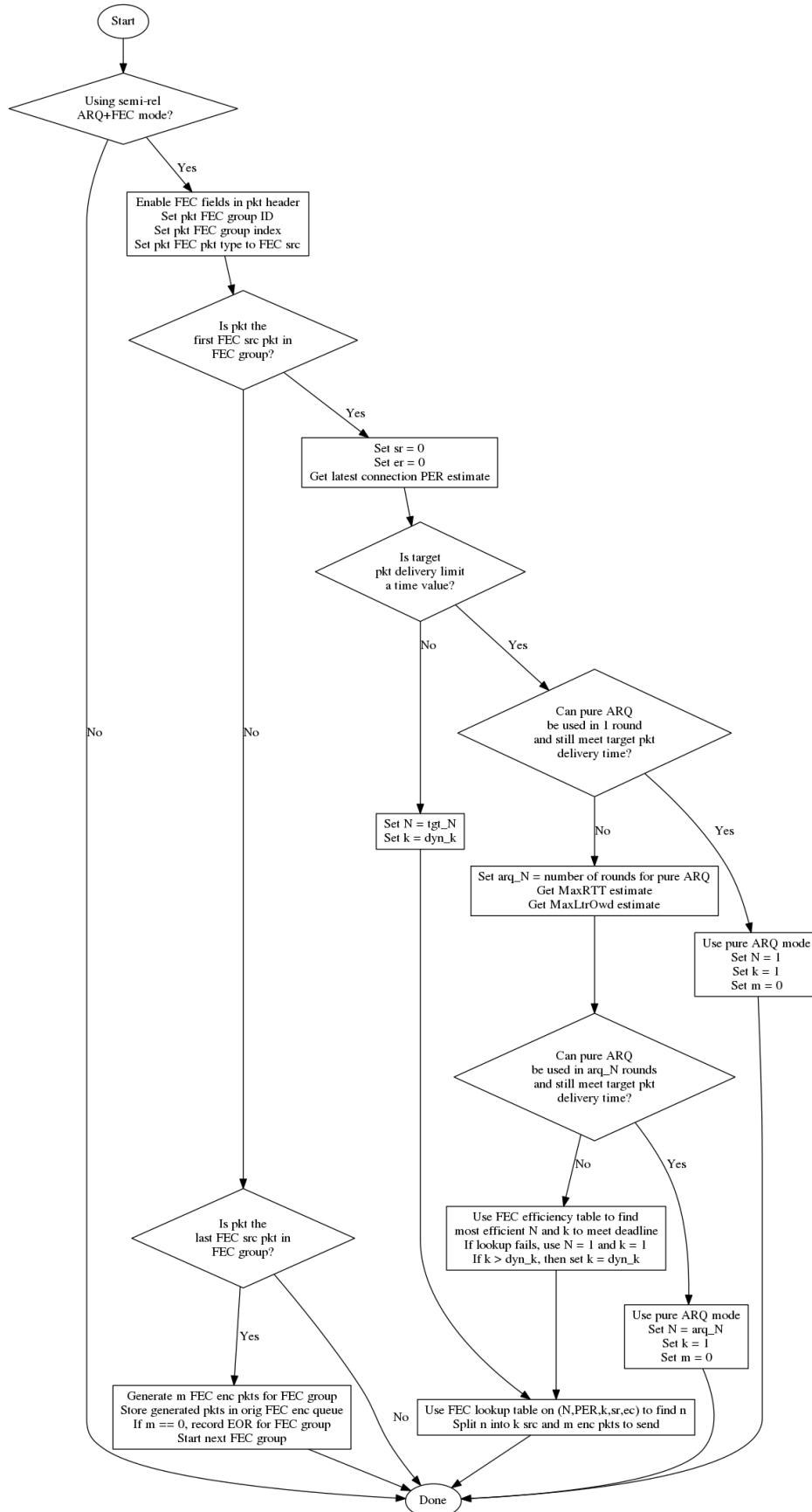
- To determine if pure ARQ can be used in a single round, the target packet receive probability for the stream (P_{recv}) and the current PER estimate (PER) are needed. Pure ARQ in a single round can be used if the following inequality is true: $PER \leq (1 - P_{recv})$.
- To determine if arg_N rounds of pure ARQ can still meet the target packet delivery time, arg_N is first computed using the target packet receive probability for the stream (P_{recv}) and the current PER estimate (PER) as described above. Then, arg_N rounds of pure ARQ can still meet the deadline if the following inequality is true:

$$TgtPktDelTime > (((arg_N - 1) \cdot MaxRtt) + MaxLtrOwd)$$

- To determine if either pure FEC ($N=1, m>0$) or coded ARQ ($N>1, m>0$) modes can be used to meet the deadline, the following procedure is used. First, the FEC lookup table is used to look up the maximum number of FEC data packets that are being sent using the target number of rounds for the group (N), the current PER (PER), the number of FEC source data packets in the group (k), zero received FEC source data packets in the group ($sr=0$), and zero received FEC encoded data packets in the group ($er=0$). Assume that the midgame lookup result is stored in $MgMaxDof$, and the endgame lookup result is stored in $EgMaxDof$. Then, each candidate FEC group (defined by N and k) can be used if the following inequality is true:

$$TgtDelTime > (((N - 1) \cdot (((MgMaxDof + 1) \cdot MaxPst) + MaxRTT)) + ((EgMaxDof \cdot MaxPst) + MaxLtrOwd))$$

Note that the algorithm chooses the candidate that meets this inequality and has the highest efficiency as reported by the FEC efficiency table.



7.3.1.3.5.4 Data Reception

The state machine shown below is the data packet reception state machine, which is used for a specific stream.

When a data packet is received, the reception timestamp is provided to SLIQ by the operating system kernel. By utilizing the timestamp provided by the kernel, any time spent by the kernel waiting to deliver the packet to SLIQ is not included in any of the following time computations. This provides the IRON system with the most accurate time calculations possible when running in user space.

The SLIQ ACK packets are used for sending data packet acknowledgement information for the stream. ACK packets contain the next expected sequence number, the largest observed sequence number (required to be in one of the ACK blocks), and ranges of sequence numbers from the next expected sequence number to the largest observed sequence number that have been received (ACK blocks).

The ACK timer is used for ACK suppression, basically allowing SLIQ to send an ACK packet for every other data packet received when the connection data flow is continuous and there are no missing data packets in any of the streams. It is implemented at the connection level. The duration of this timer is currently set to 40 milliseconds. If there are any missing data packets in any of the streams within the connection, then an ACK packet is sent immediately upon receiving a data packet.

These ACK rules are extended for two cases when additional ACK packets are sent. Note that extra ACKs are maintained for each stream within the connection separately.

1. The first case is when a stream enters a post-recovery situation, where the receipt of a data packet causes the stream to go from a state where data packets were missing to a state where there are no missing data packets. In this case, three additional ACK packets are scheduled for the stream using the ACK timer. Sending these additional ACK packets in this situation increases the probability that the ACK information will get back to the sender, allowing the sender to avoid unnecessary data packet retransmissions.
2. The second case is when a stream is using the semi-reliable ARQ+FEC reliability mode. In this case, each data packet reception causes an ACK packet to be sent immediately, and $\text{ceil}(3 + (20 \cdot PER))$ additional ACK packets are scheduled using the ACK timer, where PER is the current local-to-remote packet error rate estimate. Sending these additional ACK packets in this situation gets the ACK information back to the sender quickly, even if many ACK packets are being lost in the network. This is important, as the semi-reliable ARQ+FEC mode is designed to meet a packet delivery deadline for a portion of the data packets being sent.

The implementation of these extra ACKs utilizes a single variable for holding the current number of extra ACKs required for the stream, and its value is always adjusted using

$$\text{ackcnt}' = \max(\text{ackcnt}, x)$$

where x is the new number of extra ACK to incorporate for the stream. The use of the $\max()$ function prevents large buildups of extra ACKs that are not necessary. The ACK timer utilizes the 40 millisecond duration as described above as long as all of the streams in the connection have extra ACK counts of zero. If one or more streams within the connection have a non-zero extra ACK count, then the ACK timer duration is set to $d = \max((1.5 \cdot irt), 1ms)$, where irt is the expected data packet inter-receive time and 1 millisecond is the minimum duration supported.

Any data packets having the persist flag set are, in fact, empty packets sent for the purposes of keeping the connection alive during periods when there are otherwise no data packets being sent (for example, when in an outage mode). Hence, the sequence number of these persist data packets is meaningless, and these packets may be discarded as shown in the diagram below.

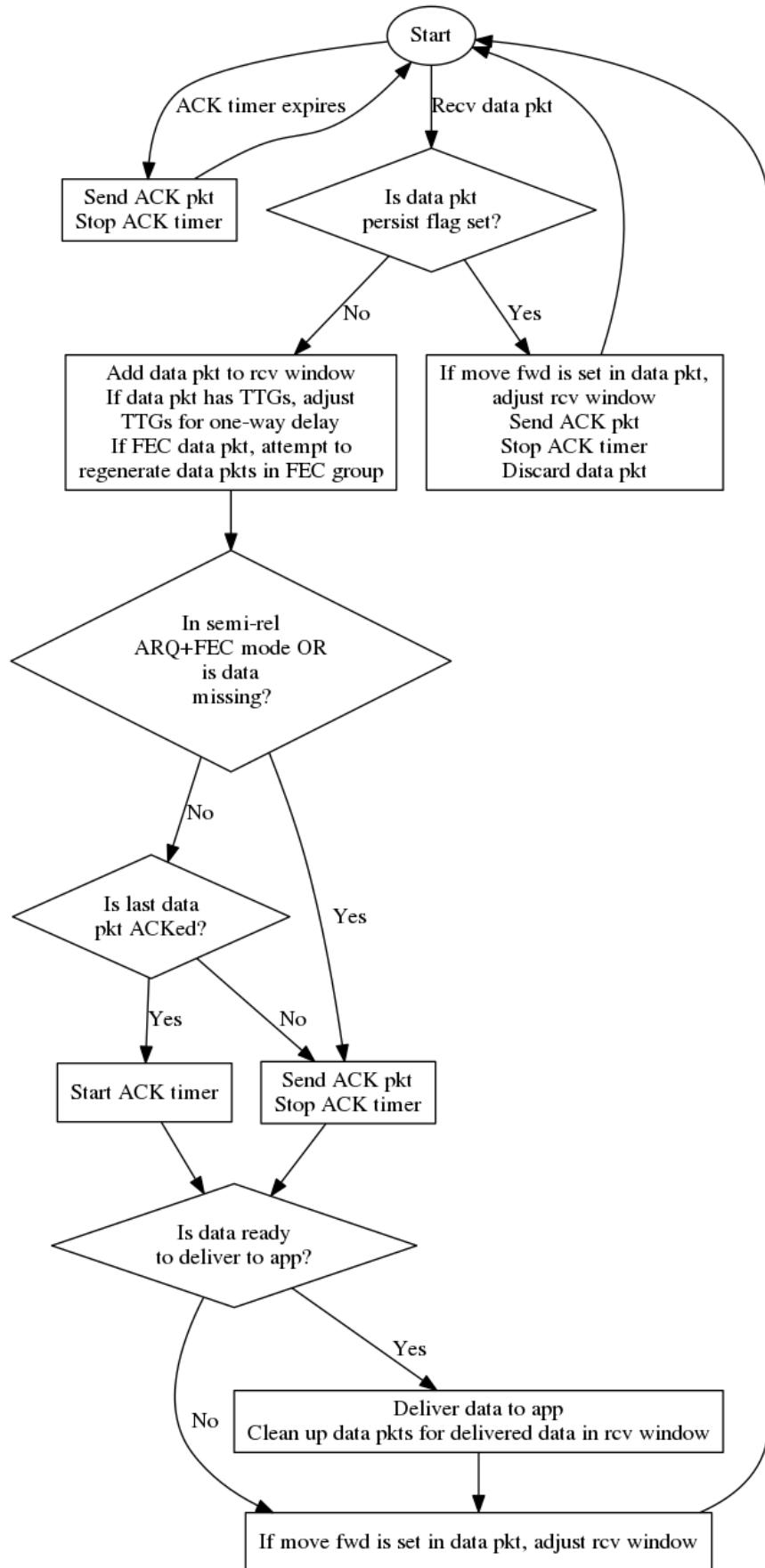
If the received data packet has any TTG values in the data packet header, they must all be adjusted down by the current remote-to-local one-way delay estimate. Once a TTG value reaches zero, it stays at zero and signifies that the data has missed its delivery deadline. The processing of TTG values depends on the type of packet received.

- If the data packet is a non-FEC data packet, then there will be a single TTG value for the data packet's payload.
- If the data packet is an FEC encoded data packet, then there will be k TTG values, one for each of the FEC group's FEC source data packets. Any FEC source data packets regenerated for the FEC group due to the receipt of this FEC encoded data packet will have their TTG values computed as follows. Each regenerated FEC source data packet's TTG value will be the newly received FEC encoded data packet's TTG value for the regenerated FEC source data packet (indexed in the TTG values using the FEC source data packet's FEC group index).
- If the data packet is an FEC source data packet, then there will be a single TTG value for the data packet's payload. Any other FEC source data packets regenerated for the FEC group due to the receipt of this FEC source data packet will have their TTG values computed as follows. Each regenerated FEC source data packet's TTG value will be the most-recently received FEC encoded data packet's TTG value for the regenerated FEC source data packet (indexed in the TTG values using the FEC source data packet's FEC group index), adjusted down by the difference between the most-recently received FEC encoded data packet's TTG value for the received FEC source data packet and the received FEC source data packet's TTG value.

The regeneration of FEC data packets takes place when an FEC group has received k FEC source and encoded data packets, and there are still missing FEC source data packets. The FEC group index of the received FEC data packets is used to organize the received packets and regenerate all of the missing FEC source data packets for decoding. Since each data packet must have a packet sequence number for proper delivery and ACKing, the FEC encoded data packets are generated with the 32-bit data packet sequence numbers appended to the FEC source data packet payloads and the payload lengths incremented by 4 bytes. When FEC source data packets are regenerated, the regenerated payloads include the 32-bit data packet sequence number at the end, allowing the sequence number to be removed and used to place the data packet in the proper

place in the receive window. Since each data packet payload may be a different length, the data packet header of each FEC encoded data packet includes an encoded packet length field. The received FEC source data packet payload lengths, incremented by 4 to include the data packet sequence numbers appended to their payloads, are used along with the received FEC encoded data packet encoded packet lengths in order to regenerate the correct payload lengths, also incremented by 4 to allow for the data packet sequence numbers appended to their payloads, for the regenerated FEC source data packets. The result is the regeneration of lost data packets with the proper sequence numbers and payload lengths.

Delivery of data to the application may be either in-order or out-of-order. This is configurable, and is set when the stream is created. Both ends of the stream must use the same delivery mode. For any latency-sensitive data that is delivered, the TTG value is adjusted down by any time that the data was spent queued up for delivery at the local SLIQ endpoint, and the new TTG value is delivered with the data to the application.



7.3.1.3.5.5 ACK Reception

The state machine shown below is the ACK reception state machine, which is used for a specific stream.

Round trip time (RTT) samples are computed using the observed packet times in the received ACK packet as follows.

- When a data packet is sent, a local timestamp is generated from the current time and inserted into the data packet Timestamp field.
- When the data packet is received at the peer, the peer records the data packet's reception time and Timestamp field in the receive window.
- When an ACK packet is generated at the peer, the receive window is checked for data packets that have been received but that have not had their observed packet times reported back to the sender. The time difference from the data packet reception time until the ACK packet generation time is added to the stored data packet timestamp in order to generate a corrected data packet timestamp. The resulting corrected data packet timestamp is stored along with the data packet's sequence number in one of the Observed Packet Timestamp and Sequence Number fields the ACK packet.
- When the ACK packet is received, all of the ACK packet's observed packet times are processed. The RTT sample for each data packet listed in the observed packet time entries is the local timestamp for the ACK packet's reception time minus the Observed Packet Timestamp from the ACK packet.
- The smoothed RTT and RTT mean variance are then computed as for TCP, which is specified in RFC 6298.

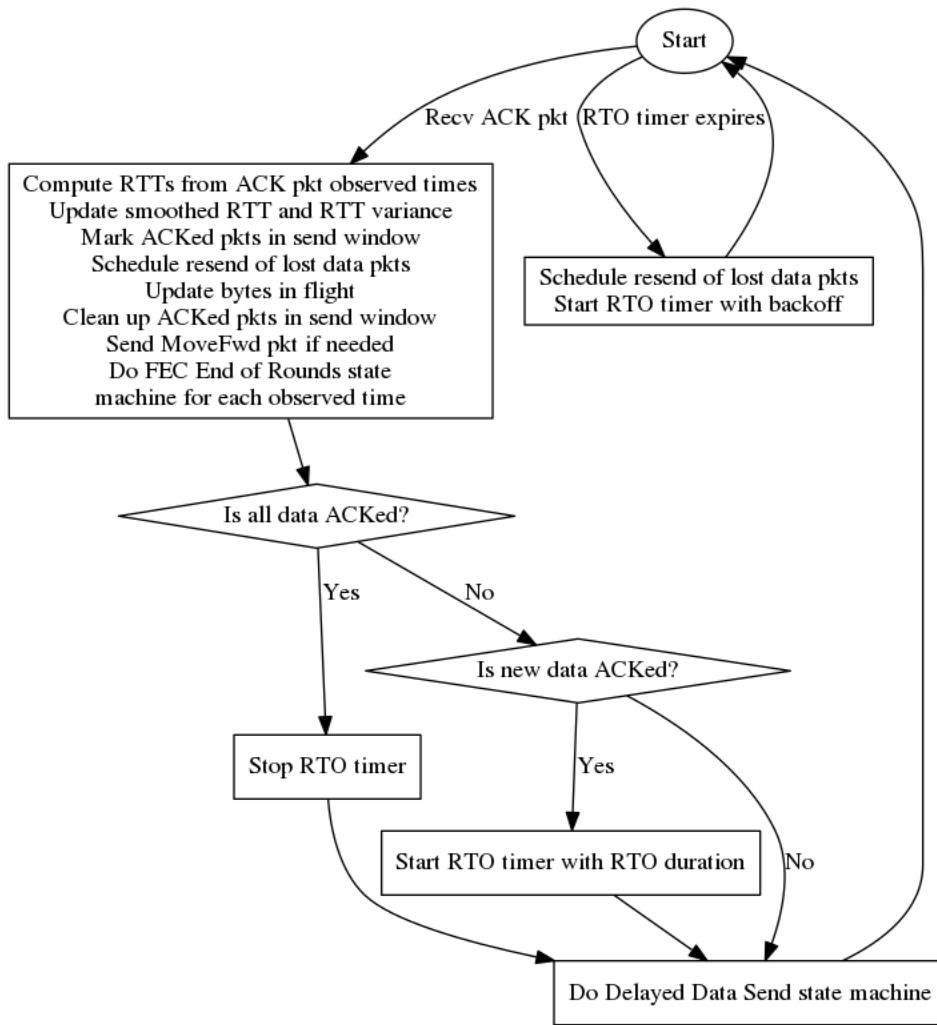
The bytes-in-flight value is the number of bytes sent in data packets to the peer that have not yet been ACKed. This value can be used by congestion control to determine if a data packet can be sent immediately or must be queued for sending later.

When fast RTOs are not required, the RTO timer duration defaults to 1 second. Once RTT estimates are available, it is set to the smoothed RTT plus 4 times the RTT mean deviation. The duration is always bounded to be between 200 milliseconds and 1 second. Each time the RTO timer expires, a back-off is applied by doubling the duration for the next timer. The settings when fast RTOs are required are detailed in the Application Data Send section above.

The congestion control algorithm (or algorithms, if configured to use multiple algorithms in parallel) specified for the connection by the client endpoint determines when unACKed packets are considered lost. This might require a data packet to be excluded from the ACK blocks in three separate ACK packet, or it might use some other rule. It is also up to the congestion control algorithm(s) to determine how many data packet retransmissions may occur upon processing each ACK packet, and if these retransmissions are to be performed immediately or must be paced along with original data packets (send pacing).

SLIQ move forward packets are sent to the receiver in order to make it skip over data packets that it has not yet received and that will no longer be sent to it. The move forward packets are not used by the stream if the stream is configured to be fully reliable. If the stream is semi-reliable,

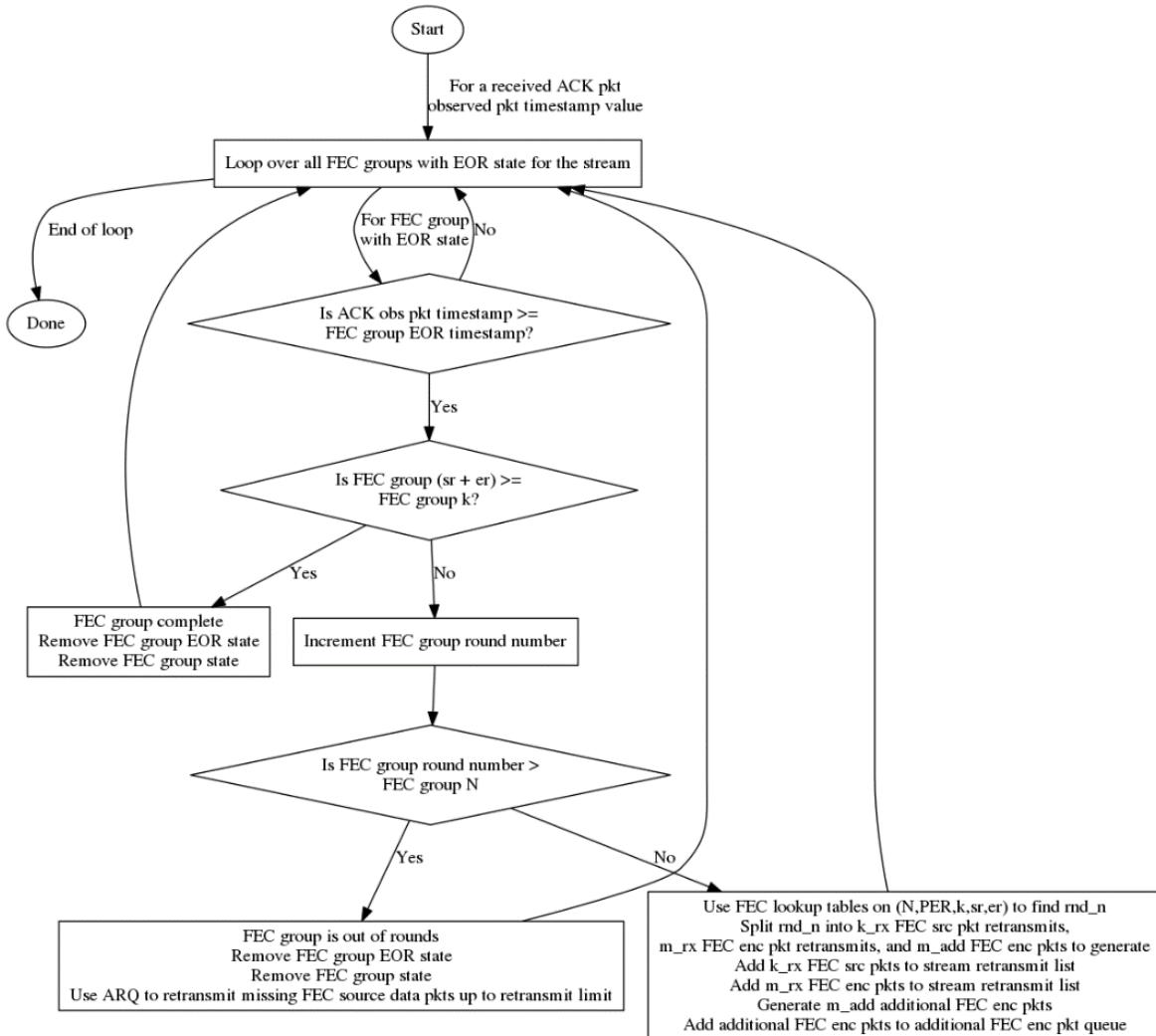
then any data packets that have not been ACKed once the retransmission limit is reached (a configurable parameter for the stream) are dropped by the sender and a move forward packet is sent to the receiver to force it to ignore those packets. If the stream is best-effort, then any data packet that is considered lost (determined by the congestion control algorithm(s)) is dropped by the sender and a move forward packet is sent to the receiver to force it to ignore those packets.



The FEC End of Rounds state machine, referenced in the ACK Reception state machine above, is shown below. This is called for each observed packet timestamp included in a received ACK packet.

Each round for an FEC group begins when the first FEC data packet is sent in the round, and ends when all of the FEC data packets scheduled for the round have been transmitted and enough time has elapsed that all of the ACK packets for those FEC data packets will have been received. Because FEC data packets and ACK packets can be lost in the network, the end of the round cannot depend on the receipt of a specific ACK packet. Instead, it must rely on the packet timestamps. The end of round (EOR) state for an FEC group is created when the last FEC data

packet (either source or encoded) for that group is sent in the current round. The EOR state includes the FEC group identifier and the EOR timestamp, which is the data packet header timestamp value that was sent in the last FEC data packet for the group in the current round. Since the other SLIQ endpoint sends the data packet header timestamp values back in ACK packet observed packet timestamp fields (adjusted up for the time between data packet reception and ACK packet transmission), as soon as a received observed packet timestamp is greater than or equal to the EOR state's EOR timestamp, then the FEC group for that EOR state has reached the end of the current round. So, instead of requiring the receipt of a specific ACK packet to trigger the EOR, the receipt of a specific ACK packet or a later ACK packet will trigger the EOR.



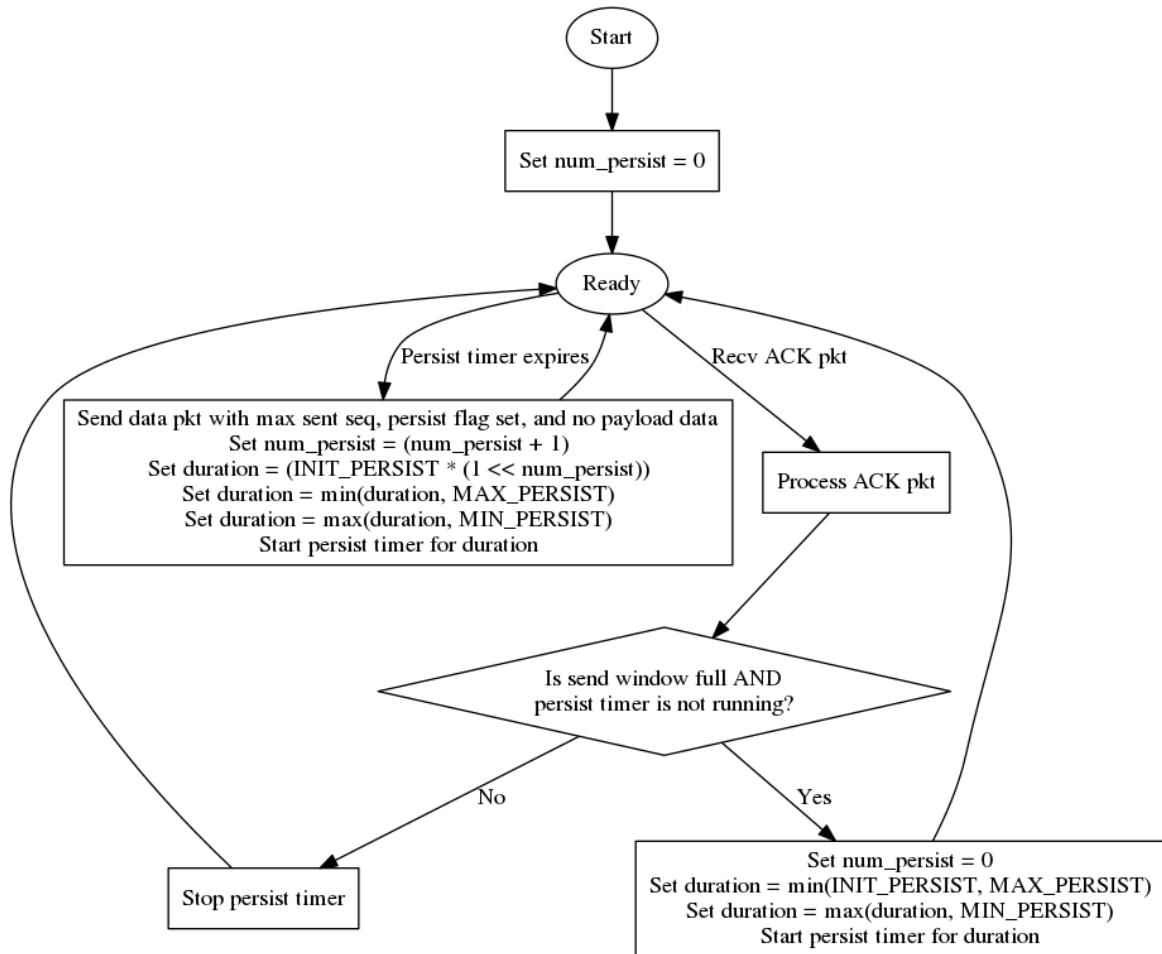
7.3.1.3.6 Persist Mode

The SLIQ protocol implements a *persist* mode that is very similar to TCP. Within a stream, when the sending endpoint runs out of available send window for sending additional data packets to the receiving endpoint, the send endpoint enters a persist mode that causes persist data packets to be sent periodically. The persist data packets do not contain data, but cause the receive endpoint

to generate ACKs, which allows the send endpoint to determine what data packets have been received and what data packets should be retransmitted to allow the send window to move forward again. Without the use of a persist mode, the sender would stop sending data when it runs out of available send window, and the receive endpoint would never send an ACK since it is not receiving any more data packets. The state machine for persist mode is shown below. Note that the INIT_PERSIST constant is 1.5 seconds, the MIN_PERSIST constant is 5.0 seconds, and the MAX_PERSIST constant is 60.0 seconds.

Persist data packets are also sent in other cases. When an RTO event occurs due to not receiving any ACK packets when some are expected, a data packet that needs to be retransmitted is usually sent in an attempt to have the receiving endpoint send an ACK packet back. However, if there are no data packets that require retransmission, then a persist data packet is sent instead. When leaving an outage condition, it is important for the sending endpoint to resynchronize with the receiving endpoint at the connection level using ACK packets, which involves having each stream within the connection send a data packet that needs to be retransmitted. However, if any of these streams do not have data packets that require retransmission, then a persist data packet is sent by the stream instead.

A persist data packet has the sequence number set to the last original data packet sequence number sent to the peer (i.e., the maximum sequence number sent thus far), has the persist flag set, and contains no data. Upon receipt, a persist data packet causes an immediate ACK packet to be sent, and the persist data packet is then discarded.



7.3.1.3.7 Outage Mode

The SLIQ protocol implements an outage mode that detects when there is no communication received from the peer when there should be (the link is "out"), and then detects when the outage is over much faster than simply relying on the RTO timer, with its back-off behavior. It operates at the connection level. The state machine for the outage mode is shown below.

The outage mode uses the existing RTO timer for its use. When the *in_outage* variable is set to true, then the RTO timer is being used for an outage. When in outage mode, the RTO timer outage duration used is the RTO duration with no back-off applied. This duration defaults to 1 second. Once RTT estimates are available, it is set to the smoothed RTT plus 4 times the RTT mean deviation. The duration is always bounded to be between 200 milliseconds and 1 second.

When the RTO time expires when in an outage most, a persist data packet is sent to the peer. A persist data packet has the sequence number set to the last original data packet sequence number sent to the peer, has the persist flag set, and contains no payload data. Upon receipt, a persist data packet causes an immediate ACK packet to be sent, and the persist data packet is then discarded.

When a data or ACK packet is received from the peer when in outage mode, the outage is over and data is purged from all of the streams in the connection as follows. If the stream is in reliable ARQ mode, then there is no transmit queue data purging. If the stream is semi-reliable ARQ or semi-reliable ARQ+FEC mode and the outage duration is greater than or equal to the retransmission limit (specified by the application) times the estimated retransmission time (the smoothed RTT plus 4 times the RTT mean deviation), then all of the data in the transmit queue is dropped. If the stream is in best-effort mode, then all of the data in the transmit queue is dropped.

The "is peer responsive" test in the state machine is as follows.

- When a data packet (either an original or a retransmission) that would generate an ACK is sent to the peer, and an ACK or data packet has been received since the last recorded data packet send time, then the current time is recorded in the connection state as the *data packet send time*.
- When an ACK or data packet is received from the peer, the current time is recorded in the connection state as the *ACK or data packet receive time*.
- If the *data packet send time* is greater than or equal to the *ACK or data packet receive time*, and the time difference from the *data packet send time* until now is greater than or equal to an expected ACK wait time, then the peer is not responsive. Otherwise, the peer is responsive.
- The expected ACK wait time takes into consideration the current packet error rate (PER) estimate and round-trip time estimate. It is computed as follows.
 - First, an adjusted PER estimate is computed that limits the PER value used to 0.9 or less:

$$per_adj = \min(per, 0.9)$$

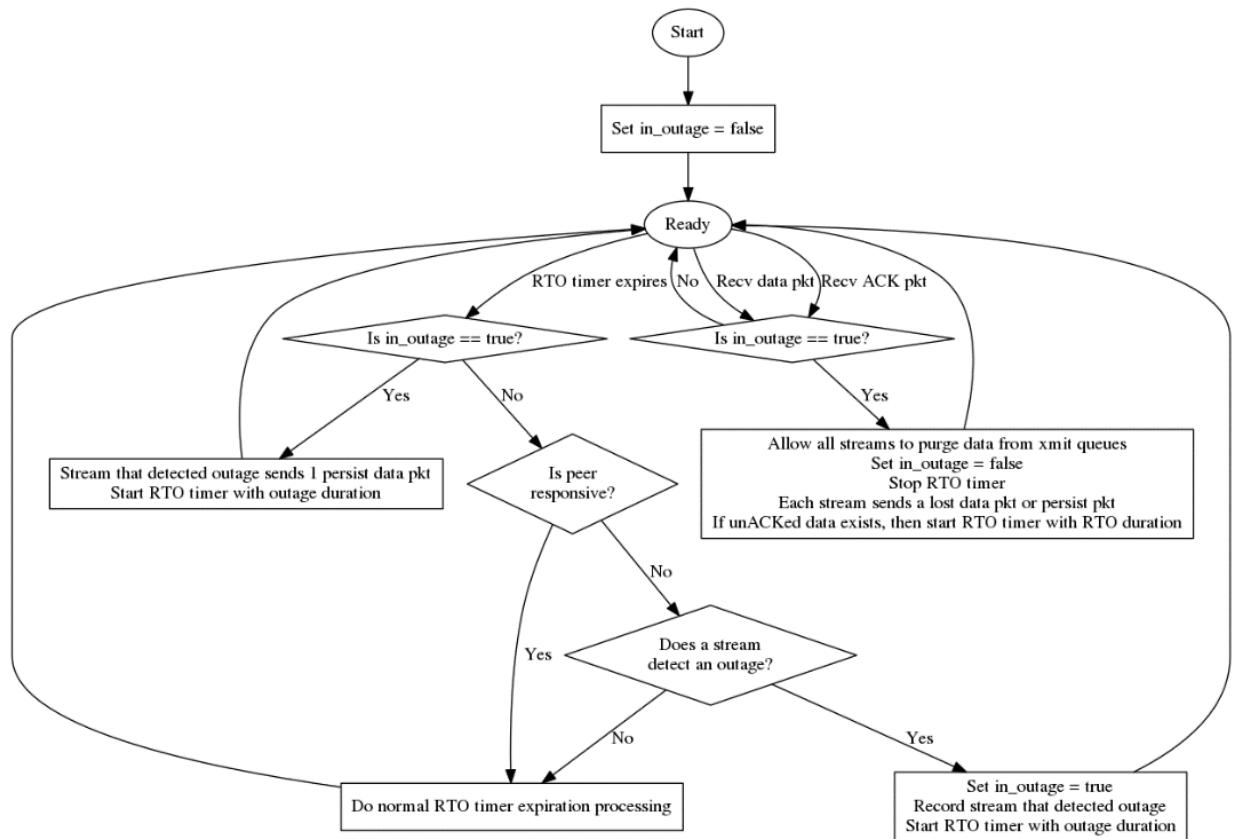
- Next, a PER correction factor is computed from the adjusted PER estimate:

$$per_corr = \frac{1}{(1-per_adj)^2}$$

- Next, an adjusted RTO time is computed as the current smoothed RTT, plus 4 times the current RTT mean deviation, plus an ACK delay time of 40 milliseconds, plus an RTO timer tolerance time of 4 milliseconds. The resulting *rto_adj* time value is bounded to a minimum value of 200 milliseconds.
- Finally, the expected ACK wait time is computed as:

$$exp_ack_wait = 2 \cdot rto_adj$$

The "does a stream detect an outage" test is applied to each stream in the connection as follows. If the connection is congestion-control blocked or the stream is flow-control blocked, and there is at least one data packet in the stream's transmit queue, then the stream has detected an outage. Otherwise, the stream has not detected an outage.



7.3.2 Congestion Control Algorithms

Internally, SLIQ provides a common congestion control API that allows adding new congestion control algorithms with a minimal amount of integration work. SLIQ currently supports CUBIC (two versions, an older Google-derived version that is no longer used and a newer version based on the relevant RFCs, papers, and the Linux kernel's implementation) and Copa (three versions, Copa Beta 1, Copa Beta 2, and Copa) congestion control algorithms. There is also a Reno mode within the older Google-derived CUBIC implementation, but it was part of the original QUIC implementation and is not used.

Because the congestion control algorithm operates at the connection level, but data packet sequence numbers are assigned at the stream level, the data packet sequence numbers are not monotonically increasing numbers to the congestion control algorithm. For this reason, the congestion control algorithm assigns its own monotonically increasing sequence numbers to the data packets as they are being sent, and this number is stored along with the other data packet information in the SentPacketManager. These new sequence numbers are called congestion control sequence numbers.

The common congestion control API consists of the following methods in a CongCtrlInterface base class. The various congestion control algorithms are implemented as child classes that inherit from the CongCtrlInterface base class. The methods that are called by SLIQ for each congestion control algorithm are listed here.

Event Method	Details
Configure	Called at startup. Includes the congestion control settings specified by the application. Returns true if the instance is configured successfully, or false if not.
Connected	Called once the SLIQ connection is established. Includes the current time and the RTT estimate from the connection handshake.
UseRexmitPacing	Called at startup. Returns true if all retransmitted data packets should be paced along with the original data packets that are being sent, or false if all retransmitted data packets should be sent immediately. Note that any scheduled retransmitted data packets always have priority over any scheduled original data packets when performing send pacing.
UseCongWinForCapEst	Called at startup. Returns true if the congestion window size returned by GetCongestionWindow should be used for computing channel and transport capacity estimates, or false if CapacityEstimate should be used instead.
UseUnaPktReporting	Called at startup. Returns true if the algorithm requires reporting of the oldest unacknowledged data packet for all streams using ReportUnaPkt, or false if not.
SetTcpFriendliness	Called to adjust the TCP friendliness or aggressiveness of the algorithm. The number of equivalent TCP flows is specified. Returns true if the change is successful, or false if not.
ActivateStream	Called when a new stream is added to the connection. Includes the assigned stream identifier and the initial stream send sequence number. Returns true if the change is successful, or false if not.
DeactivateStream	Called when a stream within the connection is closed. Includes the stream identifier of the stream being closed. Returns true if the change is successful, or false otherwise.
OnAckPktProcessingStart	Called before a series of OnRttUpdate, OnPacketLost, and OnPacketAcked calls are made for a collection of ACK headers received within a single UDP/IP packet. Includes the packets receive time.
OnRttUpdate	Called when an RTT estimate is computed based on an observed packet time received in an ACK header. Includes the stream identifier, the time that the ACK header was received, the Timestamp field from the ACK header, the local timestamp when the ACK header was received, the sequence number of the data packet being ACKed, the congestion control sequence number assigned to the data packet being ACKed (see OnPacketSent), the RTT estimate, the number of payload bytes in the data packet being ACKed, and the congestion control specific value that was stored when the packet was sent (see OnPacketSent) or resent (see OnPacketResent).
OnPacketLost	Called when a data packet is listed as missing in a received ACK packet and is at least three data packets behind the reported largest observed data packet sequence number. Includes the stream identifier, the time that the ACK header was received, the sequence number of the missing data packet, the congestion control sequence number assigned to the missing data packet (see OnPacketSent), and the number of payload bytes in the missing data packet. Returns true if the data packet should be considered lost, or false if it should not be considered lost yet. Once a data packet is considered lost, this event stops being called for the data packet.
OnPacketAcked	Called when a data packet that was sent is ACKed. Includes the stream identifier,

Event Method	Details
	the time that the ACK header was received, the sequence number of the data packet being ACKed, the congestion control sequence number assigned to the data packet being ACKed (see OnPacketSent), the next expected data packet sequence number from the ACK header, and the number of payload bytes in the data packet being ACKed. Only called once for each data packet.
OnAckPktProcessingDone	Called after a series of OnRttUpdate, OnPacketLost, and OnPacketAcked calls are made for a collection of ACK headers received within a single UDP/IP packet. Includes the packet's receive time.
OnPacketSent	Called when an original data packet has been sent the first time. Not called on data packet retransmissions (see OnPacketResent). Includes the stream identifier, the data packet's send time, the sequence number of the data packet assigned by the stream, the number of payload bytes in the data packet, and the total number of bytes transmitted. Returns the congestion control assigned sequence number to the data packet, and a congestion control specific value to be stored and passed into OnRttUpdate when it is called for the data packet later.
OnPacketResent	Called when a data packet has been resent. Not called on the data packet's initial transmission (see OnPacketSent). Includes the stream identifier, the data packet's resend time, the sequence number of the data packet, the congestion control sequence number assigned to the data packet (see OnPacketSent), the number of payload bytes in the data packet, the total number of bytes transmitted, a flag indicating if this retransmission is due to an RTO event, and a flag indicating if this congestion control algorithm sent the original transmission. Returns a congestion control specific value to be stored and passed into OnRttUpdate when it is called for the data packet later.
ReportUnaPkt	Called to report the oldest unacknowledged data packet for a stream. Only called if UseUnaPktReporting returned true. Includes the stream identifier, a flag indicating if the stream currently has an oldest unacknowledged data packet, and the oldest unacknowledged data packet sequence number (only used if the previous flag is true).
RequireFastRto	Called periodically to determine if the use of fast RTO timer logic is required or not. Returns true if fast RTO timer logic is to be used, or false if not.
OnRto	Called when the RTO timer expires. Includes a flag indicating if the oldest missing data packet on the highest priority stream was retransmitted as part of the RTO.
OnOutageEnd	Called when an outage is over.
UpdateCounts	Called to update the packets in flight, bytes in flight, and pipe size (per RFC 6675, Section 2). Includes the adjustment amounts for all three counts. Implemented in the base class, not in the child classes.
CanSend	Check if an original data packet can be sent now due to congestion control. Includes the current time and the number of payload bytes in the data packet. Returns true if the original data packet can be sent immediately, or false if not.
CanResend	Check if a fast retransmission of a data packet can be sent now due to congestion control. Includes the current time, the number of payload bytes in the data packet, and a flag specifying if this congestion control algorithm sent the original transmission. Returns true if the retransmission can be sent, or false if not. If true is returned and UseRexmitPacing returned true at startup, then the retransmission must be paced using TimeUntilSend instead of being sent immediately.
TimeUntilSend	Get the time of the next data packet transmission in order to implement send pacing. Includes the current time. If the time returned is zero, then the transmission can occur immediately. Always called for original data packet transmissions, but only called for data packet retransmissions if UseRexmitPacing

Event Method	Details
	returned true at startup.
SendPacingRate	Get the current send pacing rate, in bits/second. Returns the current send pacing rate, or zero if the rate is unknown.
SendRate	Get the current send rate, in bits/second. Returns the current send rate.
GetSyncParams	Get any congestion control synchronization parameters to be sent to the peer. Returns a flag indicating if there is a parameter to be sent, an unsigned 16-bit sequence number, and the unsigned 32-bit parameter to be sent. Called when SLIQ has an opportunity to include a Congestion Control Synchronization header along with other headers that are to be sent in a single UDP/IP packet.
ProcessSyncParams	Called to process received Congestion Control Synchronization header parameters from the peer. Includes the header receive time, the received unsigned 16-bit sequence number, and the received unsigned 32-bit parameter.
ProcessCcPktTrain	Called to process received Congestion Control Packet Train header parameters from the peer. Includes the header receive time and the received Congestion Control Packet Train header.
InSlowStart	Check if the algorithm is currently in slow start. Returns true if currently in slow start, or false if not.
InRecovery	Check if the algorithm is currently in fast recovery. Returns true if currently in fast recovery, or false if not.
GetCongestionWindow	Get the current congestion window size, in bytes. Returns the current congestion window size. If the congestion control algorithm does not implement a congestion window, then 0 must be returned.
GetSlowStartThreshold	Get the current slow start threshold, in bytes. Returns the current slow start threshold. If the congestion control algorithm does not implement a slow start threshold, then 0 must be returned.
GetCongestionControlType	Get the congestion control type value.
Close	Closes the congestion control algorithm.

The framework for all congestion control algorithms also maintains state variables for the following items.

- *bif* - The number of bytes in flight. This is the number of payload data bytes sent to the peer that have not been acknowledged yet.
- *pif* - The number of packets in flight. This is the number of data packets sent to the peer that have not been acknowledged yet.
- *pipe* - The pipe size in bytes. Defined in RFC 6675, Sections 2 and 4, as the number of payload data bytes that have not been ACKed and have not been considered lost yet, plus the number of payload data bytes that have not been ACKed and have been retransmitted.

These may or may not be used by the individual congestion control algorithms as needed. Since these state variables are updated by the congestion control framework, its updating logic inside of UpdateCounts is not included in the following state machines.

The designs of the CUBIC, Copa Beta 1, Copa Beta 2, and Copa congestion control algorithms are described in the following subsections.

7.3.2.1 CUBIC

The SLIQ CUBIC congestion control algorithm is a clean-slate implementation that is based on the TCP CUBIC congestion control algorithm as described in the relevant RFCs and implemented in the Linux kernel. CUBIC is loss-based, which means that it uses packet losses to estimate the amount of congestion at the bottleneck link between the two endpoints. It does this by assuming that all packet losses are due to network switch buffer overflows (i.e., congestion losses). This CUBIC implementation also includes hybrid slow start, proportional rate reduction, and send pacing.

Parts of the CUBIC algorithm are too complex to include in detail here. This includes the CUBIC window size adjustment logic, the hybrid slow start window size adjustment logic, and the proportional rate reduction window size adjustment logic. Instead of detailed state machines for these items, a summary of how these work and links to additional information is provided here.

- For the CUBIC window size adjustment logic, the window size is a CUBIC function of time since the last congestion event. The inflection point of this function is set to the window size prior to the last congestion event. The CUBIC function provides a concave portion for quickly ramping up the window size to the size before the last congestion event, and a convex portion for probing for more bandwidth, slowly at first then very rapidly. Note that CUBIC does not rely on the receipt of ACKs to increase the window size, it uses the last congestion event instead. Materials are readily available on the Internet describing this logic in detail, including [this](#) paper. Note that this logic has been modified in SLIQ to allow a number of TCP-equivalent flows to be specified in order to adjust the normal CUBIC α and β parameters, which results in different levels of aggressiveness.
 - The β parameter, which is the window size multiplicative back-off factor when a packet loss occurs (and is actually $(1 - \beta)$ when referencing β from the CUBIC paper), goes from being a constant of 0.7 to

$$(((\text{num_flows} - 1) + 0.7)/\text{num_flows})$$

when *num_flows* is adjusted using SetTcpFriendliness. This makes the algorithm more aggressive (a higher β value) when there are more TCP-equivalent flows specified.

- The α parameter, which is the TCP-fair additive increment per RTT (listed as $((3 \cdot \beta)/(2 - \beta))$ when referencing β from the CUBIC paper), is equal to $((3 \cdot (1 - \beta))/(1 + \beta))$. Since β is adjusted to be more aggressive when more TCP-equivalent flows are specified, α is adjusted for aggressiveness as well.
- The hybrid slow start window size adjustment logic is an extension to the normal TCP congestion control slow start algorithm. Slow start is the initial startup phase, and lasts until the first packet loss. This logic provides a signal to slow start for switching to congestion avoidance without incurring an extremely large number of packet losses, which is typical of the normal slow start phase. It does this by monitoring packet RTT delays and the inter-ACK delays, and when these delays start to increase, it indicates that the path is

getting congested. When these early signs of congestion are detected, this logic transitions SLIQ from slow start to congestion avoidance. Note that this implementation contains both the delay increase and ACK train length detection logic, and either of these algorithms may be disabled in the code. (Google avoided using the ACK train length detection logic in their CUBIC implementation because they claim that it interacts poorly with send pacing. We have left ours enabled.) A paper available [here](#) provides full details on this algorithm.

- The proportional rate reduction window size adjustment logic is an enhancement to the normal TCP fast recovery algorithm. When operating in congestion avoidance and a packet loss is detected, fast recovery is used to control the amount of data sent while the loss is repaired. This algorithm attempts to minimize excess window adjustments during fast recovery, and maintains its own window size that is only used during fast recovery. See RFC 6937 for full details on this algorithm.

The following acronyms are used in the following design description:

- SS - Slow Start.
- HSS - Hybrid Slow Start.
- PRR - Proportional Rate Reduction.

The following constants are used by CUBIC:

- MSS - The maximum segment size, which is set to 1460 bytes. This is simply a constant used to set various congestion control algorithm parameters.

The following state variables are maintained by CUBIC:

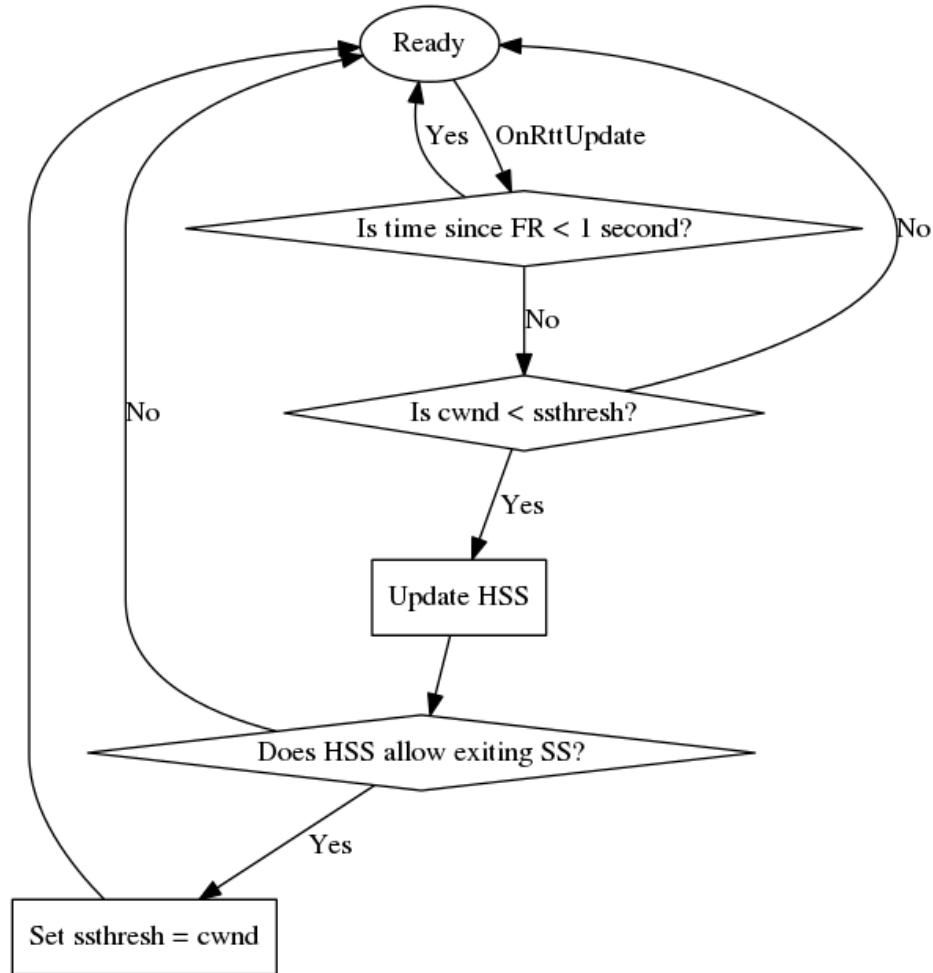
- *beta* - The primary aggressiveness parameter. Initialized to 0.7 and updated when the aggressiveness is changed.
- *cwnd* - The congestion window size in bytes. Initialized to (3 * MSS).
- *ssthresh* - The slow start threshold window size in bytes. Initialized to a very large value, approximately 1.4 million times the MSS.
- *in_fr* - A flag recording if currently in Fast Recovery or not. Initialized to false.
- *in_rto* - A flag recording if currently handling an RTO event. Initialized to false.
- *smoothed_rtt* - The smoothed RTT value for the connection as maintained by SLIQ. Implemented as defined for TCP in RFC 6298. Initialized to 1 second and updated for each RTT estimate.

CUBIC implements the congestion control API as shown in the following table and state machine diagrams.

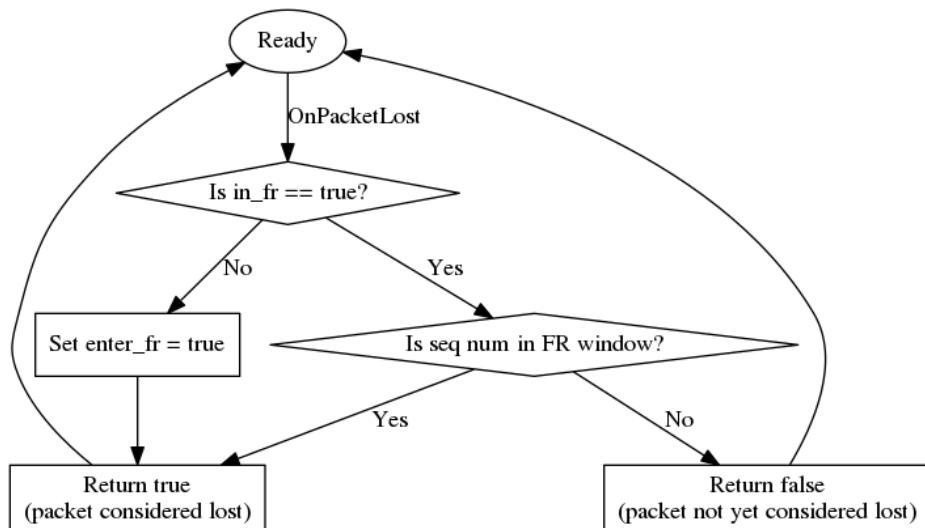
Event	CUBIC Processing Details
Configure	Initialize all state variables as listed above.
Connected	Ignored.
UseRexmitPacing	Returns true.
UseCongWinForCapEst	Returns true.
UseUnaPktReporting	Returns true.

Event	CUBIC Processing Details
SetTcpFriendliness	Updates the <i>beta</i> parameter based on the specified number of equivalent TCP flows using the equation above.
ActivateStream	Adds state information for the new stream, which is used for tracking the oldest unacknowledged packet in each stream.
DeactivateStream	Removes state information for the stream, which is used for tracking the oldest unacknowledged packet in each stream.
OnAckPktProcessingStart	Resets the ACK packet processing state. Includes <i>pre_ack_bif</i> set to the current <i>bif</i> value for use in <i>cwnd</i> testing and updates while <i>bif</i> is being updated, and a flag <i>enter_fr</i> used for recording if the OnPacketLost processing needs the OnAckPktProcessingDone call to enter FR at that time.
OnRttUpdate	See state machine below.
OnPacketLost	See state machine below.
OnPacketAcked	Set <i>in_rto</i> to false.
OnAckPktProcessingDone	See state machine below.
OnPacketSent	Assign the next congestion control sequence number to the packet. If no packets were sent for a while, then shift the CUBIC epoch start to keep the congestion window growth to a CUBIC curve. If <i>in_fr</i> is true, then update PRR. Update the next send time using a pacing rate of 2.0 times the rate (<i>cwnd / smoothed_rtt</i>) if (<i>cwnd < (ssthresh / 2)</i>), or 1.2 times the rate (<i>cwnd / smoothed_rtt</i>) otherwise.
OnPacketResent	If the retransmission is due to an RTO event, then return immediately. If <i>in_fr</i> is true, then update PRR. Update the next send time using a pacing rate of 2.0 times the rate (<i>cwnd / smoothed_rtt</i>) if (<i>cwnd < (ssthresh / 2)</i>), or 1.2 times the rate (<i>cwnd / smoothed_rtt</i>) otherwise.
ReportUnaPkt	Update the oldest unacknowledged packet for the stream.
RequireFastRto	Returns false.
OnRto	Reduce <i>ssthresh</i> to MAX((<i>cwnd * beta</i>), (2 * MSS)) if needed, reset <i>cwnd</i> to MSS, reset CUBIC following the <i>cubic_reset()</i> logic in the CUBIC paper, and reset HSS. If <i>in_fr</i> is true, then set <i>in_fr</i> to false and exit FR. Set <i>in_rto</i> to true.
OnOutageEnd	If <i>in_fr</i> is false, then set <i>cwnd</i> to MAX(number of bytes in flight, (3 * MSS)) and begin PRR.
CanSend	See state machine below.
CanResend	If <i>in_fr</i> is true, then return the result of the PRR test, otherwise return true.
TimeUntilSend	Returns the time from now until the next send time, or zero if now is beyond the next send time.
SendPacingRate	Returns a pacing rate of 2.0 times the rate (8 * <i>cwnd / smoothed_rtt</i>) if (<i>cwnd < (ssthresh / 2)</i>), or 1.2 times the rate (8 * <i>cwnd / smoothed_rtt</i>) otherwise.
SendRate	Returns the rate (8 * <i>cwnd / smoothed_rtt</i>) bits per second.
GetSyncParams	Returns false.
ProcessSyncParams	Ignored.
ProcessCcPktTrain	Ignored.
InSlowStart	Returns (<i>cwnd < ssthresh</i>).
InRecovery	Returns <i>in_fr</i> .
GetCongestionWindow	Returns <i>cwnd</i> bytes.
GetSlowStartThreshold	Returns <i>ssthresh</i> bytes.
Close	Ignored.

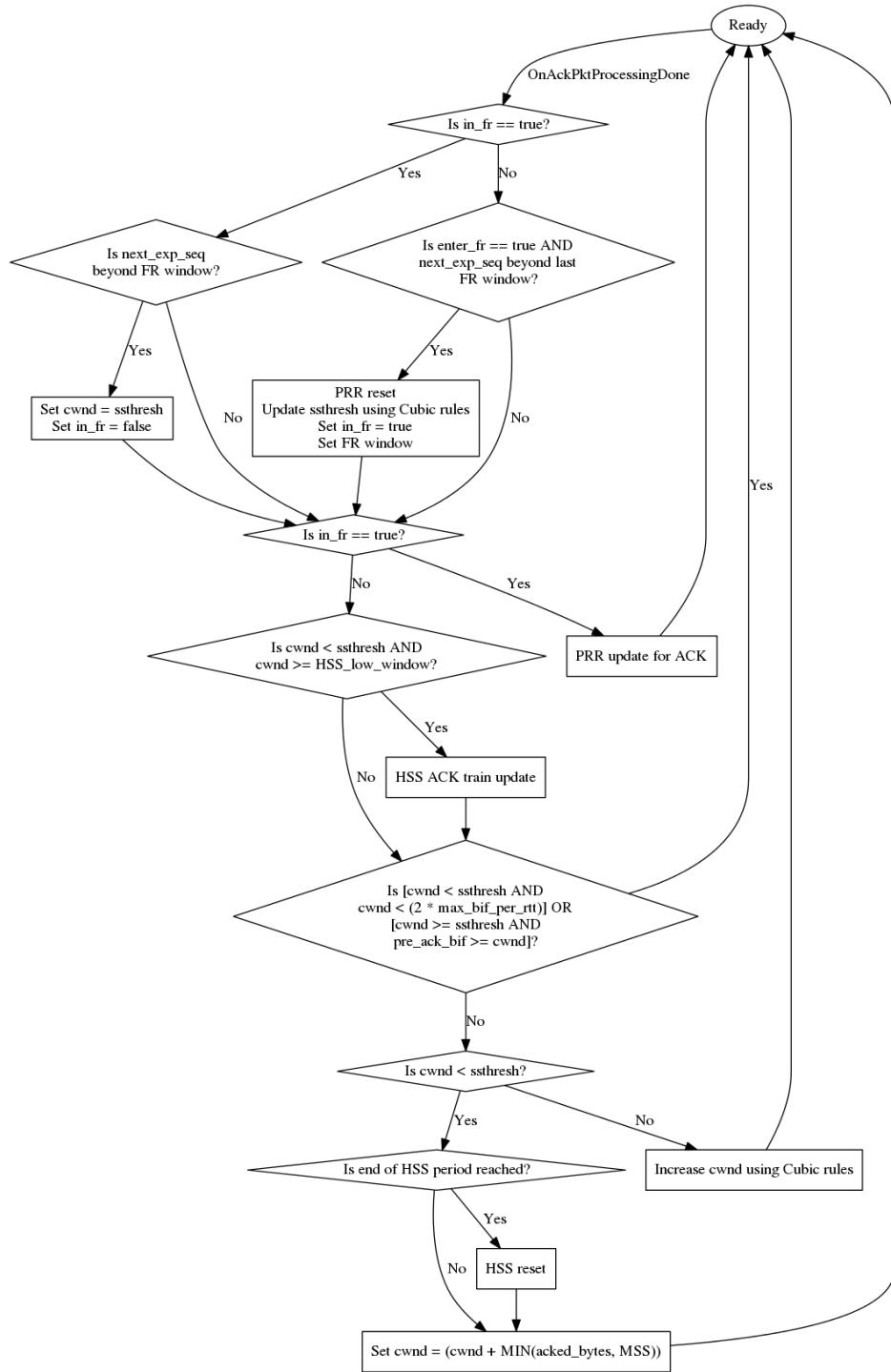
The OnRttUpdate event state machine is shown below. This logic updates hybrid slow start when necessary and controls when slow start is exited.



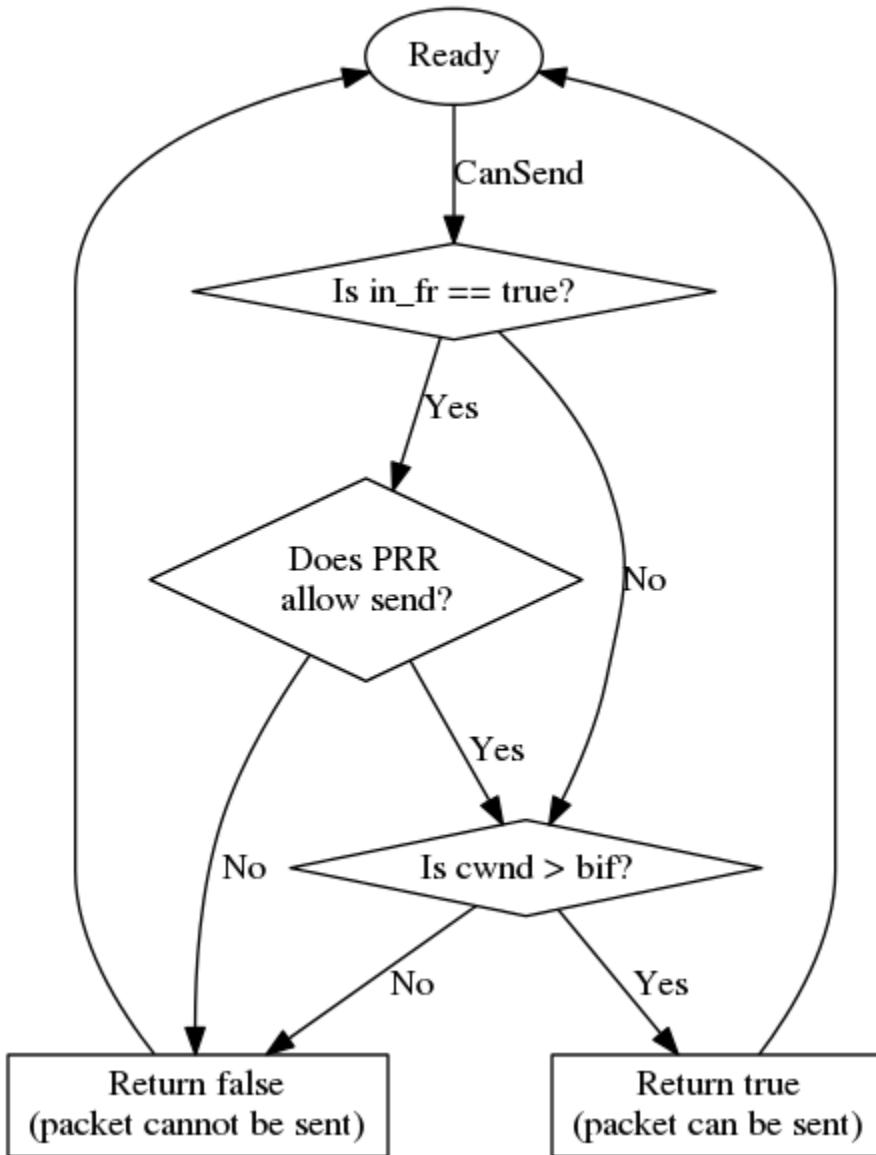
The OnPacketLost state machine is shown below. This logic handles determining if a missing packet that is at least 3 packets behind the reported largest observed packet sequence number should be considered lost and deciding when to enter fast recovery.



The OnAckPktProcessingDone state machine is shown below. This logic controls fast recovery, hybrid slow start, and updates to the congestion window. Note how the window size is not adjusted while in fast recovery, and there are conditions on the number of bytes-in-flight that must be met for the window size to be updated in either slow start or congestion avoidance.



The CanSend state machine is shown below. This logic controls the number of bytes-in-flight to avoid congestion.



7.3.2.2 Copa Beta 1

The first version of the Copa congestion control algorithm, called *Copa Beta 1* in order to be clear that it was the first beta version of Copa implemented by BBN, was developed by MIT. This algorithm is similar to TCP Vegas in that it is a delay-based algorithm, which means that it uses packet RTT calculations to estimate the amount of congestion in the network at the bottleneck link. This is fundamentally different than CUBIC in that it does not need packet losses to detect congestion, and allows Copa Beta 1 to operate with very little packet loss and a lower end-to-end packet latency. However, the RTT calculations are critical to this algorithm's operation, and events such as route changes can be difficult to detect and handle appropriately. The Copa Beta 1 implementation includes enhancements based on testing and communications with MIT.

However, it still has deficiencies that are addressed by the Copa congestion control algorithm. As such, the Copa congestion control algorithm should be used instead of this older version of the algorithm.

Copa Beta 1 uses the average switch delay, d_s , which is defined as the mean time spent by a packet in the bottleneck queue, as the measure of congestion. Each time that an RTT calculation is made due to the reception of an ACK packet, the Copa Beta 1 algorithm rate update rule sets the packet inter-send time, denoted by τ , to:

$$\tau = \max(\delta \cdot d_s, \tau_p/2)$$

where δ is an algorithmic parameter that controls the aggressiveness of the algorithm, and τ_p is the mean inter-send time when the last ACKed packet was sent (which was approximately one RTT ago). The $\tau_p/2$ term is a defensive measure that prevents the send rate from increasing more than double each RTT. The time spent by a single packet in the bottleneck queue may be estimated as:

$$d_s = (RTT - RTT_{min})$$

where RTT is a time-based weighted moving average of the computed RTT values using a weight of $8^{(-t/RTT_{min})}$ for an RTT value that was received t seconds before the current time, and RTT_{min} is the minimum RTT observed thus far. The outcome of these equations is that Copa Beta 1 attempts to stabilize at a point where it maintains $1/\delta$ packets queued at the bottleneck switch.

Copa Beta 1 also uses an "advance RTT update" mechanism to account for data packets that have been delayed for longer than the current average RTT. If the ACK for a data packet has been delayed for longer than the current average RTT estimate, then it assumes temporarily that a "virtual RTT sample" has just arrived whose value is equal to the current time minus the time the data packet was sent. This virtual sample is maintained until a data packet that was sent later is ACKed. This causes the average RTT estimate to be more accurate than it would otherwise be, and allows Copa Beta 1 to rapidly slow down when no ACKs are received for a long period of time.

Copa Beta 1's δ parameter may be a constant value, or it may be set dynamically using a policy controller. Since the optimal value of δ for a given network setting typically varies, setting it dynamically provides a more adaptable congestion control algorithm. If a policy controller is used, it is run periodically, typically once every $N \cdot RTT$ seconds, where N is an integer greater than or equal to one. Each time the policy controller is run, it uses the current latency values, loss rate, sending rate, and any other collected parameters in an objective function. The result of the objective function is to either increase or decrease δ , or do nothing. If δ is to be adjusted, then it should be done using an additive-increase multiplicative-decrease (AIMD) mechanism, while not setting it below a minimum allowable value. An optional policy controller for maximizing throughput while ignoring loss has been implemented for Copa Beta 1 by the IRON team.

When using the inter-send time τ to send data packets, Copa Beta 1 normally creates randomized inter-send times from τ using Markovian transmissions. This is done to avoid any adverse synchronization effects between different senders, as well as to obtain unbiased estimates of the packet delay. If this randomization is not done, then the algorithm is termed "Deterministic Copa Beta 1" since the packet inter-send times are deterministic. Simulations of Copa Beta 1 and Deterministic Copa Beta 1 by MIT have shown that Deterministic Copa Beta 1 gives lower delays and higher throughputs than Copa Beta 1, and our testing of both in IRON have confirmed this result. Thus, IRON has moved to using Deterministic Copa Beta 1 only.

At startup, Copa Beta 1 allows 10 data packets to be sent with a fixed inter-send time of 0.1 seconds. These are called probe packets, and are used to get a reasonable value for RTT_{min} . Once these 10 data packets have been ACKed, the normal Copa Beta 1 algorithms are used. During normal operation, Copa Beta 1 does not limit packet transmissions based on a congestion window. It relies entirely on the computed packet inter-send time in order to limit packet transmissions.

While CUBIC forces data packet retransmissions to be sent immediately, Copa Beta 1 uses send pacing of both original data packets and data packet retransmissions. Any waiting data packet retransmissions are always sent before any waiting original data packet transmissions.

Given the different ways that Copa Beta 1 may be configured, the Copa Beta 1 name uses prefixes and suffixes to reflect the selected configuration options. "Copa Beta 1" uses randomized inter-send times, while "Deterministic Copa Beta 1" uses the calculated inter-send times directly. If δ is a constant, then the version of Copa Beta 1 is labeled with the δ value as a suffix, such as "Deterministic Copa Beta 1 0.1" when a fixed δ value of 0.1 is being used. If the policy controller for maximizing throughput while ignoring loss (described above) is used, then a suffix of "M" (short for "Maximum Throughput") is used, such as "Deterministic Copa Beta 1 M".

The following constants are used by Copa Beta 1:

- NPS - The nominal packet size, which is set to 1000 bytes. This is used to adapt Copa Beta 1's computations from units of packets to bytes, in order to support variable packet sizes.
- DATA_HDR_BASE_SIZE - The base size of the SLIQ data header in bytes. Set to 16 bytes.
- PKT_OVERHEAD - The nominal packet overhead size, which is set to 54 bytes (26 bytes for Ethernet, 20 bytes for IP, and 8 bytes for UDP).
- NUM_PROBES - The number of probe packets. Set to 10.
- MAX_DELTA - The maximum allowable value for δ . Set to 0.1.
- MIN_DELTA - The minimum allowable value for δ . Set to 0.004.
- MAX_IST - The maximum allowable inter-send time in seconds. Set to 0.2 seconds.
- MIN_IST - The minimum allowable inter-send time in seconds. Set to 0.000008 seconds.
- PC_RTTS - The number of RTTs between runs of the policy controller. Set to 4.
- PC_ADD_INC - The Copa Beta 1 M policy controller additive increase value. Set to 0.0025.

- PC_MULT_DEC - The Copa Beta 1 M policy controller multiplicative decrease value. Set to 1.0/1.1.
- PC_MAX_INT - The maximum policy controller update interval in seconds. Set to 1 second.
- PC_SYNC_INT - The maximum policy controller synchronization interval in seconds. Set to 2 seconds.
- PC_SYNC_THRESH - The policy controller delta update threshold. Set to 0.1.
- QUIET_PERIOD - The maximum amount of time that a paced send can be late without the sender considered to be quiescent. Set to 0.01 seconds.

The following state variables are maintained by Copa Beta 1:

- *mode* - The current operating mode, either CONSTANT_DELTA or MAX_THROUGHPUT. Initialized to the mode specified in the Copa Beta 1 configuration.
- *random_send* - A flag recording if inter-send times should be randomized or not. Initialized to the flag specified in the Copa Beta 1 configuration.
- *delta* - The Copa Beta 1 algorithmic parameter for aggressiveness (δ). Initialized to the value specified in the Copa Beta 1 configuration, or 0.1 if either not specified in the configuration or using Copa Beta 1 M.
- *local_sync_delta* - The locally computed policy controller delta value. Initialized to 0.1.
- *remote_sync_delta* - The remotely computed policy controller delta value. Initialized to -1.0, which is an invalid delta value.
- *sync_param* - The congestion control synchronization parameter to report to the other endpoint. Initialized to 0.
- *prev_sync_param* - The previous congestion control synchronization parameter reported to the other endpoint. Initialized to 0.
- *prev_sync_time* - The time of the previous congestion control synchronization transmission. Initialized to 0.
- *intersend_time* - The inter-send time to use for the next packet transmission/retransmission in seconds. Initialized to 0.1 seconds.
- *calc_intersend_time* - The calculated inter-send time in seconds (τ). Initialized to 0.1 seconds.
- *prev_intersend_time* - The calculated inter-send time, in seconds, in effect when the latest data packet that has been ACKed was sent (τ_p). Initialized to 0 seconds.
- *min_rtt* - The minimum RTT observed in seconds (RTT_{min}). Initialized to 3600 seconds.
- *rtt_acked* - The weighted moving average RTT, in seconds, for data packets that have been ACKed. Initialized to 0 seconds.
- *rtt_unacked* - The weighted moving average RTT, in seconds, for data packets that have not been ACKed in the expected amount of time. Initialized to 0 seconds.
- *unacked_pkts* - An array of information for data packets that have not been ACKed yet. Each element contains a sequence number, a send time, the calculated inter-send time in effect when the packet was sent, an ACKed flag, a resent flag, and a skip until resent flag. The *una_cc_seq* and *nxt_cc_seq* variables determine the current range of elements being used. Initialized to be an empty array.
- *una_cc_seq* - The congestion control sequence number for the lowest unACKed data packet. This is the lowest element present in the *unacked_pkts* array. Initialized to 0.

- *nxt_cc_seq* - The congestion control sequence number for the next data packet to be sent. This is one element beyond the highest element present in the *unacked_pkts* array. Initialized to 0.
- *ack_cc_seq* - The highest congestion control sequence number ACKed thus far. Initialized to 0.
- *next_send_time* - The earliest absolute time, in seconds, for the next data packet transmission/retransmission. Initialized to the start time.
- *prev_delta_update_time* - The previous absolute time, in seconds, when the policy controller was last run. Initialized to the start time.
- *timer_tolerance* - The expected accuracy of the SLIQ event timer in seconds. Initialized to 0.001 seconds.
- *send_cnt* - The number of packets sent between policy controller updates. Initialized to 0.
- *quiescent_cnt* - The number of quiescent periods between policy controller updates. Initialized to 0.
- *num_pkts_acked* - The total number of data packets that have been ACKed. Initialized to 0.

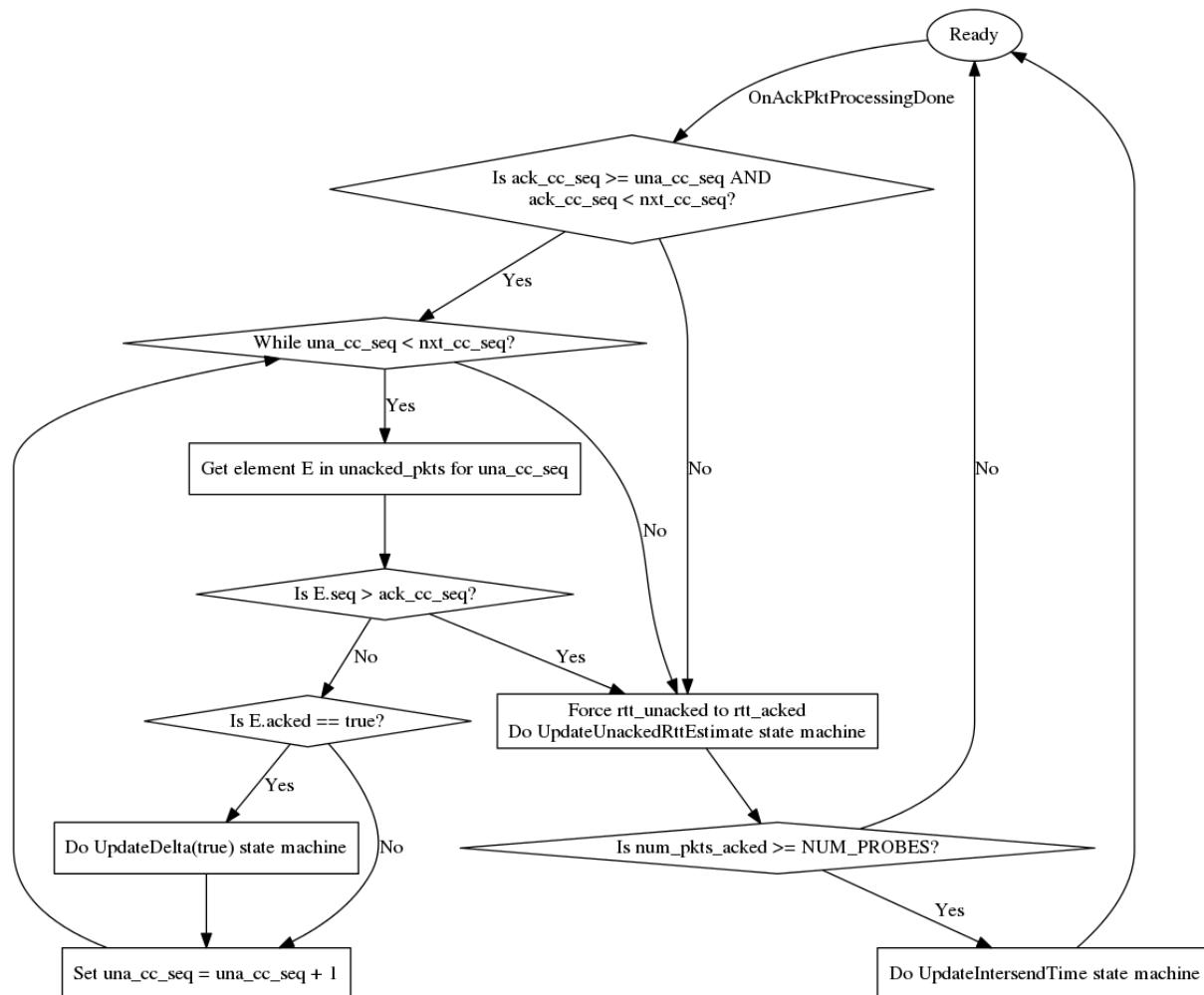
Copa Beta 1 implements the congestion control API as shown in the following table and state machine diagrams.

Event		Copa Beta 1 Processing Details
Configure	Initialize all state variables as listed above.	
Connected	Ignored.	
UseRexmitPacing	Returns true.	
UseCongWinForCapEst	Returns false.	
UseUnaPktReporting	Returns false.	
SetTcpFriendliness	Ignored.	
ActivateStream	Ignored.	
DeactivateStream	Ignored.	
OnAckPktProcessingStart	Ignored.	
OnRttUpdate	If the RTT measurement is less than <i>min_rtt</i> , then set <i>min_rtt</i> to the RTT measurement. If <i>rtt_acked</i> is greater than or equal to <i>min_rtt</i> , then <i>rtt_acked</i> is updated with the RTT measurement, otherwise <i>rtt_acked</i> is forced to be equal to the current <i>min_rtt</i> .	
OnPacketLost	Returns true.	
OnPacketAcked	If the ACKed data packet's congestion control sequence number is less than <i>una_cc_seq</i> or greater than or equal to <i>nxt_cc_seq</i> , then return immediately. If the <i>unacked_pkts</i> element for the data packet has the ACKed flag already set to true, then return immediately. Mark the data packet as ACKed in <i>unacked_pkts</i> by setting the ACKed flag to true. If the data packet's congestion control sequence number is greater than <i>ack_cc_seq</i> , then set <i>ack_cc_seq</i> to the data packet's congestion control sequence number and set <i>prev_intersend_time</i> to the stored inter-send time in <i>unacked_pkts</i> in order to set τ_p . Increment <i>num_pkts_acked</i> by one. Call into any configured policy controller to update <i>delta</i> (δ) using the <i>UpdateDelta(false)</i> state machine.	
OnAckPktProcessingDone	See state machine below.	
OnPacketSent	Assign the next congestion control sequence number <i>nxt_cc_seq</i> to the data packet and increment <i>nxt_cc_seq</i> by one. Add an element to the <i>unacked_pkts</i> array for the data packet, setting the element's sequence number, send time, the	

Event	Copa Beta 1 Processing Details
	current <i>calc_intersend_time</i> value, the ACKed flag to false, the resent flag to false, and the skip until resent flag to false. Force the current <i>rtt_unacked</i> to be equal to the current <i>rtt_acked</i> . Do the UpdateUnackedRttEstimate state machine. If <i>num_pkts_acked</i> is greater than or equal to NUM_PROBES, then do the UpdateIntersendTime state machine. Do the UpdateNextSendTime(bytes) state machine. Increment <i>send_cnt</i> by one.
OnPacketResent	If this is the same congestion control algorithm that allowed the data packet to be sent and the data packet's congestion control sequence number is greater than or equal to <i>una_cc_seq</i> or less than <i>nxt_cc_seq</i> , then look up the data packet in the <i>unacked_pkts</i> array and update its send time, update its current <i>calc_intersend_time</i> value, and set its resent flag to true. Force the current <i>rtt_unacked</i> to be equal to the current <i>rtt_acked</i> . Do the UpdateUnackedRttEstimate state machine. If <i>num_pkts_acked</i> is greater than or equal to NUM_PROBES, then do the UpdateIntersendTime state machine. If the resend is not due to an RTO event, then do the UpdateNextSendTime(bytes) state machine.
ReportUnaPkt	Ignored.
RequireFastRto	Returns false.
OnRto	Ignored.
OnOutageEnd	Loop through the <i>unacked_pkts</i> array from <i>una_cc_seq</i> to (<i>nxt_cc_seq</i> - 1) looking for the oldest element send time with the element's resent flag set to false. If an element is found, then the last known good inter-send time is the element's stored inter-send time value, otherwise the last known good inter-send time is 0.1 seconds. Loop through the <i>unacked_pkts</i> array from <i>una_cc_seq</i> to (<i>nxt_cc_seq</i> - 1) again, setting each element's inter-send time to the last known good inter-send time, and setting the element's skip until resent flag to true if its ACKed flag is false. Set <i>calc_intersend_time</i> and <i>intersend_time</i> to the last known good inter-send time. Set <i>prev_intersend_time</i> to twice the last known good inter-send time.
CanSend	Returns true if there is room for another data packet in the <i>unacked_pkts</i> array, or false if not.
CanResend	Returns true.
TimeUntilSend	If the current time <i>now</i> plus <i>timer_tolerance</i> is greater than or equal to <i>next_send_time</i> , then return 0 seconds, otherwise return (<i>next_send_time</i> - <i>now</i>) seconds.
SendPacingRate	Returns $((8 * (\text{NPS} + \text{PKT_OVERHEAD})) / \text{calc_intersend_time})$ bits per second.
SendRate	Returns $((8 * (\text{NPS} + \text{PKT_OVERHEAD})) / \text{calc_intersend_time})$ bits per second.
GetSyncParams	This call is used to send the delta value computed by the local policy controller to the other endpoint in order to synchronize the two delta values. If <i>mode</i> is equal to MAX_THROUGHPUT (i.e., the Copa Beta 1 M policy controller is in use), the endpoint is the client side, and <i>sync_param</i> is not equal to 0, then do all of the following steps, otherwise return false immediately. Set the returned parameter value to <i>sync_param</i> . Set <i>sync_param</i> to 0. Return true. Otherwise, return false.
ProcessSyncParams	This call is used to process a received delta value from the remote policy controller from the other endpoint in order to synchronize the two delta values. If <i>mode</i> is equal to MAX_THROUGHPUT (i.e., the Copa Beta 1 M policy controller is in use), the endpoint is the server side, and the received parameter value is not equal to 0, then do all of the following steps, otherwise return immediately. Set <i>prev_sync_time</i> to the current time. Convert the received parameter value to a delta value (between MIN_DELTA and MAX_DELTA). If the received delta value is not equal to <i>remote_sync_delta</i> , then set <i>remote_sync_delta</i> to the received delta value. If the previous conditional was true and the absolute value of the difference

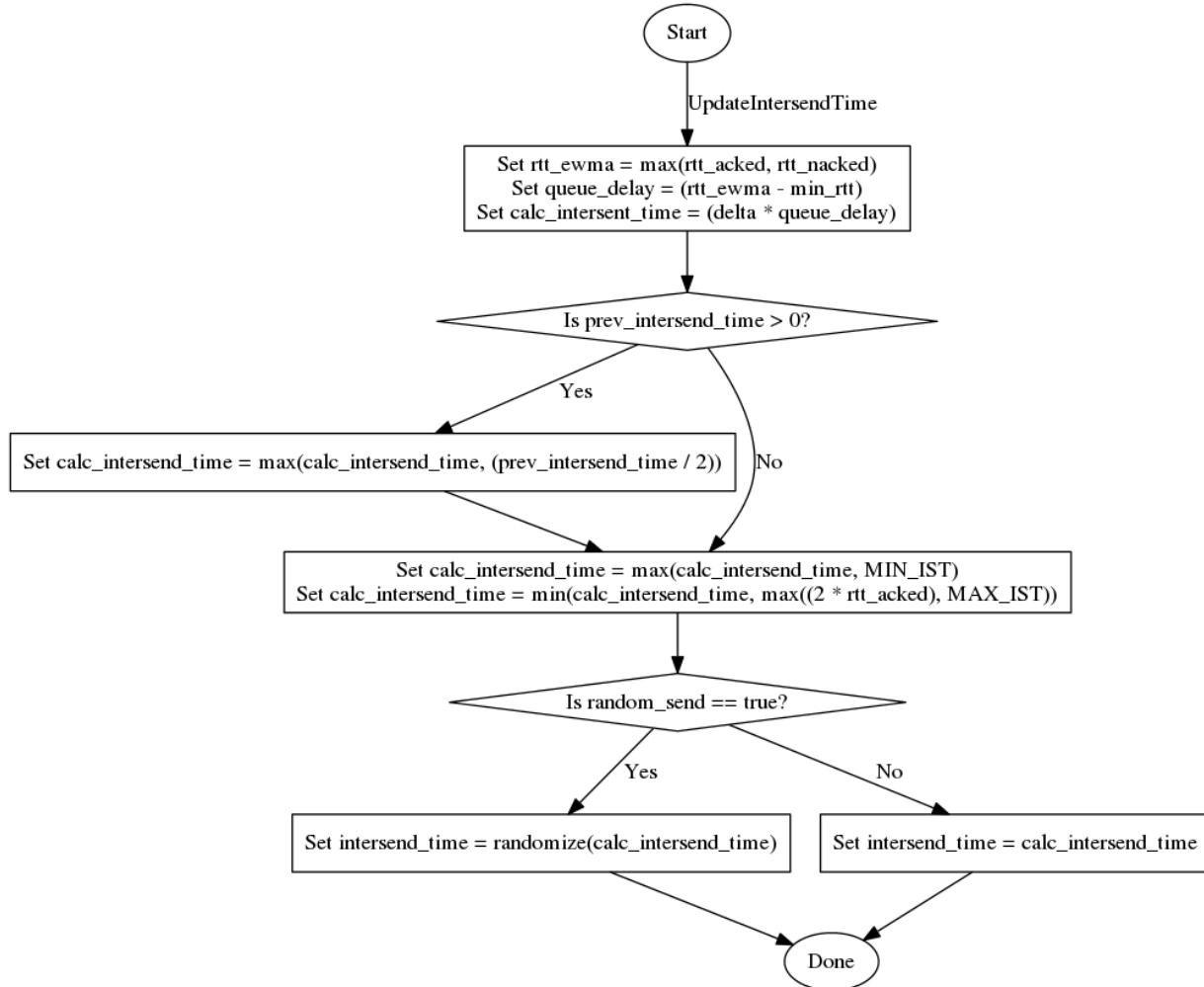
Event	Copa Beta 1 Processing Details
	between <i>remote_sync_delta</i> and <i>local_sync_delta</i> is less than or equal to <i>PC_SYNC_THRESH</i> , then set <i>delta</i> to <i>remote_sync_delta</i> .
ProcessCcPktTrain	Ignored.
InSlowStart	If <i>num_pkts_acked</i> is less than <i>NUM_PROBES</i> , then return true, otherwise return false.
InRecovery	Returns false.
GetCongestionWindow	Returns 0.
GetSlowStartThreshold	Returns 0.
Close	Ignored.

The OnAckPktProcessingDone state machine is shown below. This logic cleans up packet state for packets that are considered lost by moving *una_cc_seq* forward, and calls UpdateDelta to allow the policy controller to handle each packet loss. Note that the current Copa Beta 1 M policy controller does not utilize this packet loss information, but other future policy controllers might.

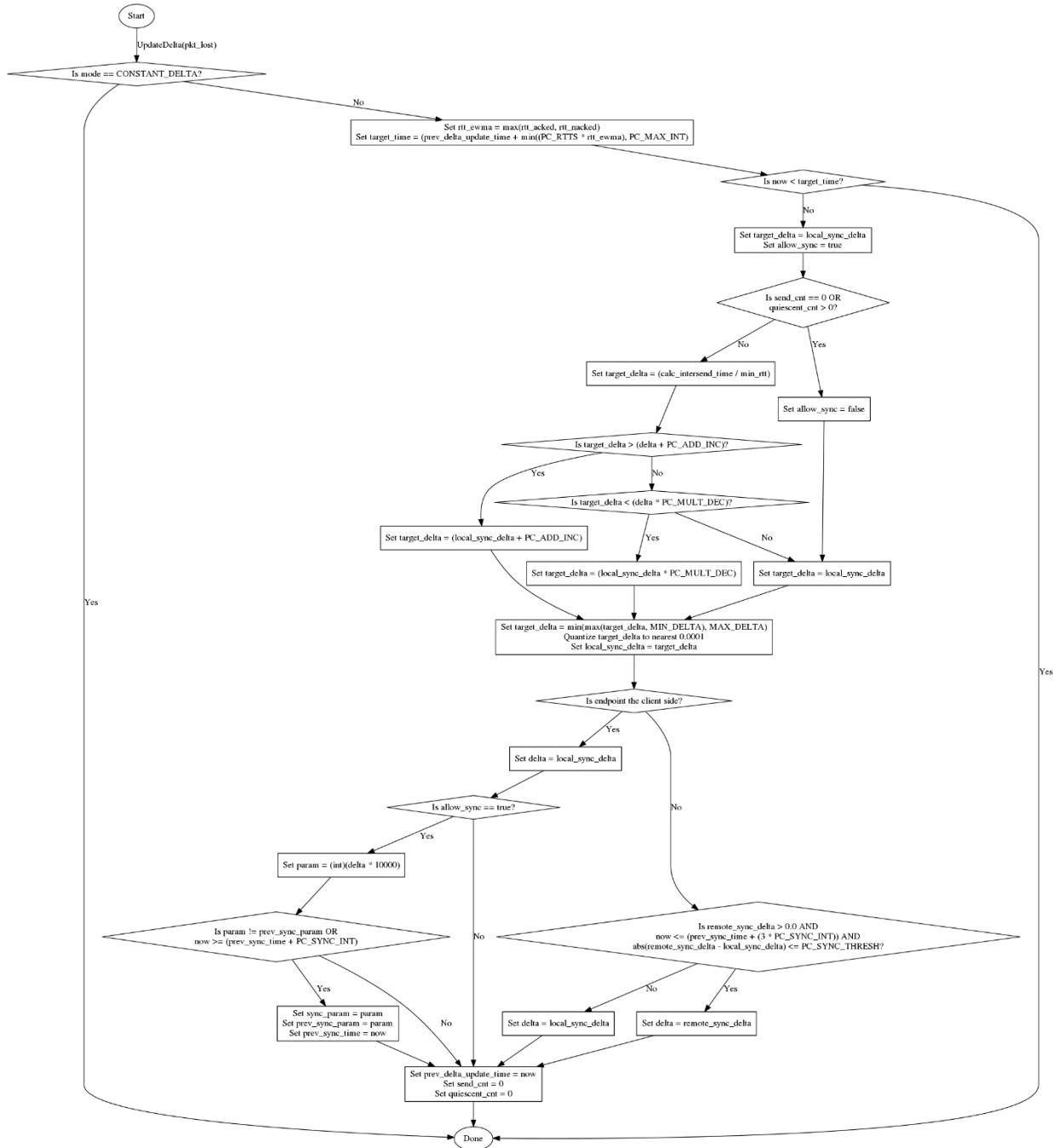


The UpdateIntersendTime state machine is shown below. This state machine is called by other state machines above, and updates the inter-send time. Note the upper and lower limits placed on

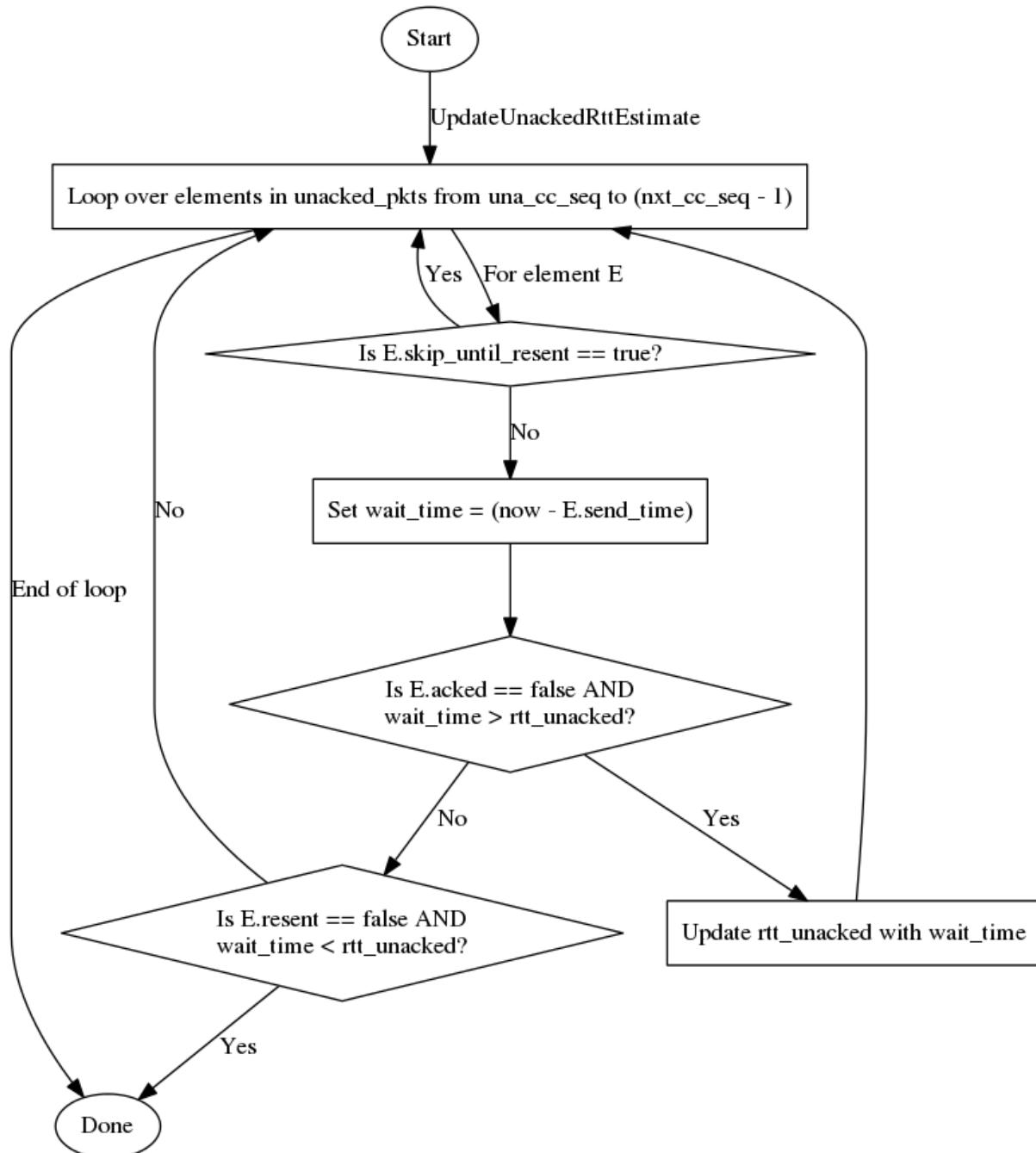
the calculated inter-send time to avoid unrealistic values and to avoid zero inter-send time values.



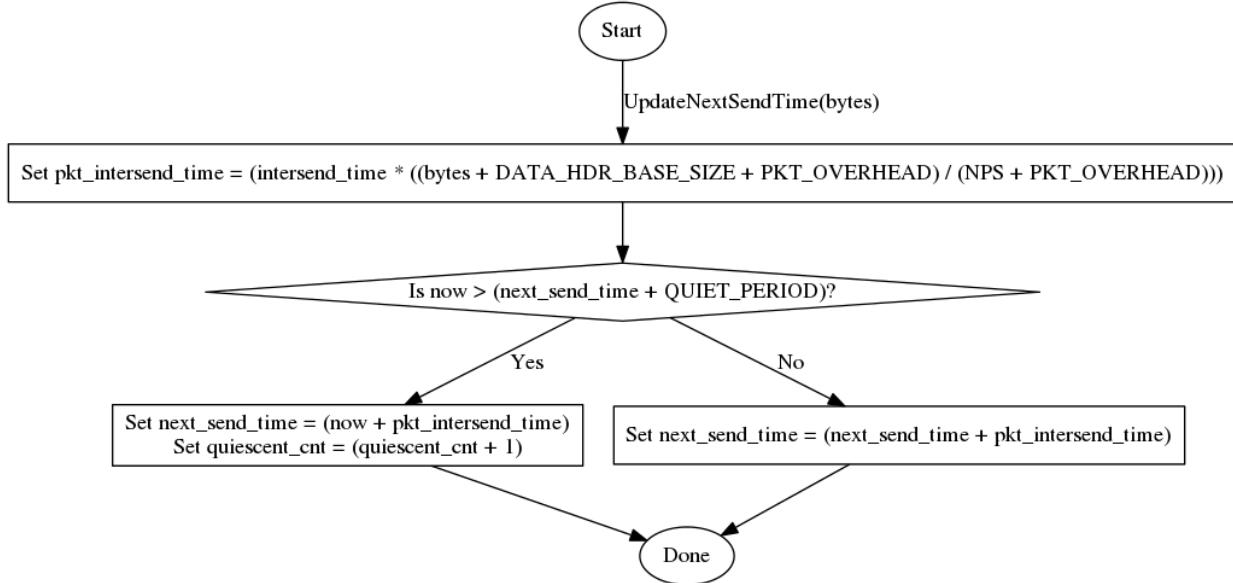
The UpdateDelta state machine is shown below. This state machine is called by other state machines above and implements the Copa Beta 1 M policy controller. It updates delta if a policy controller is currently in use. Note that the packet_lost flag passed into this state machine is not currently used. When enabled, the Copa Beta 1 M policy controller is run every 4 RTTs, or at least once per second. Much of the complexity in this state machine comes from the client side sending its computed delta value to the sever side periodically in order to improve bidirectional performance.



The UpdateUnackedRttEstimate state machine is shown below. This state machine is called by other state machines above, and updates the RTT estimate for packets that have not been ACKed in the expected amount of time (the advance RTT update mechanism). The two extra conditional statements skip elements that must be skipped until they are resent and break out of the loop when it is not possible to have a wait time that is greater than *rtt_unacked*.



The UpdateNextSendTime state machine is shown below. This state machine is called by other state machines above, and updates the next send time based on the number of bytes just sent. If the application does not attempt to send the next data packet within the QUIET_PERIOD, then the next_send_time must be jumped forward in order to prevent incorrect send pacing behavior.



7.3.2.3 Copa Beta 2

The Copa Beta 2 congestion control algorithm was also developed by MIT, and is the second beta version of Copa implemented by BBN. This algorithm is similar to Copa Beta 1 with the addition of a congestion window, the elimination of the policy controller, and the addition of a TCP compatible mode. With no option for a policy controller, Copa Beta 2 uses a fixed value for δ when not in TCP compatible mode. Copa Beta 2 also eliminates the option of randomizing the inter-send times, always using deterministic inter-send times. The Copa Beta 2 implementation includes enhancements based on testing and communications with MIT. However, it still has deficiencies that are addressed by the Copa congestion control algorithm. As such, the Copa congestion control algorithm should be used instead of this older version of the algorithm.

Copa Beta 2 uses both a send rate λ and has units of packets per second, and a congestion window, which is called *cwnd* and has units of packets, in order to limit how fast to send data packets and to control how many data packets may be outstanding (unACKed) at any time. Like Copa Beta 1, it uses the average switch delay, which is defined as the mean time spent by a packet in the bottleneck queue, as the measure of congestion. Each time that an RTT calculation is made due to the reception of an ACK packet, the Copa Beta 2 algorithm estimates the queueing delay d_q as:

$$d_q = RTT - RTT_{min}$$

where RTT is the RTT sample and RTT_{min} is the minimum RTT observed thus far, both in seconds. It then calculates the target send rate:

$$\lambda_t = 1/(\delta \cdot d_q)$$

and the current send rate:

$$\lambda = (cwnd/RTT)$$

where δ is the algorithmic parameter for aggressiveness, $cwnd$ is the current congestion window size in packets, and RTT is the RTT sample in seconds. Next, the current and target send rates are compared in order to adjust the congestion window size. If λ is less than or equal to λ_t , then $cwnd$ is increased:

$$cwnd' = cwnd + (v/(\delta \cdot cwnd))$$

where v is a velocity parameter. If λ is greater than, λ_t then $cwnd$ is decreased:

$$cwnd' = cwnd - (v/(\delta \cdot cwnd))$$

The velocity parameter, v , speeds up convergence of the congestion window, and is initialized to 1. Each time a $cwnd$ change is made, the direction of the change, either up or down, is noted. At the end of each RTT period, the number of $cwnd$ changes up and the number of $cwnd$ changes down are compared with the total number of $cwnd$ changes. If at least 2/3 of the changes are in one direction, then the direction of $cwnd$ change is considered to be in that direction. When there is a direction of change and the direction of $cwnd$ change is the same as it was during the previous RTT, v is doubled. Otherwise, it is reset to 1. In order to prevent v from being greater than 1 during normal $cwnd$ oscillations, the doubling of v is started only after the direction of $cwnd$ change has remained the same for three consecutive RTTs. To prevent excessive $cwnd$ growth, v is limited so that the send rate can never more than double once per RTT.

In order to send an original data packet with Copa Beta 2, the current number of bytes-in-flight must be less than $(1000 \cdot cwnd)$. The multiplication comes from the nominal packet size, which is 1000 bytes, which is used to convert Copa Beta 2's units of packets to bytes. When the ACK for a data packet is received, the $cwnd$ adjustments listed above are scaled by the ratio of the ACKed packet size in bytes divided by the nominal packet size in order to account for variable packet sizes. In order to prevent $cwnd$ from growing indefinitely when the application is continually sending data but at a rate lower than the bottleneck link rate, $cwnd$ increases are skipped when $cwnd$ is greater than twice the number of nominal packets in flight. Finally, all data packet transmissions and retransmissions are paced at a rate of $\lambda = cwnd/RTT$ nominal sized packets per second, which is then adjusted for the actual packet sizes.

Copa Beta 2 implements a fast startup phase in order to quickly adapt to the bottleneck link. The RTT estimate from the SLIQ connection handshake is used in order to send 11 nominally sized packets in each of the first two RTTs (22 packets total), but in back-to-back packet pairs. Since application data may not be ready for these packets, SLIQ uses Congestion Control Packet Train packets with a SLIQ header plus payload length of 1000 bytes for the 22 data packets, and Congestion Control Packet Train packets with an empty payload for the 22 ACK packets. By sending back-to-back packet pairs and observing the received ACKs for these packets, the bottleneck link

rate μ can be estimated. The ACKs for first packet pair are ignored, as testing has shown that the estimates from these packets can be very erratic. The ACKs for the following 10 packet pairs are used to estimate μ . RTT samples are calculated from the received ACKs for just the first packets in each of the 10 packet pairs, and the minimum (R_{min}) and maximum (R_{max}) of these RTT values are noted. The ACK inter-receive times, measured from the receipt of the first ACK until the receipt of the second ACK, are calculated for each of the 10 packet pairs, and converted into rate estimates using the calculation:

$$\mu_i = 1/irt_i$$

where irt_i is the ACK inter-receive time for a packet pair in seconds, and μ_i is the estimated bottleneck link rate in packets per second. The mean of these rate estimates is then μ , the bottleneck link rate estimate in packets per second. The initial send rate to use once fast startup is over is then:

$$\lambda = \min(\mu, (2/(\delta \cdot (R_{max} - R_{min}))))$$

Copa Beta 2 includes a TCP compatible mode, which allows it to compete with TCP flows. Since TCP uses buffer-filling, loss-based congestion control algorithms, the Copa Beta 2 algorithm detailed above will end up losing to TCP due to the bottleneck link queues being kept full most of the time. To counter this, Copa Beta 2 has two distinct modes of operation:

- The *default mode*, where $\delta = 0.5$, and
- A *TCP mode*, where δ is adjusted dynamically to match TCP's aggressiveness.

Copa Beta 2 switches between these two modes depending on whether or not it detects a competing, long-running TCP flow. If, at any time t , the sender detects a nearly-empty queue in $(t, t - (2 \cdot RTT))$ and also in $(t - (2 \cdot RTT), t - (4 \cdot RTT))$, it uses the default mode. Otherwise, it uses the TCP mode. Copa Beta 2 considers a nearly-empty queue as any queueing delay d_q lower than 10% of the maximum queueing delay observed in the last four RTTs. When in TCP mode, delta is varied using any TCP-style algorithm desired. The one chosen for implementation is an Additive Increase Multiplicative Decrease (AIMD) rule on $1/\delta$, where $1/\delta$ is increased by 1 for an RTT period where there are no lost packets, and where $1/\delta$ is multiplied by 0.5 whenever there is a lost packet within an RTT period. Due to complications with getting TCP mode to work reliably in a wide range of network settings as well as bad interactions between TCP mode and the selective damper (described below), the Copa Beta 2 implementation currently only uses default mode. The rest of this section does not include the TCP mode since it is not currently enabled.

Copa Beta 2 naturally can automatically track decreases to RTT_{min} . In order to handle cases where RTT_{min} increases, possibly due to a route change or some other network event, an additional mechanism is needed. The IRON team has come up with a relatively simple way to achieve this. First, the minimum observed RTT within a sampling period is tracked over 8 sampling periods, where the sampling period is the maximum of either the current RTT or 12 milliseconds in order to get enough RTT samples to work with. At the end of a sampling period, the

oldest minimum RTT sample and the lowest minimum RTT sample from all of the other 7 samples are found. Next, the potential minimum RTT difference is calculated as the lowest minimum RTT sample found minus the current RTT_{min} . If the oldest minimum RTT sample is either greater than RTT_{min} plus 60% of the potential minimum RTT difference or less than RTT_{min} plus 40% of the potential minimum RTT difference, then the oldest minimum RTT sample is compared with the lowest minimum RTT sample and the lower value becomes the candidate minimum RTT. Otherwise, the oldest minimum RTT sample is disregarded, as it is a "half-step" RTT change, where a data packet was sent to the destination before the network event and the ACK packet was send back after the network event, and the lowest minimum RTT sample becomes the candidate minimum RTT. Finally, if the candidate minimum RTT is at least 10% greater than RTT_{min} and higher than RTT_{min} by 0.5 milliseconds, then the candidate minimum RTT becomes the new RTT_{min} value. Otherwise, RTT_{min} is not changed.

During testing of this implementation of Copa Beta 2 over networks with very high latencies ($RTT \geq 0.2$ seconds), it was found that large, slow oscillations in the send rate can occur. These oscillations tend to occur when the fast startup phase results in a poor estimate of the proper send rate. When these oscillations do occur, they are not effectively damped by the algorithm and can continue forever. The result is the bottleneck queue being filled with far too many packets for long periods of time, alternating with shorter periods where the send rate is far too low. These conditions lead to extremely high packet delivery latencies followed by an under-utilization of the channel. To avoid this situation, the IRON team has added a selective damper to Copa Beta 2. This damper has four states that it uses: *MonitorHigh*, *MonitorLow*, *Hold*, and *Wait*. It starts in the *MonitorHigh* state, where it is monitoring the RTT calculations from ACK packets until $d_q \cdot \lambda > 200$, which is when more than 200 packets are queued in the bottleneck link. This is the trigger that a large oscillation is occurring, and it causes the damper to transition to the *MonitorLow* state. In the *MonitorLow* state, it monitors the RTT calculations from ACK packets until $d_q \cdot \lambda < 1/\delta$, which is when just less than the desired number of packets are queued in the bottleneck link. At this point, the data packet that is being ACKed was sent when the congestion window size was optimal. Thus, it looks up the *cwnd* that was in use when the data packet that generated the RTT calculation was sent, sets *cwnd* to this value, and transitions into the *Hold* state. The *Hold* state lasts for one RTT and prevents any changes to *cwnd* during this time. This is an attempt to send at the proper rate while waiting for the ACKs for these data packets to start arriving. After the RTT period is over, it transitions into the *Wait* state. The *Wait* state lasts for one RTT while the *cwnd* is allowed to be changed and the damper does nothing. After the RTT period is over, it transitions into the *MonitorHigh* state and starts over. In testing the selective damper, the large, slow oscillations typically disappear after one complete cycle through the states, allowing Copa Beta 2 to operate normally afterward.

The following constants are used by Copa Beta 2:

- DEFAULT_DELTA - The default delta value. Set to 0.5.
- NPS - The nominal packet size, which is set to 1000 bytes. Used to adapt the computations from units of packets to bytes, in order to support variable packet sizes.
- DATA_HDR_BASE_SIZE - The base size of the SLIQ data header in bytes. Set to 16 bytes.

- **PKT_OVERHEAD** - The nominal packet overhead size, which is set to 54 bytes (26 bytes for Ethernet, 20 bytes for IP, and 8 bytes for UDP).
- **CONN_RTT_ADJ** - The SLIQ connection handshake RTT adjustment amount. Set to 0.025 seconds.
- **QUIET_PERIOD** - The maximum amount of time that a paced send can be late without the sender considered to be quiescent. Set to 0.01 seconds.
- **MRT_MIN_PERIOD** - The minimum update period for the minimum RTT tracking mechanism. Set to 0.012 seconds.
- **HUGE_RTT** - The RTT value used when the RTT is not to be used. Set to 3600.0 seconds.
- **FS_PAIRS** - The number of fast startup packet pairs. Set to 11.
- **MRT_NUM_PERIODS** - The number of sampling periods that the minimum RTT tracking mechanism watches. Set to 8.
- **MRT_NUM_IST** - The number of inter-send times that the minimum RTT tracking mechanism stores for use. Set to 10.
- **MRT_RATIO** - The necessary increase ratio for the minimum RTT tracking mechanism to accept the change. Set to 1.1.
- **MRT_AMOUNT** - The necessary amount for the minimum RTT tracking mechanism to accept the change. Set to 0.0005 seconds.
- **MRT_RESET_THRES** - The RTT threshold value below which the minimum RTT tracking mechanism will perform a reset versus an adjustment. Set to 0.080 seconds.
- **FAST_RTO_CWND_THRES** - The fast RTO congestion window threshold value in packets. Set to 32.0.
- **MIN_CWND** - The minimum congestion window size in packets. Set to 2.
- **DAMPER_THRES** - The number of packets in the bottleneck queue that initiates the selective damper. Set to 200.

The following state variables are maintained by Copa Beta 2:

- *state* - The Copa Beta 2 operating state. Either NOT_CONNECTED, FAST_STARTUP, or CLOSED_LOOP. Initialized to NOT_CONNECTED.
- *fast_startup* - The fast startup information, including the number of packet pairs sent, an array of packet pair send times, an array of packet pair receive times, an array of RTT estimates, an array of channel rate estimates, and a timer. All state is initialized to zero.
- *min_rtt_tracking* - The minimum RTT tracking information, including the recent minimum RTT value, an array of minimum RTT values for each sampling period with the count of elements and next element index, an array of inter-send times with the count of elements and next element index, and the previous state update time. All state is initialized to zero.
- *damper* - The selective damper information, including the current damper state (MONITOR_HIGH, MONITOR_LOW, HOLD, or WAIT), and a hold count. Initialized to MONITOR_HIGH and zero.
- *delta* - The algorithmic parameter for aggressiveness (δ). Initialized to 0.5.
- *last_rtt* - The last RTT measurement in seconds. Initialized to HUGE_RTT.
- *min_rtt* - The minimum RTT in seconds (RTT_{min}). Initialized to HUGE_RTT.
- *cwnd* - The congestion window size in packets. Initialized to 3.0.

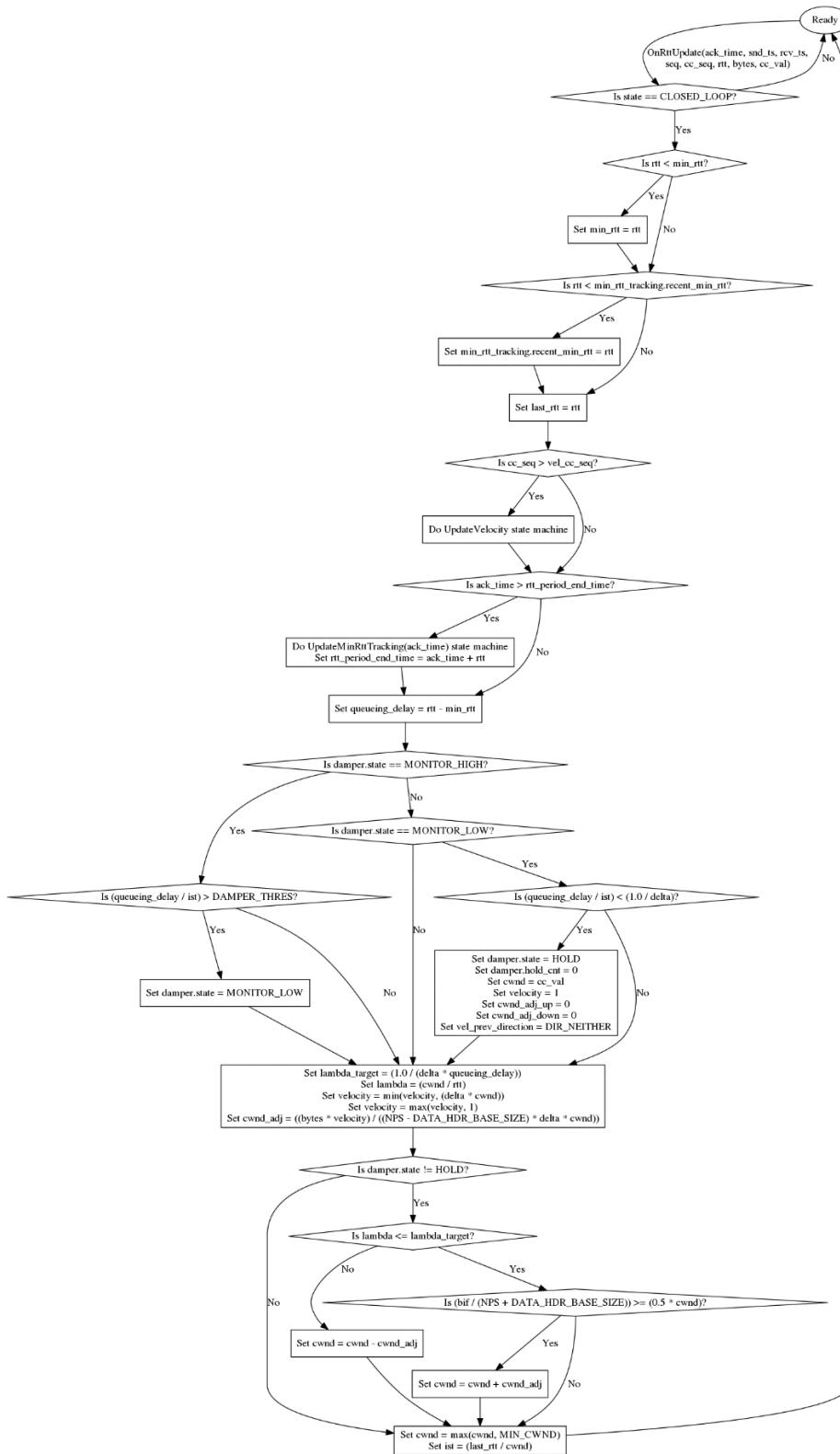
- *ist* - The packet inter-send time in seconds ($1/\lambda$). Initialized to 1.0.
- *velocity* - The congestion window adjustment velocity parameter. Initialized to 1.
- *cwnd_adj_up* - The number of congestion window adjustments up. Initialized to 0.
- *cwnd_adj_down* - The number of congestion window adjustments down. Initialized to 0.
- *vel_prev_direction* - The congestion window adjustment direction from the previous RTT period. Maybe be DIR_NEITHER, DIR_UP, or DIR_DOWN. Initialized to DIR_NEITHER.
- *vel_same_direction_cnt* - The number of velocity adjustments in the same direction. Initialized to 0.
- *vel_cc_seq* - The congestion control sequence number at the start of the current velocity update period. Initialized to 0.
- *nxt_cc_seq* - The next congestion control sequence number to be sent. Initialized to 0.
- *rtt_period_end_time* - The end time of the current RTT period. Initialized to the current time.
- *next_send_time* - The time that the next data packet can be sent. Initialized to the current time.
- *timer_tolerance* - The expected accuracy of the SLIQ event timer in seconds. Initialized to 0.001 seconds.

Copa Beta 2 implements the congestion control API as shown in the following table and state machine diagrams.

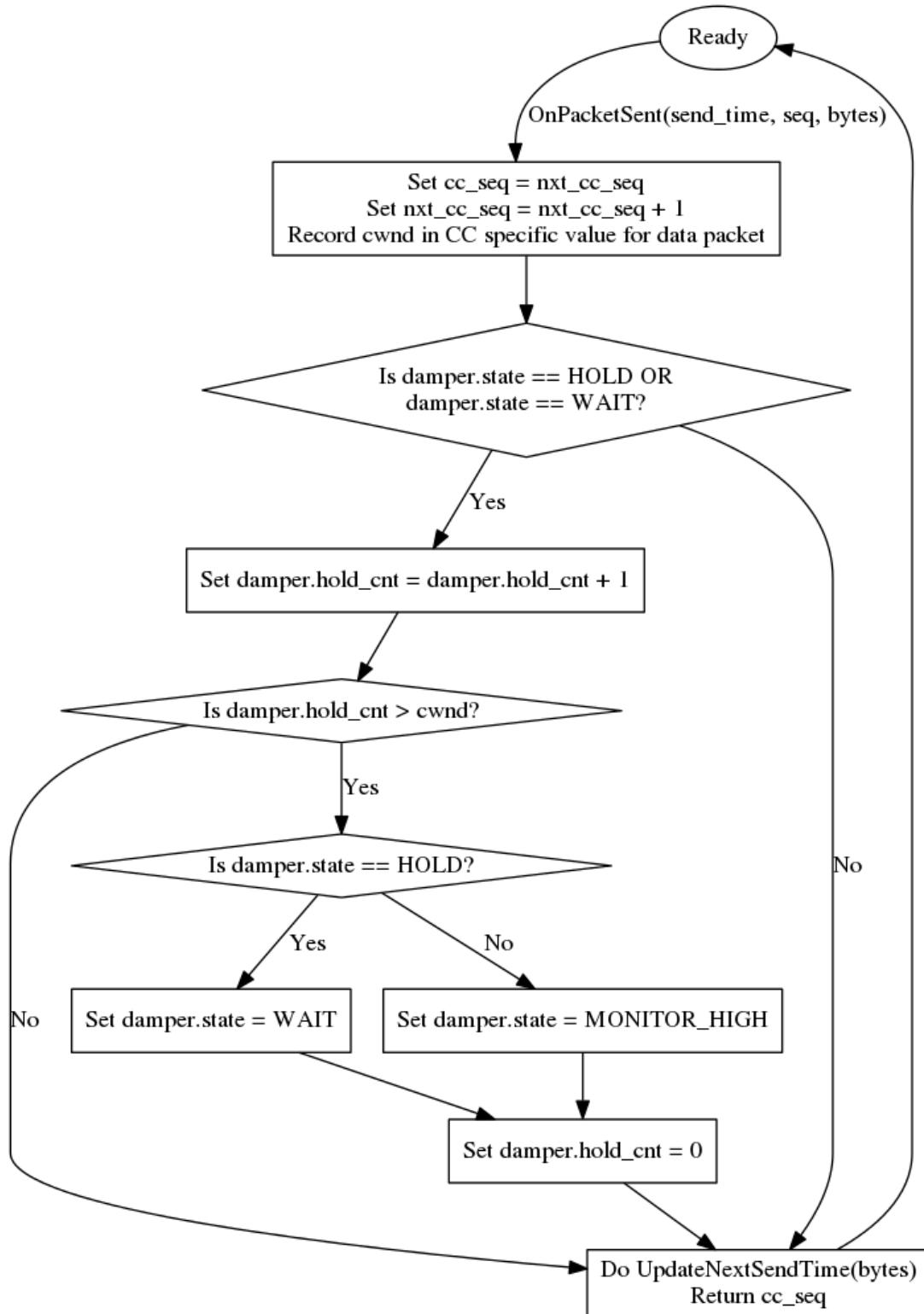
Event	Copa Beta 2 Processing Details
Configure	Initialize all state variables as listed above.
Connected	If <i>state</i> is not equal to NOT_CONNECTED, then log an error and return immediately. Set <i>state</i> to FAST_STARTUP. Set <i>last_rtt</i> to the SLIQ connection handshake RTT plus CONN_RTT_ADJ. Set <i>min_rtt</i> to the SLIQ connection handshake RTT. Clear all of the <i>fast_startup</i> state. Initiate the sending of the FS_PAIRS data packet pairs ($2 * \text{FS_PAIRS}$ Congestion Control Packet Train packets) with the packet train packet type set to 0 and containing packet train sequence numbers from 0 to $((2 * \text{FS_PAIRS}) - 1)$ for fast startup. Send the first packet pair immediately and start a timer to send the other packet pairs every $(2 * \text{last_rtt} / \text{FS_PAIRS})$ seconds. Once the last packet pair is sent, start a timer to call FsDoneCallback after $(2 * \text{FS_PAIRS} * \text{last_rtt})$ seconds, or a maximum of 1 second.
UseRexmitPacing	Returns true.
UseCongWinForCapEst	Returns true.
UseUnaPktReporting	Returns false.
SetTcpFriendliness	Ignored.
ActivateStream	Ignored.
DeactivateStream	Ignored.
OnAckPktProcessingStart	Ignored.
OnRttUpdate	See state machine below.
OnPacketLost	Returns true.
OnPacketAcked	Ignored.
OnAckPktProcessingDone	Ignored.
OnPacketSent	See state machine below.
OnPacketResent	See state machine below.
ReportUnaPkt	Ignored.
RequireFastRto	Since Copa Beta 2 implements a congestion window and can operate with non-

Event	Copa Beta 2 Processing Details
	congestion packet losses, fast RTOs are required when the congestion window size is small. If $cwnd$ is less than FAST_RTO_CWND_THRES, then return true, otherwise return false.
OnRto	Ignored.
OnOutageEnd	Ignored.
CanSend	If $state$ is equal to CLOSED_LOOP and $(cwnd * (NPS - DATA_HDR_BASE_SIZE))$ is greater than bif , then return true, otherwise return false.
CanResend	Returns true.
TimeUntilSend	If the current time now plus $timer_tolerance$ is greater than or equal to $next_send_time$, then return 0 seconds, otherwise return $(next_send_time - now)$ seconds.
SendPacingRate	Returns $((8 * (NPS + PKT_OVERHEAD)) / ist)$ bits per second.
SendRate	Returns $((8 * (NPS + PKT_OVERHEAD)) / ist)$ bits per second.
GetSyncParams	Returns false.
ProcessSyncParams	Ignored.
ProcessCcPktTrain	See state machine below.
InSlowStart	If $state$ is not equal to CLOSED_LOOP, then return true, otherwise return false.
InRecovery	Returns false.
GetCongestionWindow	Returns $(cwnd * (NPS - DATA_HDR_BASE_SIZE))$ bytes.
GetSlowStartThreshold	Returns 0.
Close	Ignored.

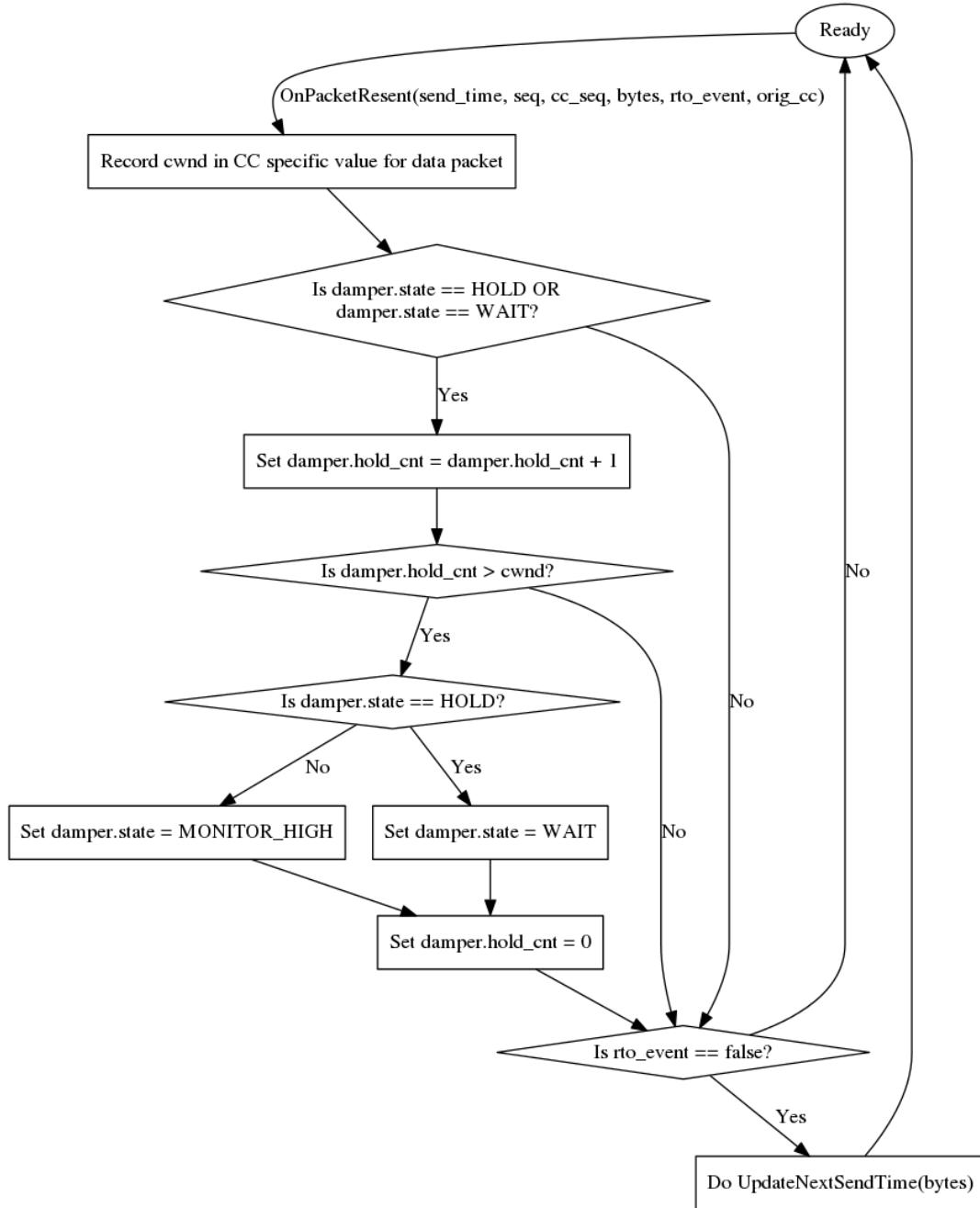
The OnRttUpdate state machine is shown below. It updates the RTT state information, updates the velocity if needed, updates the minimum RTT tracker if needed, updates the congestion window, and calculates the new packet inter-send time to use for pacing packet transmissions.



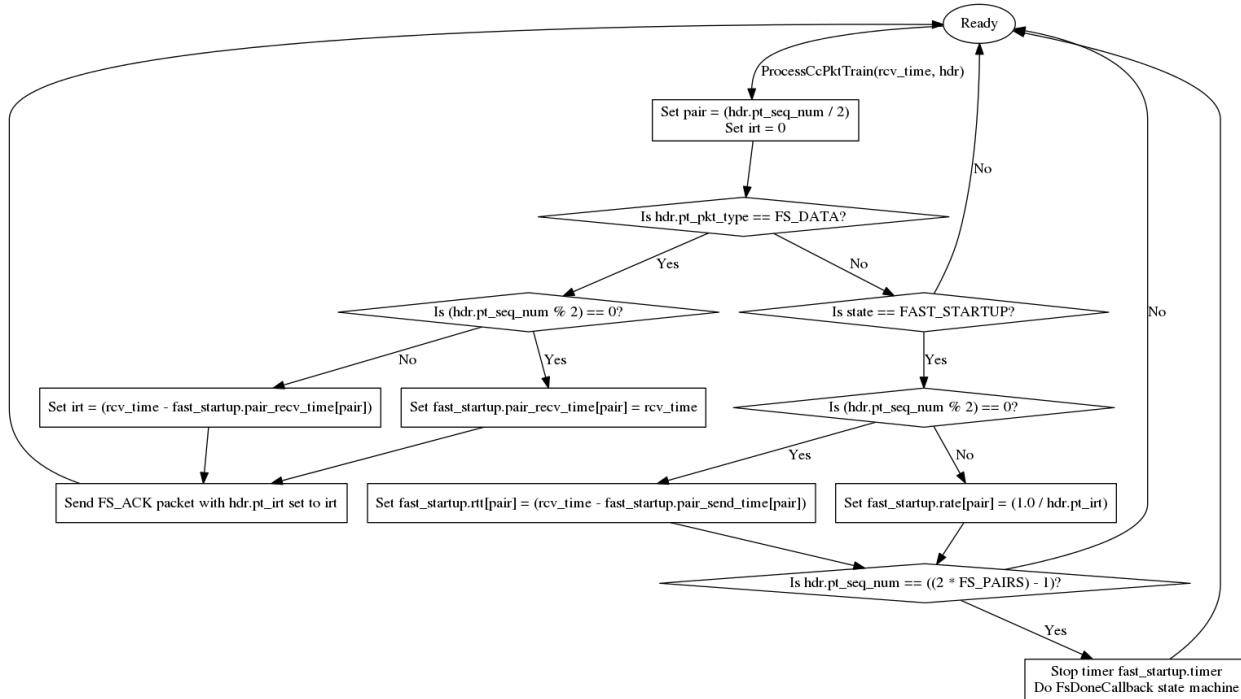
The OnPacketSent state machine is shown below. It assigns the next congestion control sequence number to the data packet being sent and records the current *cwnd* value in the congestion control specific state for the data packet. It also updates the selective damper state as needed.



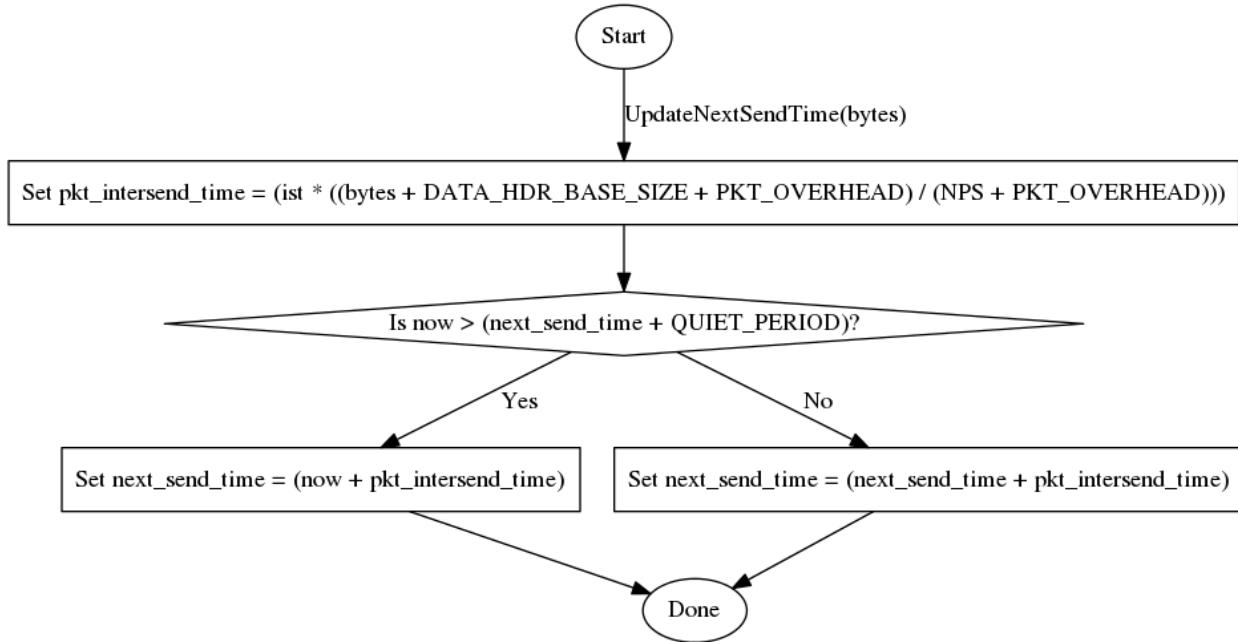
The OnPacketResent state machine is shown below. It updates the current *cwnd* value stored in the congestion control specific state for the data packet, and it updates the selective damper state as needed. If the resend is not due to an RTO event, then it calls the UpdateNextSendTime state machine passing the size of the data packet in bytes.



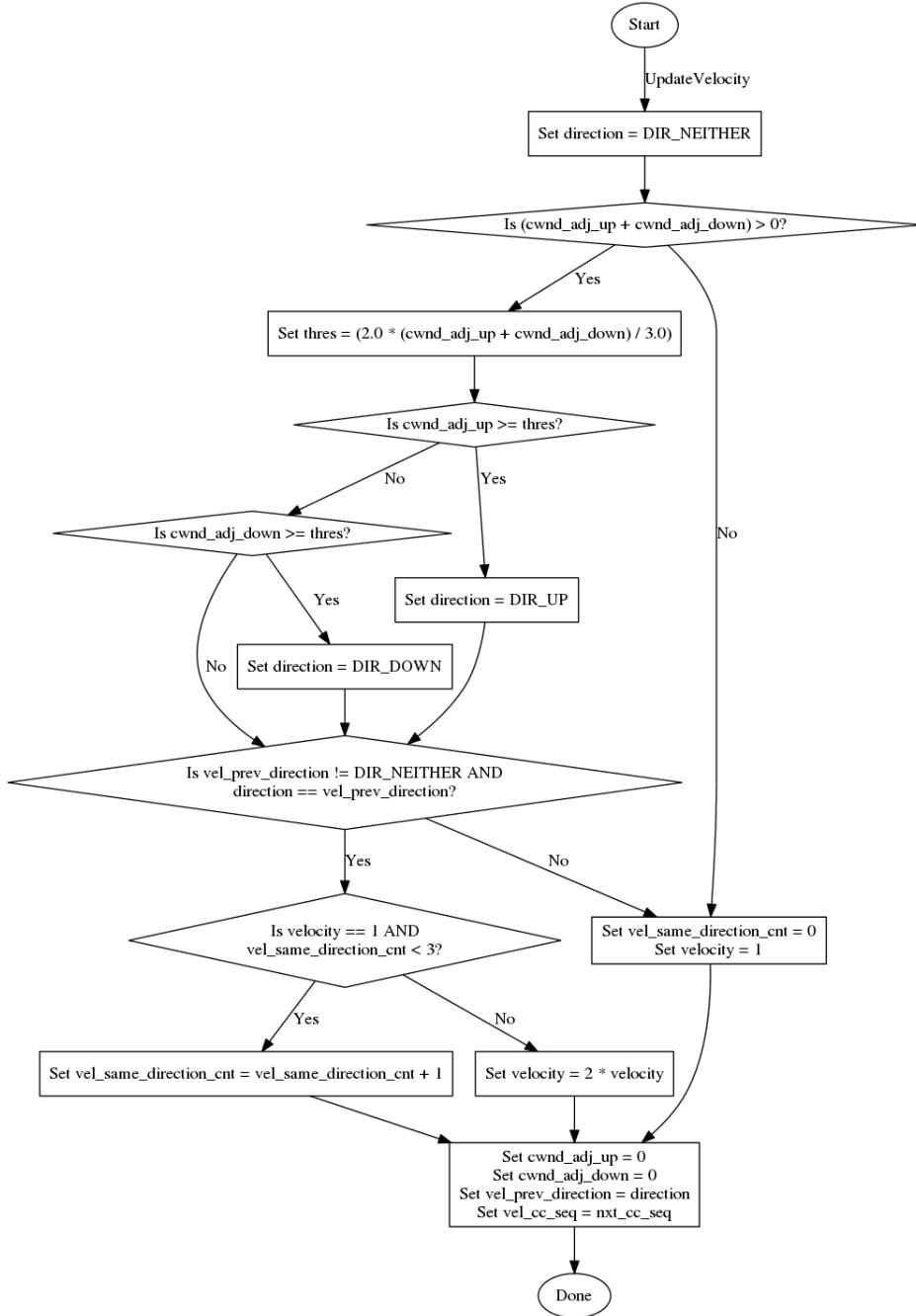
The ProcessCcPktTrain state machine is shown below. This logic handles the received SLIQ Congestion Control Packet Train data packets (packet train packet type 0) during fast startup and generates the necessary SLIQ Congestion Control Packet Train ACK packets (packet train packet type 1). Data packet pairs with even packet train sequence numbers cause the receive time to be recorded and a packet pair ACK packet to be sent back, while those with odd sequence numbers cause the inter-receive time to be computed and sent back in a packet pair ACK packet. Packet pair ACK packets with even sequence numbers cause the RTT measurement to be recorded, while those with odd sequence numbers cause the rate estimate to be recorded. If the last packet pair ACK packet (with packet train sequence number 21) is received, then the timer is stopped and FsDoneCallback is called immediately.



The UpdateNextSendTime state machine is shown below. This state machine is called by other state machines above, and updates the next send time based on the number of bytes just sent. If the application does not attempt to send the next data packet within the QUIET_PERIOD, then the next_send_time must be jumped forward in order to prevent incorrect send pacing behavior.

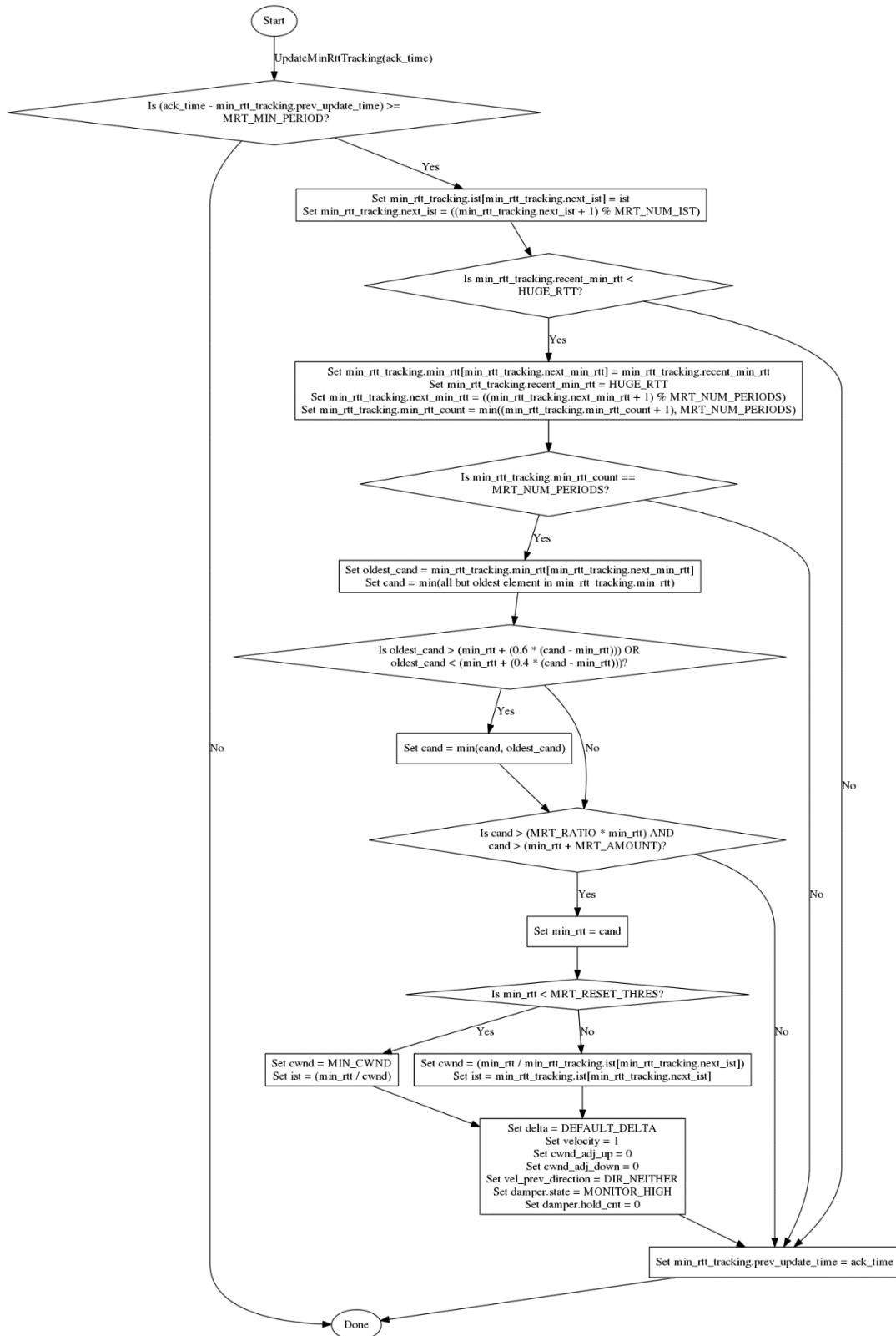


The UpdateVelocity state machine is shown below. It is called by the other state machines above. This logic updates the velocity parameter.

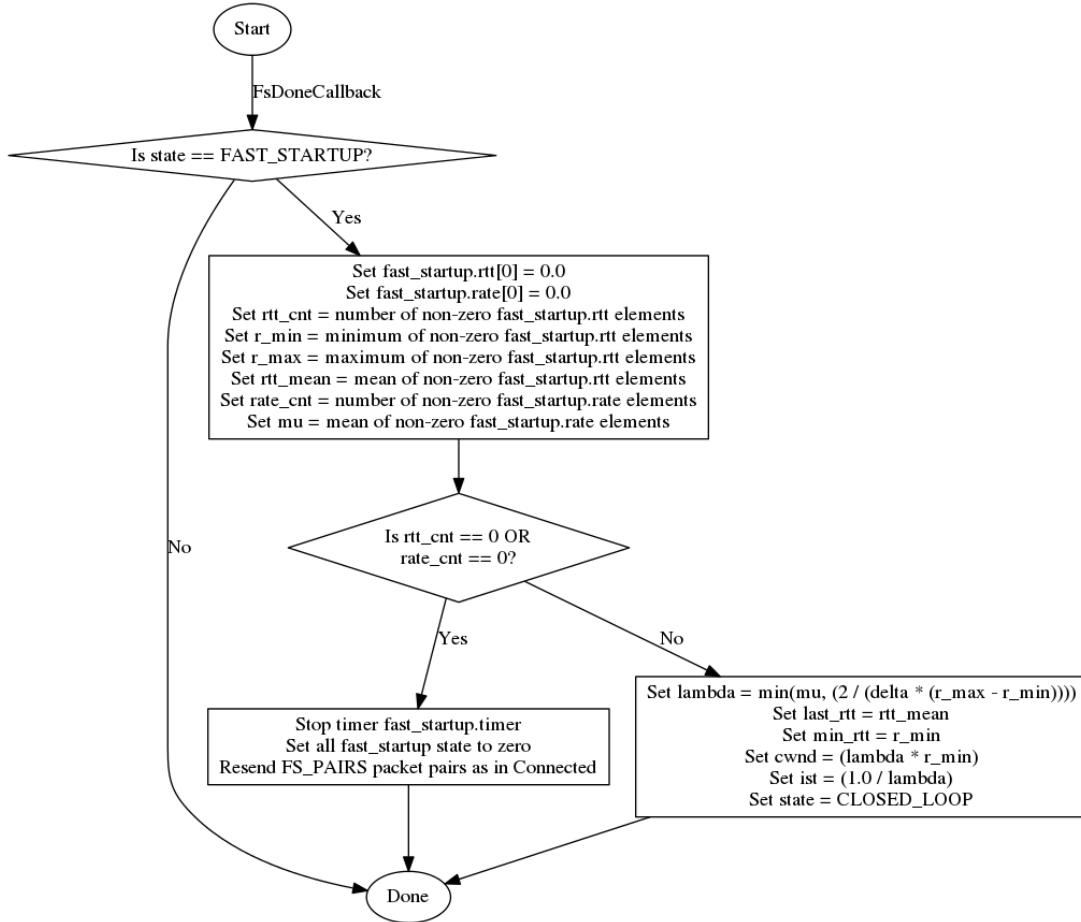


The UpdateMinRttTracking state machine is shown below. It is called by the other state machines above. This logic determines when there has been an increase in the minimum RTT on the network and updates the minimum RTT accordingly. Note that it avoids the "half-step" RTT change issue described above. When the minimum RTT is changed, if the network latency is low, then it simply resets the *cwnd* and *ist* parameters since Copa Beta 2 will adapt quickly to the new network conditions. However, if the network latency is high when the minimum RTT is

changed, then it adjusts the existing *cwnd* and *ist* parameters since resetting them will cause a long delay while Copa Beta 2 adapts to the new network conditions.



The FsDoneCallback state machine is shown below. It is called by the other state machines above. It erases the first RTT and rate estimate data, then processes the fast startup RTT and rate estimates into starting values for the *last_rtt*, *min_rtt*, *cwnd*, and *ist* state. If there were not enough fast startup estimates gathered, then the fast startup procedure is restarted.



7.3.2.4 Copa

The Copa congestion control algorithm is the final version of Copa developed by MIT, and is an evolution of the Copa Beta 2 algorithm described above. This algorithm is similar to Copa Beta 2 with the addition of a standing RTT for combating network latency jitter, a simpler minimum RTT tracking approach, improved rules for updating the velocity, and a different startup approach. Like Copa Beta 2, Copa uses a fixed value for the algorithmic aggressiveness parameter δ . Although MIT has specified a TCP compatible mode for this version of Copa, it has not been implemented yet and is not included in this documentation. In order to provide compatibility with TCP, this Copa congestion control algorithm should be used alongside the CUBIC congestion control algorithm for a SLIQ connection. The Copa implementation includes enhancements based on testing and communications with MIT. This version of Copa should be used instead of the Copa Beta 1 and Copa Beta 2 versions due to known deficiencies with these two earlier algorithms.

Copa uses both a send rate λ for controlling how fast data packets are sent, and a congestion window, $cwnd$, to control how many data packets may be outstanding (unACKed) at any time. It uses the average switch delay, which is defined as the mean time spent by a packet in the bottleneck queue, as the measure of congestion. Each time that an RTT calculation is made due to the reception of an ACK packet, the Copa algorithm updates three different RTT parameters. The RTT calculation itself utilizes the timestamp values within the SLIQ headers in order to eliminate the reverse path queueing delay from the RTT measurements as follows.

- The timestamp delta, $delta_{ts}$, is calculated for each ACK packet received by subtracting the ACK header packet timestamp (which was the remote timestamp when the ACK packet was sent) from the ACK packet's local receive timestamp.
- The timestamp delta values are tracked over time in order to maintain the smallest timestamp delta value, $delta_{min}$, observed over the last τ time window. The value of τ is equal to either $7 \cdot RTT_{min}$ (explained below) or 0.2 seconds, whichever is larger. The resulting $delta_{min}$ value is an estimate of the timestamp difference when the reverse path queueing delay is zero.
- The raw RTT value, RTT_{raw} , for a data packet being ACKed is calculated by subtracting the ACK packet's observed packet timestamp for the data packet (which was the local timestamp when the data packet was sent, updated for any hold time at the peer) from the ACK packet's local receive timestamp.
- The RTT measurement for the data packet being ACKed is then:

$$RTT = RTT_{raw} + delta_{min} - delta_{ts}$$

Thus, the RTT measurement includes the transmission delay, the round-trip propagation delay, and the forward path queueing delay at the bottleneck link.

The first RTT parameter to be updated is a standard smoothed RTT estimate, called $srtt$, that is initialized to the SLIQ connection handshake RTT estimate and then updated with each RTT measurement:

$$srtt' = ((\alpha \cdot RTT) + ((1 - \alpha) \cdot srtt)), \alpha = 1/16$$

The second RTT parameter is the standing RTT, $RTT_{standing}$, which is the smallest RTT measurement observed within the last $srtt/2$ time window. It represents the RTT corresponding to the "standing" queue in the forward path bottleneck link, and eliminates the effects of any high-frequency jitter in the RTT measurements.

The third RTT parameter is the minimum RTT, RTT_{min} , which is the smallest RTT measurement observed within the last τ time window. The value of τ is equal to either $28 \cdot RTT_{min}$ or 0.8 seconds, whichever is larger. By using a sliding window for setting RTT_{min} , Copa can handle latency increases due to network events. In order to achieve symmetric results in bidirectional tests, the RTT_{min} values are exchanged in SLIQ Congestion Control Synchronization packets, and each endpoint uses the smaller of the local and remote values as the RTT_{min} .

Copa then estimates the queueing delay in the forward path bottleneck link, d_q , as:

$$d_q = RTT_{standing} - RTT_{min}$$

It then calculates the target send rate, in packets per second, as:

$$\lambda_t = 1/(\delta \cdot d_q)$$

where $\delta = 0.5$, and the current send rate, in packets per second, as:

$$\lambda = cwnd/RTT_{standing}$$

Next, the current and target send rates are compared in order to adjust the congestion window, which has units of packets. If λ is less than or equal to λ_t , then $cwnd$ is increased:

$$cwnd' = cwnd + (v/(\delta \cdot cwnd))$$

If λ is greater than λ_t , then $cwnd$ is decreased:

$$cwnd' = cwnd - (v/(\delta \cdot cwnd))$$

The velocity parameter, v , is used to speed up convergence of the congestion window. It is initialized to 1. Once per congestion window, the current $cwnd$ value is compared to the $cwnd$ value at the start of the window. If the current $cwnd$ is larger, then the direction of change is "up". If the current $cwnd$ is smaller, then the direction of change is "down". If the current direction is the same as in the previous window, then v is doubled, otherwise v is reset to 1. Additionally, the doubling of v only begins after the direction has remained the same for three consecutive windows. This keeps the value of v at 1 during steady state operation. To prevent excessive $cwnd$ growth, v is limited so that the send rate can never more than double once per window.

In order to send an original data packet with Copa, the current number of bytes-in-flight must be less than $(1000 \cdot cwnd)$. The multiplication comes from the nominal packet size, which is 1000 bytes, and is used to convert Copa's units of packets to bytes. When the ACK for a data packet is received, the $cwnd$ adjustments listed above are scaled by the ratio of the ACKed packet size in bytes divided by the nominal packet size in order to account for variable packet sizes. In order to prevent $cwnd$ from growing indefinitely when the application is continually sending data but at a rate lower than the bottleneck link rate, $cwnd$ increases are skipped when $cwnd$ is greater than twice the number of nominal packets in flight. Finally, all data packet transmissions and retransmissions are paced at a rate of $\lambda = cwnd/RTT_{standing}$ nominal sized packets per second, which is then adjusted for the actual packet sizes. Originally, MIT specified the pacing rate as

$$\lambda = 2 \cdot cwnd/RTT_{standing}$$

nominal sized packets per second, but this higher rate caused issues with bidirectional Copa flows.

Copa implements two different startup approaches, slow-start and fast startup. Only one of the approaches is used during startup of a Copa flow. The approach is selected based on the SLIQ

connection handshake RTT estimate that Copa starts with. If the RTT estimate is smaller than some threshold, then slow-start is used, otherwise fast startup is used. This is done because slow-start can start sending application data immediately, while fast startup requires sending control packets to characterize the network before application data can be sent. However, it can take many RTTs for slow-start to converge at the correct send rate, which only makes sense to do if the RTT is small, while fast startup obtains all of its estimates within three RTTs. The threshold value used in the Copa implementation is currently 50 milliseconds.

When Copa selects slow-start, it doubles $cwnd$ once per RTT until λ exceeds λ_t . It does this by increasing $cwnd$ by the size of the ACKed packet divided by the nominal packet size for each ACK packet received. Once the current send rate exceeds the target send rate, slow-start is over and the normal $cwnd$ update rules are used.

When Copa selects fast startup, it characterizes the link using special packets. The RTT estimate from the SLIQ connection handshake is used in order to send 11 nominally sized packets in each of the first two RTTs (22 packets total), but in back-to-back packet pairs. Since application data may not be ready for these packets, SLIQ uses Congestion Control Packet Train packets with a SLIQ header plus payload length of 1000 bytes for the 22 data packets, and Congestion Control Packet Train packets with an empty payload for the 22 ACK packets. By sending back-to-back packet pairs and observing the received ACKs for these packets, the bottleneck link rate μ can be estimated. The ACKs for first packet pair are ignored, as testing has shown that the estimates from these packets can be very erratic. The ACKs for the following 10 packet pairs are used to estimate μ . RTT samples are calculated from the received ACKs for just the first packets in each of the 10 packet pairs, and the minimum (R_{min}) and maximum (R_{max}) of these RTT values are noted. The ACK inter-receive times, measured from the receipt of the first ACK until the receipt of the second ACK, are calculated for each of the 10 packet pairs, and converted into rate estimates using the calculation:

$$\mu_i = 1/irt_i$$

where irt_i is the ACK inter-receive time for a packet pair in seconds, and μ_i is the estimated bottleneck link rate in packets per second. The mean of these rate estimates is then μ , the bottleneck link rate estimate in packets per second. The initial send rate to use once fast startup is over is then:

$$\lambda = \min(\mu, (2/(\delta \cdot (R_{max} - R_{min}))))$$

During testing of this implementation of Copa over networks with very high latencies ($RTT \geq 0.2s$), it was found that large, slow oscillations in the send rate can occur. These oscillations tend to occur when the fast startup phase results in a poor estimate of the proper send rate. When these oscillations do occur, they are not effectively damped by the algorithm and can continue forever. The result is the bottleneck queue being filled with far too many packets for long periods of time, alternating with shorter periods where the send rate is far too low. These conditions lead to extremely high packet delivery latencies followed by an under-utilization of the channel. To avoid this situation, the ION team has added a selective damper to Copa. This damper has four states that it uses: *MonitorHigh*, *MonitorLow*, *Hold*, and *Wait*. It starts in the *MonitorHigh* state,

where it is monitoring the RTT calculations from ACK packets until $d_q \cdot \lambda > 40$, which is when more than 40 packets are queued in the bottleneck link. This is the trigger that a large oscillation is occurring, and it causes the damper to transition to the MonitorLow state. In the MonitorLow state, it monitors the RTT calculations from ACK packets until $d_q \cdot \lambda < 1/\delta$, which is when just less than the desired number of packets are queued in the bottleneck link. At this point, the data packet that is being ACKed was sent when the congestion window size was optimal. Thus, it looks up the *cwnd* that was in use when the data packet that generated the RTT calculation was sent, sets *cwnd* to this value, and transitions into the Hold state. The Hold state lasts for one RTT and prevents any changes to *cwnd* during this time. This is an attempt to send at the proper rate while waiting for the ACKs for these data packets to start arriving. After the RTT period is over, it transitions into the Wait state. The Wait state lasts for one RTT while the *cwnd* is allowed to be changed and the damper does nothing. After the RTT period is over, it transitions into the MonitorHigh state and starts over. In testing the selective damper, the large, slow oscillations typically disappear after one complete cycle through the states, allowing Copa to operate normally afterward.

During testing of this implementation of Copa over networks containing significant amounts of jitter, it was found that the jitter resistance provided by the *RTT_{standing}* calculations was not sufficient. When the amount of jitter causes the RTT measurements to very rarely approach *RTT_{min}*, the jitter ends up being treated as queueing delay and the send rate ends up artificially low. A configurable anti-jitter configuration parameter was added to Copa to allow operation at nearly full rate over these networks. This parameter works as follows. After *RTT_{min}* is updated using the RTT measurement, the anti-jitter parameter, which is specified in seconds, is subtracted from the RTT measurement without letting the result go below the *RTT_{min}* value. This subtracts enough jitter to allow the RTT measurements to approach *RTT_{min}* on a regular basis, which is required for Copa to operate correctly. This adjusted RTT measurement is then used in all of the other Copa calculations. This anti-jitter parameter is not automatically tuned by Copa. Instead, manual tuning of it is required when operating over networks of this type.

In order to avoid using a true sliding window for the *RTT_{standing}* and *RTT_{min}* calculations, which could require a lot of computational resources (both memory and processing), a sliding window based on a fixed number of "bins" is used in the implementation instead. This trades a small amount of window timing inaccuracy for a smaller, constant memory footprint and a lower, constant processing time.

The following constants are used by Copa:

- FS_PAIRS - The number of fast startup packet pairs. Set to 11.
- DT_BINS - The number of bins in the delay tracker. Set to 32.
- SRTT_ALPHA - The smoothed RTT alpha parameter. Set to 1/16.
- SS_THRES - The slow-start RTT threshold. Set to 0.05 seconds.
- QUIET_PERIOD - The maximum amount of time that a paced send can be late without the sender considered to be quiescent. Set to 0.01 seconds.
- HUGE_RTT - The RTT value used when the RTT is not to be used. Set to 7200.0 seconds.
- PP_RTT_ADJ - The packet pair RTT adjustment amount. Set to 0.025 seconds.

- MIN_CWND - The minimum congestion window size in packets. Set to 2.
- INC_CWND_THRES - The maximum congestion window size for always allowing a congestion size window increase. Set to 16.
- INC_CWND_RATIO - The maximum portion of the congestion window that can be unused in order to allow congestion window size increases. Set to 0.5.
- DAMPER_THRES - The number of packets in the bottleneck queue that initiates the selective damper. Set to 40.
- PKT_OVERHEAD - The nominal packet overhead size, which is set to 54 bytes (26 bytes for Ethernet, 20 bytes for IP, and 8 bytes for UDP).
- NPS - The nominal packet size, which is set to 1000 bytes. Used to adapt the computations from units of packets to bytes, in order to support variable packet sizes.
- DATA_HDR_BASE_SIZE - The base size of the SLIQ data header in bytes. Set to 16 bytes.
- FAST_RTO_CWND_THRES - The fast RTO congestion window threshold value in packets. Set to 32.0.

The following state variables are maintained by Copa:

- *state* - The Copa operating state. Either NOT_CONNECTED, FAST_STARTUP, SLOW_START, or CLOSED_LOOP. Initialized to NOT_CONNECTED.
- *report_min_rtt* - The flag controlling if the local minimum RTT should be reported to the other endpoint. Initialized to false.
- *fast_startup* - The fast startup state information, including the number of packet pairs sent, an array of packet pair send times, an array of packet pair receive times, an array of RTT estimates, an array of channel rate estimates, and a timer. All state is initialized to zero.
- *standing_rtt_state* - The standing RTT sliding window state information, including the recent minimum time value, the recent observation time, a circular array of time value/observation time bins, and the previous update time. All state is initialized to zero.
- *min_rtt_state* - The minimum RTT sliding window state information, including the recent minimum time value, the recent observation time, a circular array of time value/observation time bins, and the previous update time. All state is initialized to zero.
- *min_ts_delta_state* - The minimum timestamp delta sliding window state information, including the recent minimum time value, the recent observation time, a circular array of time value/observation time bins, and the previous update time. All state is initialized to zero.
- *velocity_state* - The velocity state information, including the previous direction, the same direction count, the starting congestion control sequence number, the starting congestion window size, and the flag recording if the congestion window size was increasing at the start of the current update window. All state is initialized to zero and false.
- *damper* - The selective damper information, including the current damper state (MONITOR_HIGH, MONITOR_LOW, HOLD, or WAIT), and a hold count. Initialized to MONITOR_HIGH and zero.
- *anti_jitter* - The configured anti-jitter parameter in seconds. Initialized from the SLIQ configuration information, and defaults to 0 if not specified.
- *delta* - The algorithmic parameter for aggressiveness (δ). Initialized to 0.5.

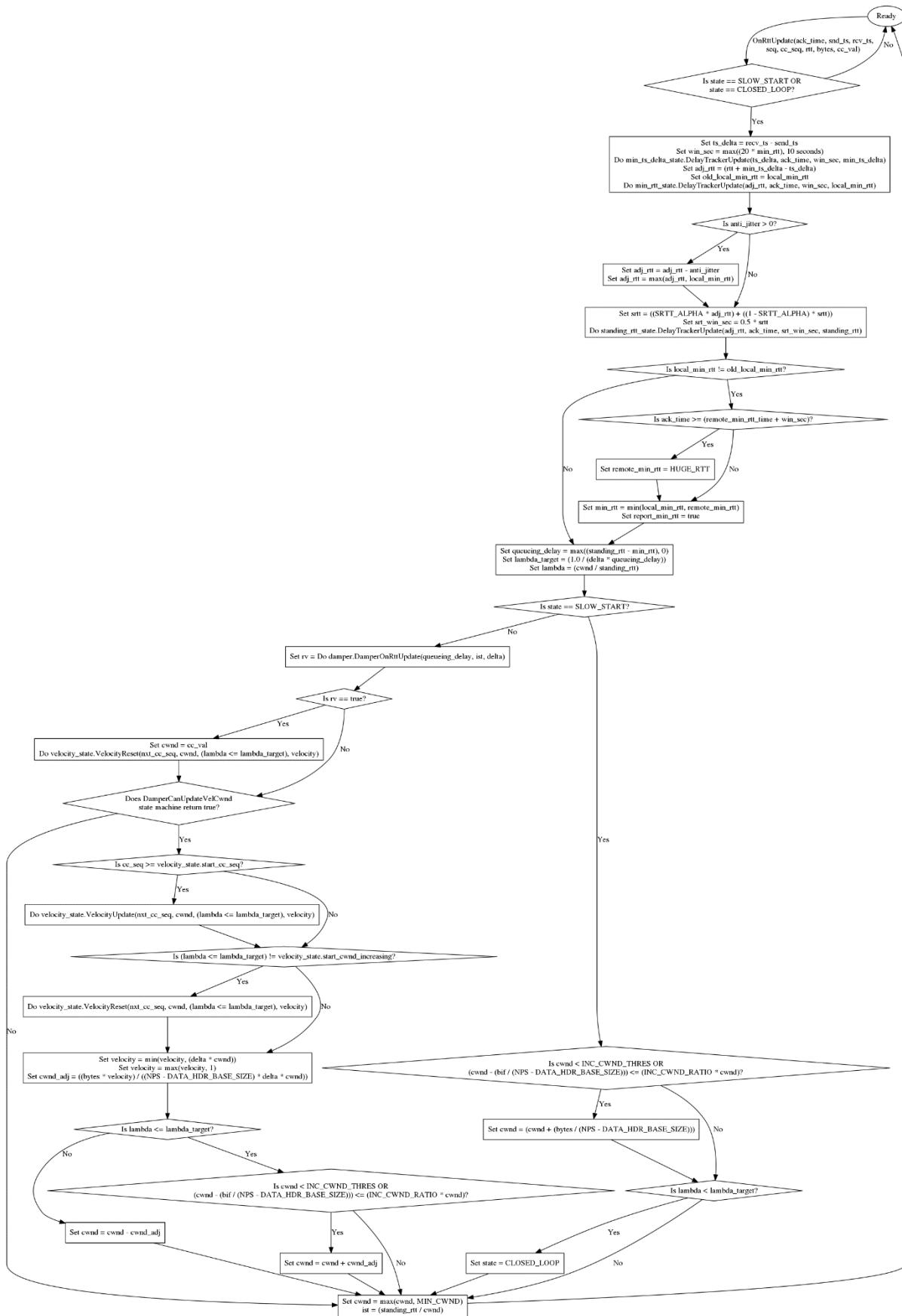
- *srtt* - The smoothed RTT in seconds. Initialized to `HUGE_RTT`.
- *standing_rtt* - The standing RTT estimate in seconds. Initialized to `HUGE_RTT`.
- *min_rtt* - The minimum RTT estimate in seconds. Initialized to `HUGE_RTT`.
- *local_min_rtt* - The local minimum RTT estimate in seconds. Initialized to `HUGE_RTT`.
- *remote_min_rtt* - The remote minimum RTT estimate in seconds. Initialized to `HUGE_RTT`.
- *min_ts_delta* - The minimum timestamp delta value in seconds. Initialized to `HUGE_RTT`.
- *cwnd* - The congestion window size in packets. Initialized to 3.0.
- *ist* - The packet inter-send time in seconds ($1/\lambda$). Initialized to 1.0.
- *velocity* - The congestion window adjustment velocity parameter. Initialized to 1.
- *nxt_cc_seq* - The next congestion control sequence number to be sent. Initialized to 0.
- *remote_min_rtt_time* - The time that the remote minimum RTT arrived. Initialized to 0.
- *next_send_time* - The time that the next data packet can be sent. Initialized to the current time.
- *timer_tolerance* - The expected accuracy of the SLIQ event timer in seconds. Initialized to 0.001 seconds.

Copa implements the congestion control API as shown in the following table and state machine diagrams.

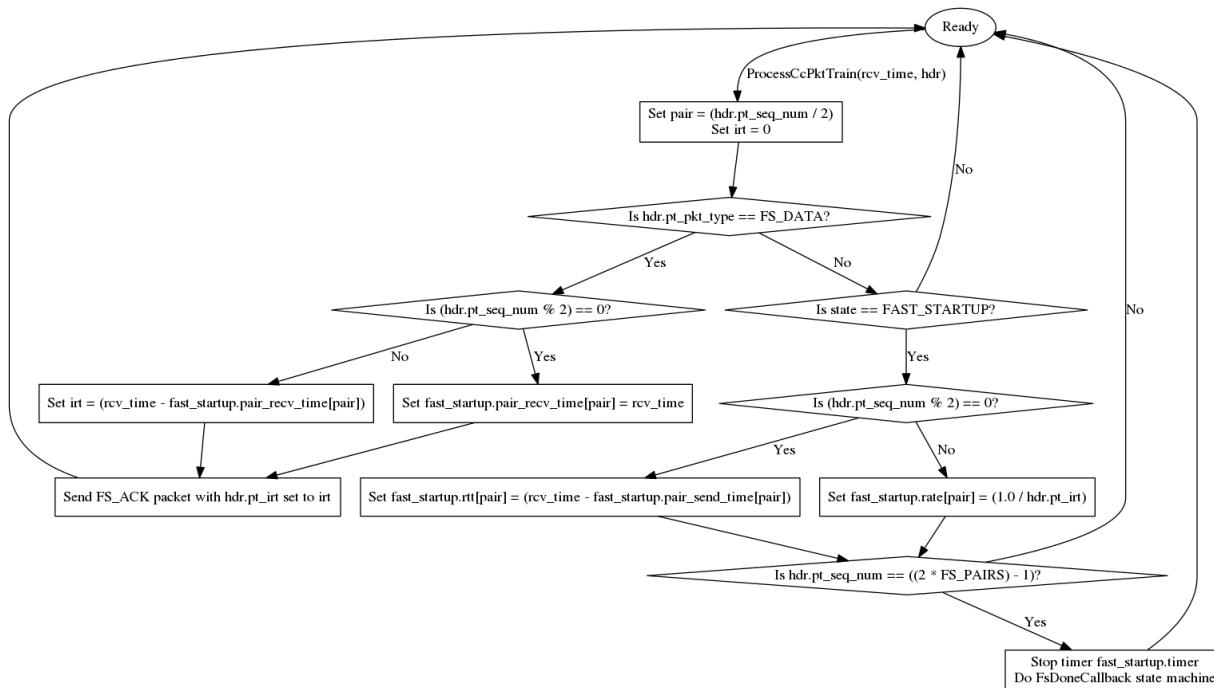
Event		Copa Processing Details
Configure		Initialize all state variables as listed above. The <i>anti_jitter</i> parameter is extracted from the SLIQ configuration information, and defaults to 0 if not specified.
Connected		If <i>state</i> is not equal to <code>NOT_CONNECTED</code> , then log an error and return immediately. Set <i>smoothed_rtt</i> , <i>standing_rtt</i> , <i>min_rtt</i> , and <i>local_min_rtt</i> all to the SLIQ connection handshake RTT. Clear all of the <i>fast_startup</i> state. If the SLIQ connection handshake RTT is greater than <code>SS_THRES</code> , then set <i>state</i> to <code>FAST_STARTUP</code> and initiate the sending of the <code>FS_PAIRS</code> data packet pairs ($2 * \text{FS_PAIRS}$ Congestion Control Packet Train packets) with the packet train packet type set to 0 and containing packet train sequence numbers from 0 to $((2 * \text{FS_PAIRS}) - 1)$ for fast startup. This requires sending the first packet pair immediately and starting a timer to send the other packet pairs every $(\text{smoothed_rtt} + \text{PP_RTT_ADJ}) / \text{FS_PAIRS}$ seconds. Once the last packet pair is sent, start a timer to call <code>FsDoneCallback</code> after $(2 * \text{FS_PAIRS} * (\text{smoothed_rtt} + \text{PP_RTT_ADJ}))$ seconds, or a maximum of 1 second. If the SLIQ connection handshake RTT is less than or equal to <code>SS_THRES</code> , then set <i>state</i> to <code>SLIQ_START</code> and <i>ist</i> to the SLIQ connection handshake RTT divided by <i>cwnd</i> to initiate slow-start.
UseRexmitPacing		Returns true.
UseCongWinForCapEst		Returns true.
UseUnaPktReporting		Returns false.
SetTcpFriendliness		Ignored.
ActivateStream		Ignored.
DeactivateStream		Ignored.
OnAckPktProcessingStart		Ignored.
OnRttUpdate		See state machine below.
OnPacketLost		Returns true.
OnPacketAcked		Ignored.
OnAckPktProcessingDone		Ignored.

Event	Copa Processing Details
OnPacketSent	Assign the next congestion control sequence number <i>nxt_cc_seq</i> to the data packet and increment <i>nxt_cc_seq</i> by one. Record the current <i>cwnd</i> value in the congestion control specific state for the data packet. Do the DamperOnPktSent state machine. Do the UpdateNextSendTime state machine passing the size of the data packet in bytes.
OnPacketResent	Update the congestion control specific state for the data packet with the current <i>cwnd</i> value. Do the DamperOnPktSent state machine. If the resend is not due to an RTO event, then do the UpdateNextSendTime state machine passing the size of the data packet in bytes.
ReportUnaPkt	Ignored.
RequireFastRto	Since Copa implements a congestion window and can operate with non-congestion packet losses, fast RTOs are required when the congestion window size is small. If <i>cwnd</i> is less than FAST_RTO_CWND_THRES, then return true, otherwise return false.
OnRto	Ignored.
OnOutageEnd	Ignored.
CanSend	If state is equal to SLOW_START or OPEN_LOOP and (<i>cwnd</i> * (NPS - DATA_HDR_BASE_SIZE)) is greater than <i>bif</i> , then return true, otherwise return false.
CanResend	Returns true.
TimeUntilSend	If the current time <i>now</i> plus <i>timer_tolerance</i> is greater than or equal to <i>next_send_time</i> , then return 0 seconds, otherwise return (<i>next_send_time</i> - <i>now</i>) seconds.
SendPacingRate	Returns ((8 * (NPS + PKT_OVERHEAD)) / <i>ist</i>) bits per second.
SendRate	Returns ((8 * (NPS + PKT_OVERHEAD)) / <i>ist</i>) bits per second.
GetSyncParams	This call is used to send the local minimum RTT value to the other endpoint in order to synchronize the two minimum RTT values. If <i>report_min_rtt</i> is true, then set the returned parameter value to <i>local_min_rtt</i> , set <i>report_min_rtt</i> to false, and return true. Otherwise, return false.
ProcessSyncParams	This call is used to process a received minimum RTT value from the other endpoint in order to synchronize the two minimum RTT values. Set <i>remote_min_rtt</i> to the received parameter value. Set <i>remote_min_rtt_time</i> to the current time. Set <i>min_rtt</i> to the minimum of <i>local_min_rtt</i> and <i>remote_min_rtt</i> .
ProcessCcPktTrain	See state machine below.
InSlowStart	If state is not equal to CLOSED_LOOP, then return true, otherwise return false.
InRecovery	Returns false.
GetCongestionWindow	Returns (<i>cwnd</i> * (NPS - DATA_HDR_BASE_SIZE)) bytes.
GetSlowStartThreshold	Returns 0.
Close	Ignored.

The OnRttUpdate state machine is shown below. It updates all of the standing RTT, minimum RTT, minimum timestamp delta, and velocity state information. The results are used to update the congestion window and the packet inter-send time to use for pacing packet transmissions.

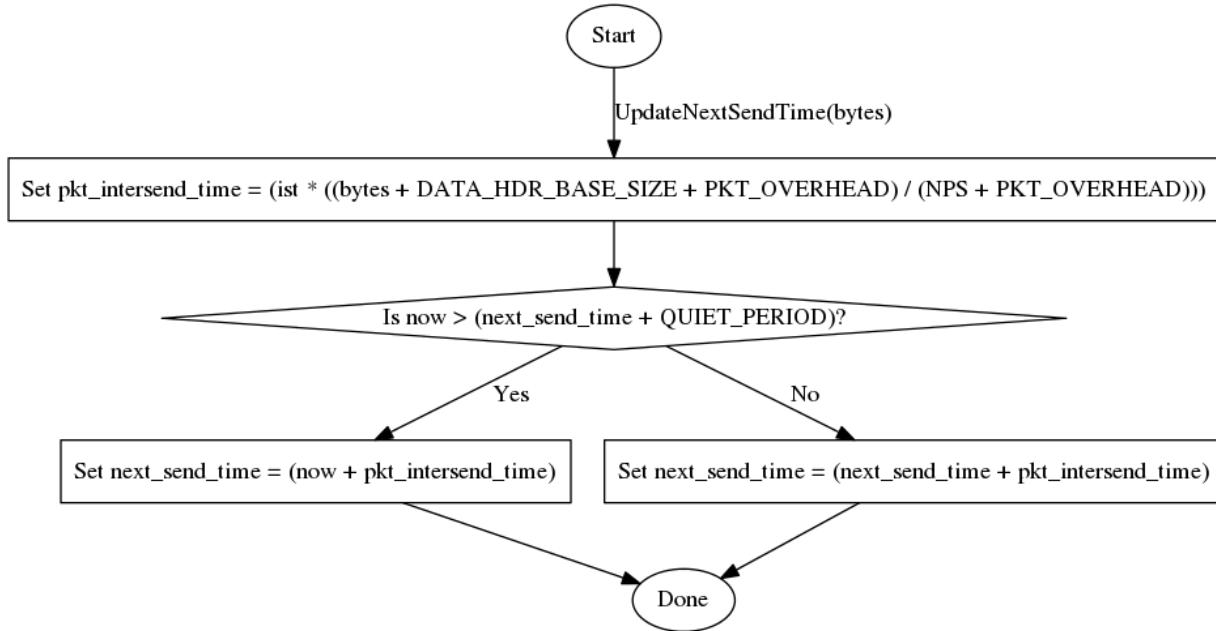


The ProcessCcPktTrain state machine is shown below. This logic handles the received SLIQ Congestion Control Packet Train data packets (packet train packet type 0) during fast startup and generates the necessary SLIQ Congestion Control Packet Train ACK packets (packet train packet type 1). Data packet pairs with even packet train sequence numbers cause the receive time to be recorded and a packet pair ACK packet to be sent back, while those with odd sequence numbers cause the inter-receive time to be computed and sent back in a packet pair ACK packet. Packet pair ACK packets with even sequence numbers cause the RTT measurement to be recorded, while those with odd sequence numbers cause the rate estimate to be recorded. If the last packet pair ACK packet (with packet train sequence number 21) is received, then the timer is stopped and FsDoneCallback is called immediately.



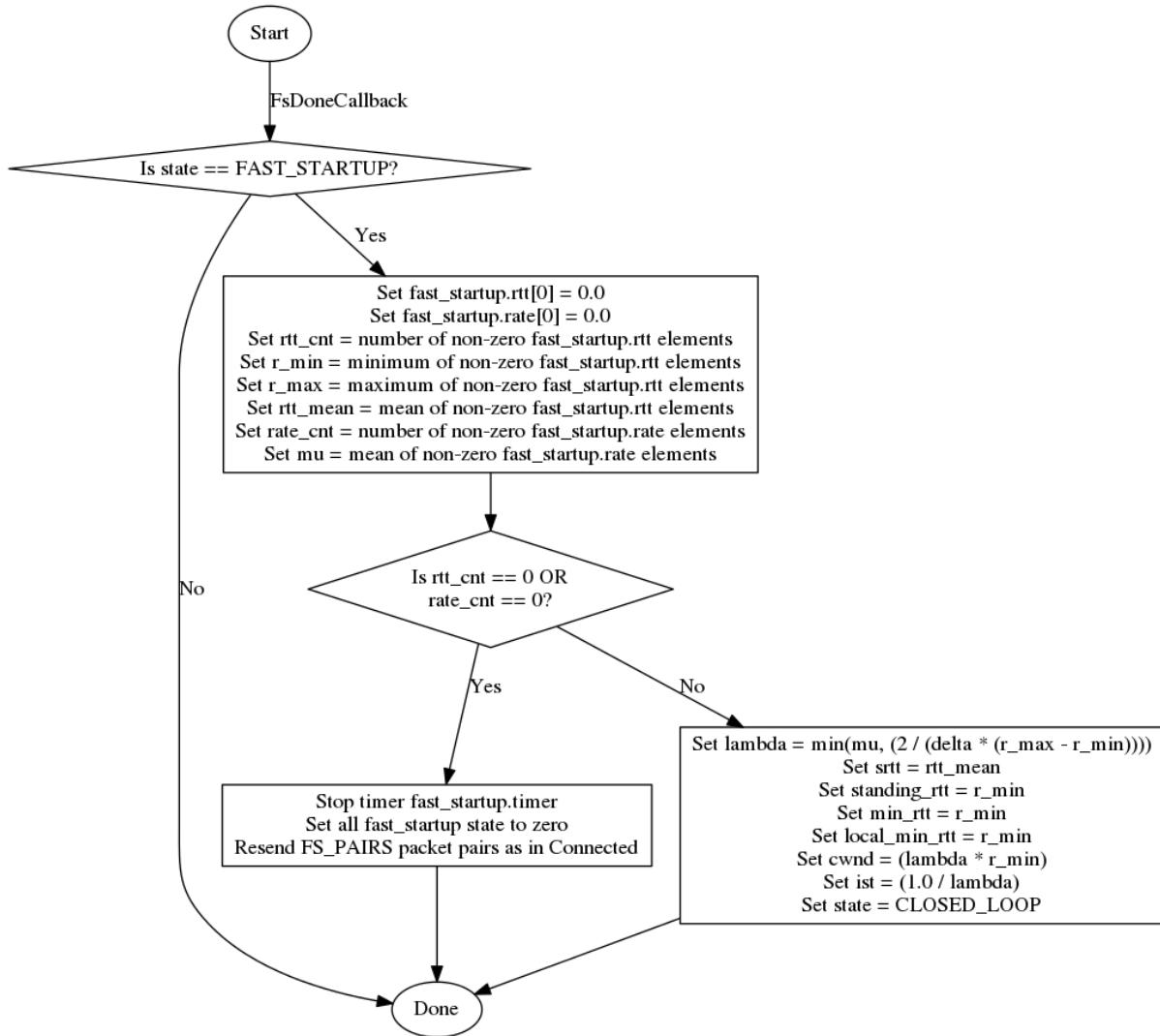
The UpdateNextSendTime state machine is shown below. This state machine is called by other state machines above, and updates the next send time based on the number of bytes just sent. If

the application does not attempt to send the next data packet within the QUIET_PERIOD, then the next_send_time must be jumped forward in order to prevent incorrect send pacing behavior.

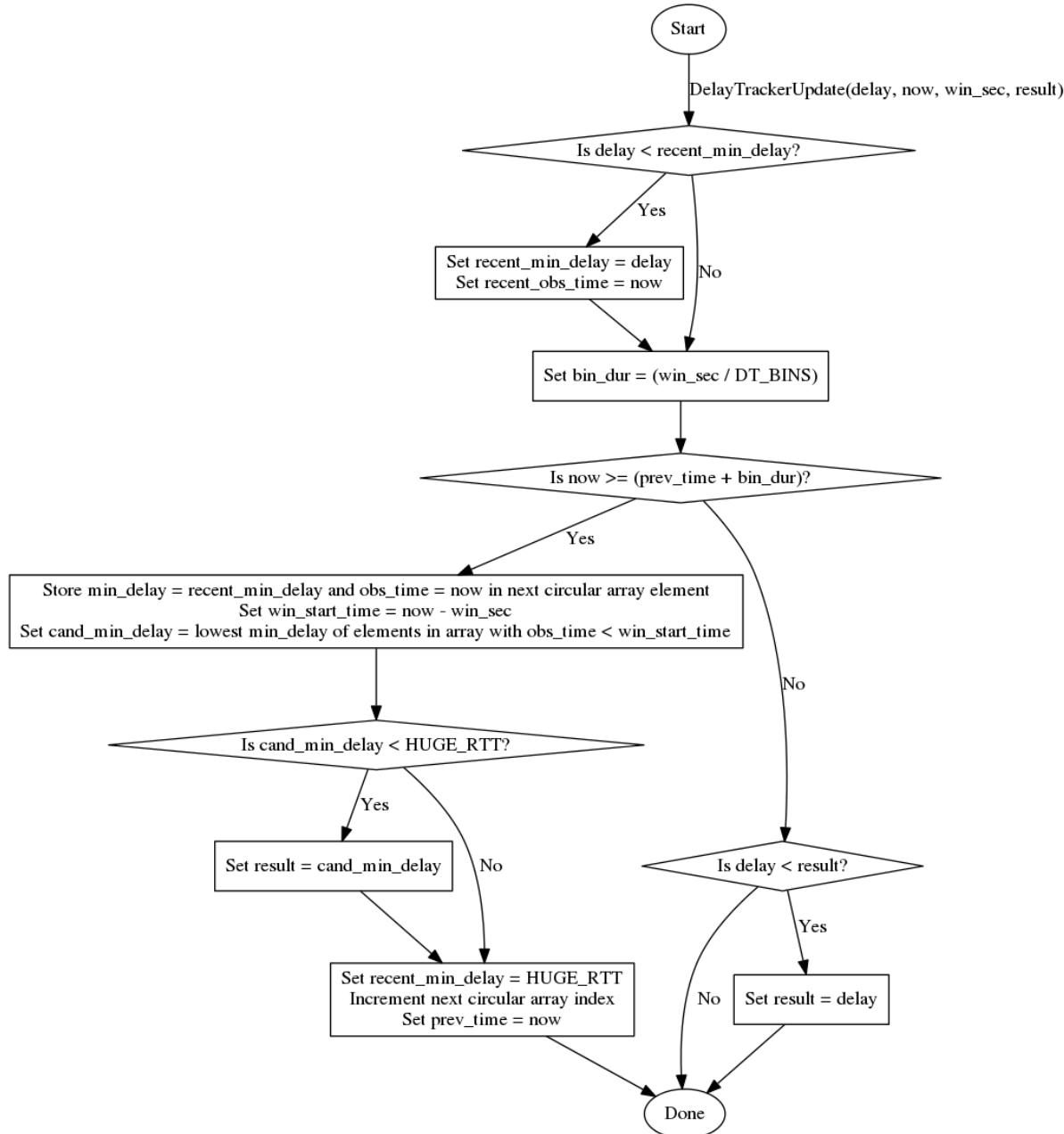


The FsDoneCallback state machine is shown below. It is called by the other state machines above. The logic erases the first RTT and rate estimate data, then processes the fast startup RTT and rate estimates into starting values for the *srtt*, *standing_rtt*, *min_rtt*, *local_min_rtt*, *cwnd*, and

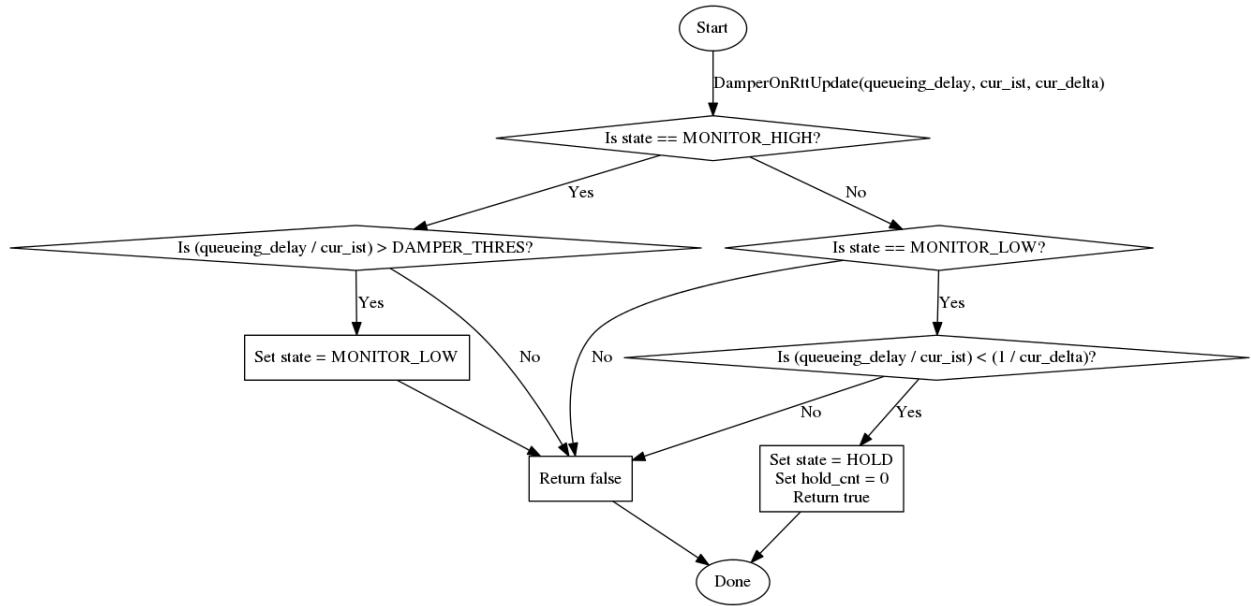
ist state. If there were not enough fast startup estimates gathered, then the fast startup procedure is restarted.



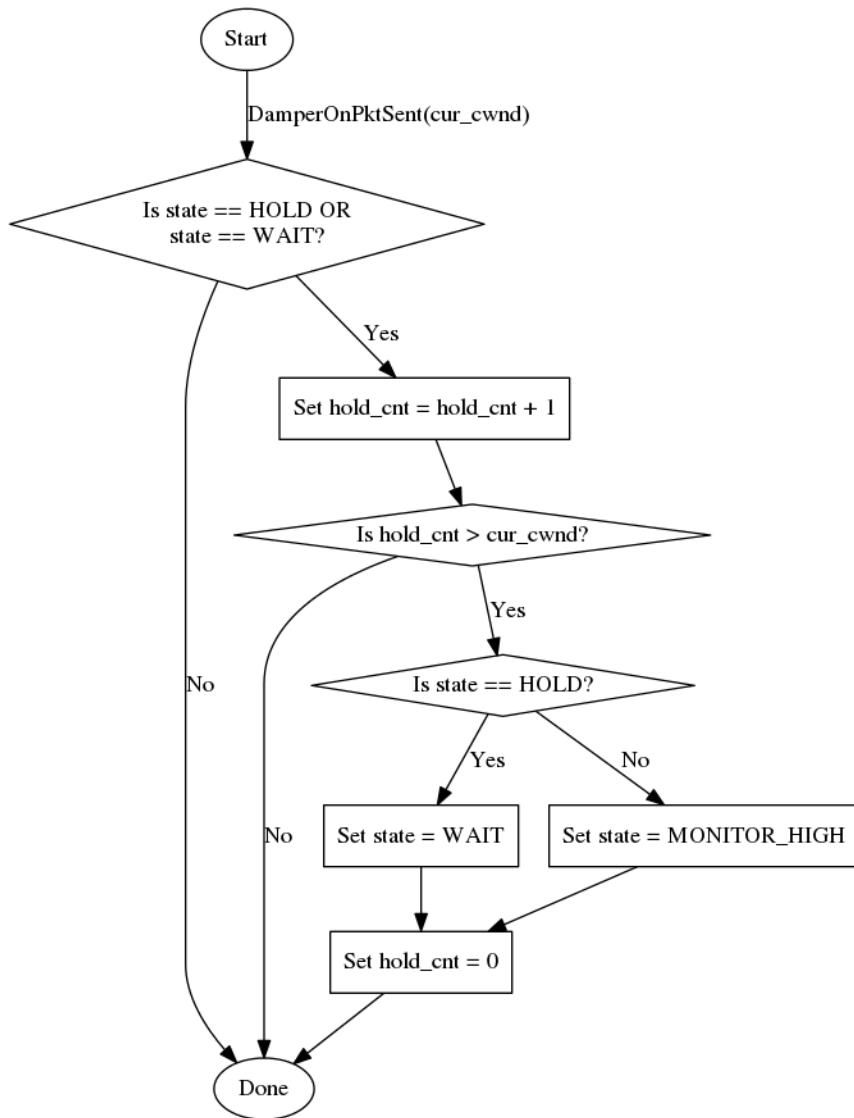
The DelayTrackerUpdate state machine is shown below. It is called by the OnRttUpdate state machine above. This logic updates the delay tracker state with a new delay value in order to find the minimum delay over a time window using DT_BINS individual bins. The delay argument is the new delay value, the now argument is the current time, the win_sec argument is the current time window in seconds, and the result argument is the current minimum delay value which is updated if needed.



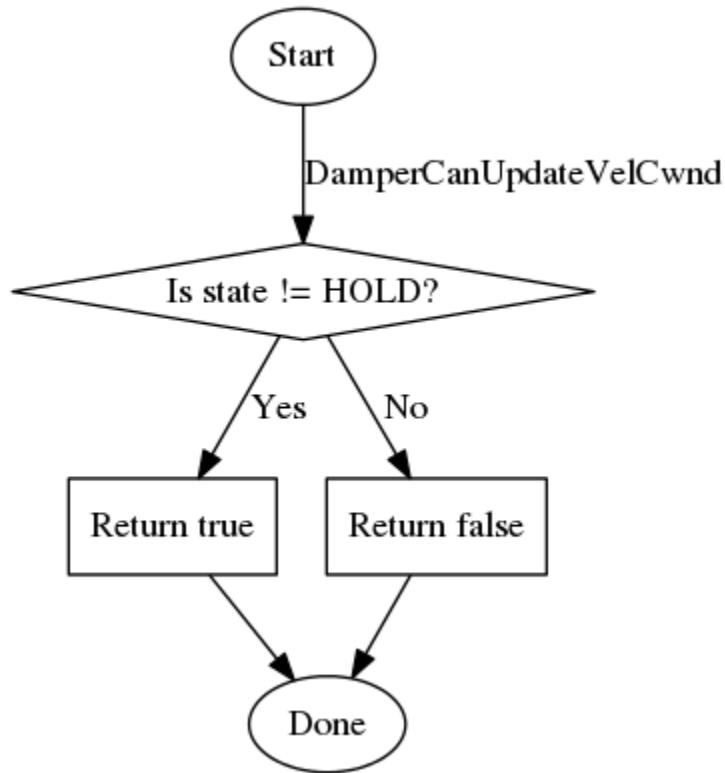
The DamperOnRttUpdate state machine is shown below. It is called by the OnRttUpdate state machine above. This logic updates the selective damper state when a new RTT measurement is processed. The queueing_delay argument is the new queueing delay estimate, the cur_ist argument is the current inter-send time, and the cur_delta argument is the current delta value. The logic returns a Boolean value, which controls if the congestion window size must be set to the value in use when the data packet that is being ACKed was sent. This is done only if true is returned.



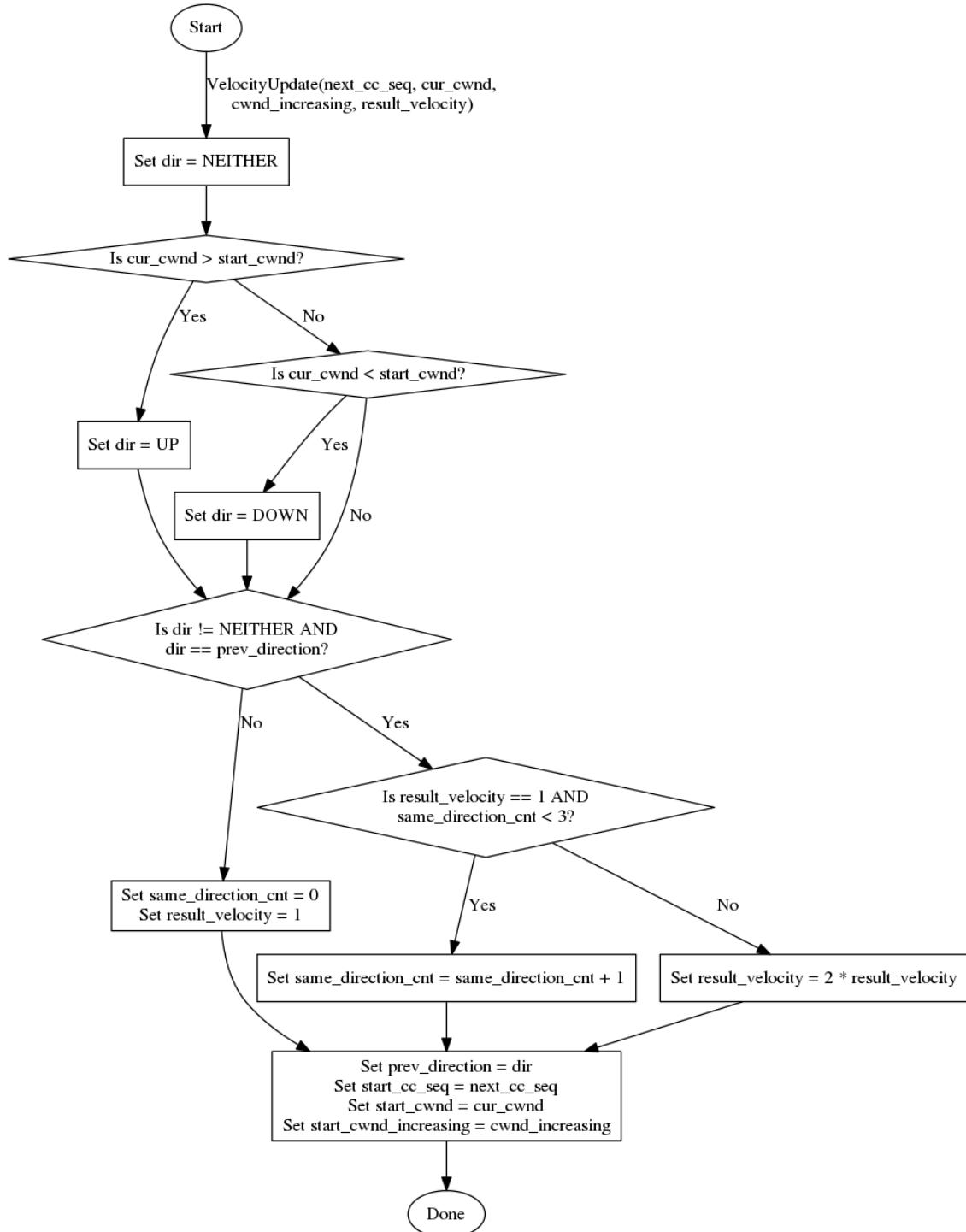
The DamperOnPacketSent state machine is shown below. It is called by the OnPacketSent and OnPacketResent state machines above. This logic updates the selective damper when a data packet is either sent or resent. The cur_cwnd argument is the current congestion window size.



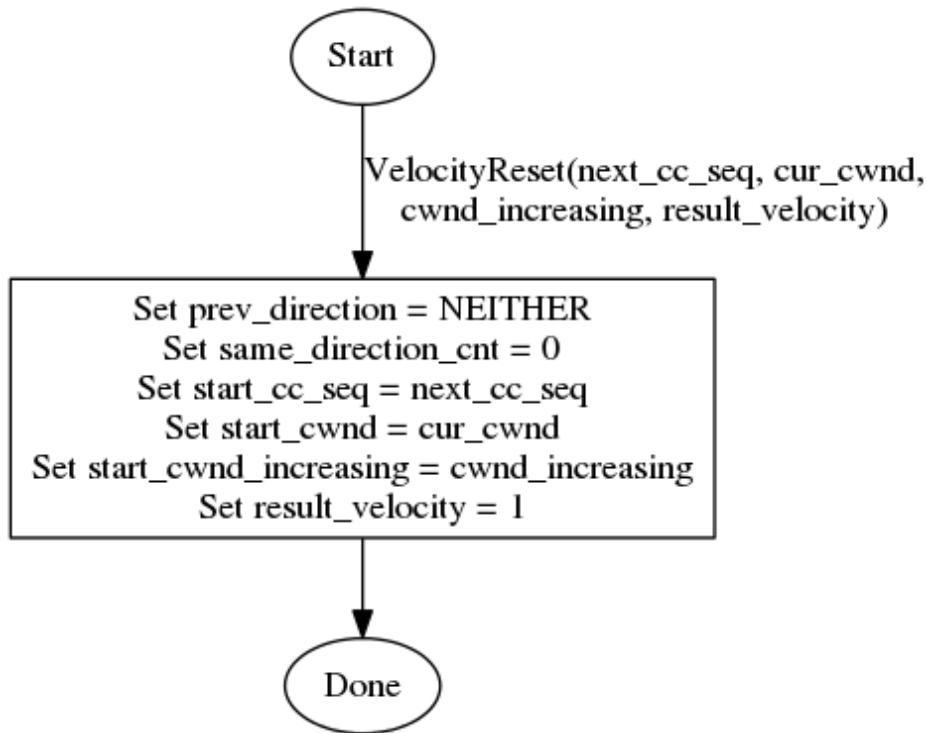
The DamperCanUpdateVelCwnd state machine is shown below. It is called by the OnRttUpdate state machine above. This logic checks if the selective damper allows the velocity and congestion window to be updated at the current time. It returns true if they can be updated, or false if not.



The VelocityUpdate state machine is shown below. It is called by the OnRttUpdate state machine above. This logic updates the velocity parameter based on the current state when needed. The next_cc_seq argument is the next data packet congestion control sequence number that will be assigned, the cur_cwnd argument is the current congestion window size, the cwnd_increasing argument is a flag indicating if the congestion window size is being increased or decreased, and the result_velocity argument is the current velocity value which is updated if needed.



The VelocityReset state machine is shown below. It is called by the OnRttUpdate state machine above. This logic resets the velocity parameter and state information when needed. The next_cc_seq argument is the next data packet congestion control sequence number that will be assigned, the cur_cwnd argument is the current congestion window size, the cwnd_increasing argument is a flag indicating if the congestion window size is being increased or decreased, and the result_velocity argument is the current velocity value which is updated.



7.3.3 SLIQ CAT

The SLIQ CAT implements the Path Controller API needed by the backpressure forwarder, and uses the SLIQ protocol to establish a tunnel over the black network. In general, the backpressure forwarder will instantiate multiple path controllers. In order for the backpressure forwarder to keep track of the different CATs, each CAT is assigned a unique integer number. Each SLIQ CAT stores its assigned integer identifier and uses it when interacting with the backpressure forwarder.

A pair of SLIQ CATs, one located at each of two different ION nodes, are used to create a single tunnel between the ION nodes over the black network. One of these SLIQ CATs must be configured as the SLIQ CAT client, and the other must be configured as the SLIQ CAT server. At startup, each SLIQ CAT determines if it should be the client or the server based on the endpoint IP addresses, and the SLIQ CAT client performs a direct connect operation to the SLIQ CAT server to create a single tunnel between the two ION nodes. The SLIQ CAT client will keep attempting to connect to the SLIQ CAT server until the connection is established. If the connection breaks during operation, the SLIQ CAT client will keep attempting to reconnect to the SLIQ CAT server.

The SLIQ CAT uses five bi-directional streams within the single connection that it maintains.

- **Stream 1:** Used for exchanging QLAM packets. This stream is set to priority 2 in SLIQ, which is the highest priority of the five streams. It is configured to use the best-effort reliability mode (no retransmissions will occur). Its transmit queue is set to operate as a FIFO, with the maximum queue size set to 1 packet, and with a head drop policy (discarding any old QLAM packet for the new QLAM packet).
- **Stream 3:** Used for exchanging Expedited Forwarding (EF) data packets, which are also call latency-sensitive data packets. This stream is set to priority 3 in SLIQ. By default, it is configured to use the semi-reliable ARQ reliability mode, although the user can configure it to use a different reliability mode. It is configured with a retransmission limit set to 5. Its transmit queue is set to operate as a FIFO, with the maximum queue size set automatically based on the backpressure forwarder's transmit queue threshold, and with no drop policy (new EF data packets cannot be sent when the transmit queue is full).
- **Stream 5:** Used for exchanging system-level control packets (currently LSA and CAT Capacity Estimate packets). This stream is set to priority 4 in SLIQ. It is configured to use the semi-reliable ARQ reliability mode, with a retransmission limit set to 5. Its transmit queue is set to operate as a FIFO, with the maximum queue size set to 100 packets, and with no drop policy (new data packets cannot be sent when the transmit queue is full).
- **Stream 7:** Used for exchanging normal (i.e., non-EF) data packets and flow-level control packets (currently GRAM and RRM packets). This stream is set to priority 5 in SLIQ. It is configured to use the semi-reliable ARQ reliability mode, with a retransmission limit set to 5. Its transmit queue is set to operate as a FIFO, with the maximum queue size set automatically based on the backpressure forwarder's transmit queue threshold, and with no drop policy (new data packets cannot be sent when the transmit queue is full).
- **Stream 9:** Optionally used for periodically sending dummy data to estimate the channel capacity. This stream is set to priority 7, which is the lowest priority of the five streams. It is configured to use the best-effort reliability mode (no retransmissions will occur). Its transmit queue is set to operate as a FIFO, with the maximum queue size set to 250 packets, and with no drop policy (new data packets cannot be sent when the transmit queue is full).

None of the stream types guarantee packet ordering at the receiver.

When the backpressure forwarder checks for the number of bytes queued in the SLIQ CAT waiting to be sent, the SLIQ CAT adds up all of the bytes in the transmit queues for just the first four streams. Thus, the returned number includes all QLAM, EF data, LSA, CAT Capacity Estimate, non-EF data, GRAM, and RRM packets that have not been sent yet. Any queued capacity estimate dummy packets are ignored when computing the number of bytes queued for backpressure forwarding. Packets for streams using semi-reliable or reliable delivery that have been already been transmitted are held in a retransmit queue until they are acknowledged by the receiver; these packets are not included as part of the number of bytes queued.

The backpressure forwarder uses the capacity estimates received from the SLIQ CATs to control the QLAM send rate to each SLIQ CAT. The send rate to each CAT is specified as a percentage

of the available channel within the backpressure forwarder configuration. As each SLIQ CAT reports an updated capacity estimate, the backpressure forwarder recomputes the QLAM sent rate for that SLIQ CAT using the size of the last QLAM packet sent.

As explained in the Backpressure Forwarder and SLIQ sections, each latency-sensitive data packet carries along a time-to-go (TTG) value that indicates the amount of time remaining before it is considered "expired." When an Expedited Forwarding (EF) packet is admitted into the network, it is assigned a configurable TTG value that is subsequently decremented at each intermediate IRON node. When the packet's TTG value reaches zero, the packet is considered to be expired. In order to adjust the EF packet TTG values as accurately as possible, the following approach is used. First, just before a packet is sent by SLIQ, the SLIQ data packet header's TTG value is computed from the packet's TTG adjusted down by the difference between the current time and the packet's receive time. Note that the packet's stored TTG and receive time values are not updated in this process in order to allow any following SLIQ data packet retransmissions to update the SLIQ data packet header's TTG value again correctly. Then, when the EF data packet is received by the neighboring SLIQ CAT, it uses the SLIQ data header timestamp and timestamp delta values in order to adjust the received SLIQ data packet header's TTG value down by the one-way delay (OWD) experienced by the packet, and the packet's receive time and updated TTG value are stored in the EF packet on the node. The OWD delay estimation is performed as follows.

- During a OWD sampling period, which is 10 seconds, the SLIQ data header send timestamp (T_s) and sender timestamp difference (D_s) values from each received packet are used along with the local clock's receive timestamp for the packet (T_r) to compute an instantaneous clock delta value:

$$\text{delta}_{\text{inst}} = (((T_r - T_s) - D_s)/2)$$

This equation computes an average of the clock difference at the two SLIQ CATs for the packet.

- During the OWD sampling period, the maximum and minimum instantaneous clock delta values, $\text{delta}_{\text{max}}$ and $\text{delta}_{\text{min}}$, are maintained.
- Once the OWD sampling period is over, the midpoint of the clock delta values is computed for use during the next OWD sampling period:

$$\text{delta}_{\text{midpt}} = ((\text{delta}_{\text{max}} + \text{delta}_{\text{min}})/2)$$

- For each EF data packet received during the next OWD sampling period that requires a TTG adjustment, the adjustment is:

$$\text{TTG}' = \text{TTG} + \text{delta}_{\text{midpt}} - (T_r - T_s)$$

Each SLIQ CAT supports the following configuration parameters that may be specified in the backpressure forwarder configuration file.

Parameter	Details
Label	An optional string label to use for the SLIQ CAT. Used to differentiate parallel CATs between two IRON nodes in the visualization tools.
Endpoints	The IPv4 addresses and optional port numbers for the local and remote endpoints of the tunnel. Must use the format "LOCAL_IP[:LOCAL_PORT]->REMOTE_IP[:REMOTE_PORT]" where the IP addresses are specified in dot decimal notation, and the optional UDP port number are specified as integers. Note that the SLIQ CAT automatically determines which end is the client and which is the server (the higher IP address will be the server). The port numbers default to 30300 if not specified. This setting is required.
EfDataRel	The optional reliability mode for Expedited Forwarding (EF) data packets. May be "ARQ" for semi-reliable ARQ mode, or "FEC(<l>,<k>)" for semi-reliable ARQ+FEC mode. For semi-reliable ARQ+FEC mode, "<p>" is the target packet delivery probability for delivering the packets within the limit "<l>". The limit "<l>" may be a floating-point time in seconds or an integer number of rounds. To determine which limit type is being specified, a time must have an "s" at the end (short for "seconds"). The probability "<p>" must be specified as a floating-point number between 0.95 and 0.999 (inclusive). The limit "<l>" must be either a time in seconds between "0.001s" and "64.0s" (inclusive), or a number of rounds between 1 and 7 (inclusive). Defaults to "ARQ".
CongCtrl	An optional string specifying the congestion control algorithm(s) to use, separated by commas. Only the client side sets the congestion control algorithms for both ends of the connection. The congestion control algorithms recognized are "Cubic" (the latest implementation), "Copa", "CopaBeta2", "CopaBeta1M", "DetCopaBeta1M", "CopaBeta1_<delta>", "DetCopaBeta1_<delta>", or "FixedRate_<bps>". Note that "<delta>" must be a floating-point number in the range 0.004 to 1.0 (inclusive). Defaults to "Cubic,Copa".
Aggr	An optional congestion control algorithm aggressiveness factor, and roughly corresponding to the equivalent transmission aggressiveness that would be observed for Aggr TCP flows in aggregate. Must be an integer greater than or equal to 1. Defaults to 1.
RttOutRej	An optional RTT outlier rejection setting. When enabled, all RTT samples are passed through a median filter to eliminate those from the maximum RTT estimate. Defaults to false (disabled).
AntiJitter	An optional Copa congestion control anti-jitter value in seconds. Only used if the congestion control algorithm includes Copa. Must be a floating-point number between 0.0 and 1.0. Defaults to 0.0 (disabled).
ActiveCapEst	An optional active capacity estimation setting. When enabled, the SLIQ CAT will fill the channel with low-priority dummy data periodically as needed to keep an accurate channel capacity estimate. Defaults to false (disabled).

7.3.3.1 SLIQ CAT Headers

The Path Controller base class currently defines five different header types for use with the SLIQ CAT, as well as any other CATs developed in the future. One of these headers, the CAT capacity estimate header, is used by itself in order to exchange information between CAT endpoints. The other four headers are used to add metadata to tunneled data packets before being transmitted over the tunnel. Like the SLIQ headers, the first byte of each CAT header has a one-byte Header Type field that identifies the type of CAT header. All CAT headers must use Header Type values between 48 and 63 to avoid collisions with other header types in use in the system.

Note that in each of the CAT headers, all multi-byte fields are transmitted in network byte order (big-endian), and all unused fields in CAT headers must be set to zero when transmitted.

Each CAT header type is described in the following subsections.

7.3.3.1.1 CAT Capacity Estimate Header

The CAT capacity estimate header is used for sending the local CAT endpoint's channel capacity estimate to the remote CAT endpoint. This header is sent by itself in a separate packet, not as a metadata header for a tunneled data packet. Each CAT endpoint reports the larger of the local and remote channel capacity estimates to the backpressure forwarder for use in determining how frequently to send the QLAM control messages.

The CAT capacity estimate header is shown below.

```
.0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+
|      Type      |      Capacity Estimate      |
+-----+-----+-----+
```

The CAT capacity estimate header fields are:

- **Type** (1 byte) - Header type, set to 0x30
- **Capacity Estimate** (3 bytes) - The local transmit channel capacity estimate in units of 1000 bits/second, rounded up to the nearest 1000 bits/second value

7.3.3.1.2 CAT Packet Destination Vector Header

The CAT packet destination vector header is used to pass metadata about tunneled data packets. It is placed before the tunneled data packet prior to being passed to the tunnel for transmission, and contains a bit vector of all destination BIDs for a multicast data packet.

The CAT packet destination vector header format is shown below.

```
.0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+
|      Type      |      Destination Bit Vector      |
+-----+-----+-----+
```

The CAT packet destination vector header fields are:

- **Type** (1 byte) - Header type, set to 0x34
- **Destination Bit Vector** (3 bytes) - The destination BIDs as a bit vector

7.3.3.1.3 CAT Packet Identifier Header

The CAT packet identifier header is used to pass metadata about tunneled data packets. It is placed before the tunneled data packet prior to being passed to the tunnel for transmission, and contains a unique packet identifier for the packet. This packet identifier is assigned when the data packet is retrieved from the packet pool by one of proxies and is used to track the data packet through the system during testing.

The CAT packet identifier header format is shown below.

```
.0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+
|      Type      | BinId |          PacketId          |
+-----+-----+-----+-----+-----+-----+
```

The CAT packet identifier header fields are:

- **Type** (1 byte) - Header type, set to 0x35
- **Source Bin Identifier** (4 bits) - The source BID for the following tunneled data packet
- **Packet Identifier** (20 bits) - The unique packet identifier assigned to the following tunneled data packet

7.3.3.1.4 CAT Packet History Header

The CAT packet history header is used to pass metadata about tunneled data packets. It is placed before the tunneled data packet prior to being passed to the tunnel for transmission, and contains an array of backpressure forwarder BIDs where the packet has previously visited. This history information is used to assist in detecting packet looping (visiting the same backpressure forwarder more than once) during testing.

The CAT packet history header format is shown below.

```
.0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+
|      Type      | Node Bin ID #0 | Node Bin ID #1 | Node Bin ID #2 |
+-----+-----+-----+-----+-----+-----+-----+
| Node Bin ID #3 | Node Bin ID #4 | Node Bin ID #5 | Node Bin ID #6 |
+-----+-----+-----+-----+-----+-----+-----+
| Node Bin ID #7 | Node Bin ID #8 | Node Bin ID #9 | Node Bin ID #10|
+-----+-----+-----+-----+-----+-----+-----+
```

The CAT packet history header fields are:

- **Type** (1 byte) - Header type, set to 0x36

The header then contains a sequence of 11 Node Bin ID fields:

- **Node Bin ID** (1 byte) - The BID of the backpressure forwarder already visited, set to 0 if not used

7.3.3.1.5 CAT Packet Latency Header

The CAT packet latency header is used to pass metadata about tunneled latency-sensitive data packets. It is placed before the tunneled data packet prior to being passed to the tunnel for trans-

mission, and contains the packet's origin timestamp and time-to-go (TTG) information. This information is used during testing for computing the actual end-to-end delay for the packet and determining if the packet has been delivered within the TTG limit.

The CAT packet latency header format is shown below.

```
.0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+
|      Type      | Unused   | V |          Origin Timestamp       |
+-----+-----+-----+
|                                Time-To-Go                         |
+-----+-----+-----+
```

The CAT packet latency header fields are:

- **Type** (1 byte) - Header type, set to 0x37
- **Flags** (1 byte) - Bit field of flags
 - **Unused** (7 bits) - Unused, set to zero
 - **V** (1 bit) - Time-to-go valid flag
- **Origin Timestamp** (2 bytes) - The origin timestamp for the following latency-sensitive data packet in milliseconds
- **Time-To-Go** (4 bytes) - The time-to-go value for the following latency-sensitive data packet in microseconds if the *V* Flag is set, set to zero if the *V* Flag is not set

7.3.4 Combining Congestion Control Algorithms

The SLIQ CATs initially operated with a single selectable congestion control algorithm for each SLIQ connection between IRON nodes. This was consistent with other existing network protocols that utilize a congestion control algorithm, such as TCP. However, it was discovered through testing that the different congestion control algorithms available to SLIQ CATs have different strengths and weaknesses in terms of coping with network events. For example, the CUBIC congestion control algorithm is good at handling variations in round trip time (RTT), but is not good at handling lossy links. On the other hand, the Copa congestion control algorithm is good at handling lossy links, but is not good at competing with other TCP flows. The congestion control algorithms also share the available network capacity with other congestion control algorithms differently, with the CUBIC congestion control algorithm typically being more aggressive than the Copa congestion control algorithm. The SLIQ CATs require the congestion control algorithms to be integrated such that it can handle any possible type of network event while maintaining its fair share of the available network capacity.

One approach to solve this shortcoming is to have a SLIQ CAT switch between numerous congestion control algorithms on the fly, while keeping all of them up to date simultaneously, ready to switch between them quickly. However, this approach requires complex logic to detect when to switch congestion control algorithms, and the switching speed may not be able to keep up with the changing network events. Further, this approach may actually provide an additional attack surface, where the network event causes constant unnecessary congestion control algorithm

switching that degrades performance. For these reasons, this approach was not selected for implementation.

Another approach to solving this shortcoming is to use multiple congestion control algorithms at the same time, in parallel, between each pair of connected IRON nodes. In normal operation, the congestion control algorithms will stabilize at an operating point that will allow IRON to utilize its share of the available network capacity. During network events (e.g., random packet loss, changes in RTT, etc.), some of the congestion control algorithms might have degraded performance. As long as one of the congestion control algorithms in use can handle the event, then the SLIQ CAT can adapt to it and continue to provide the IRON nodes with the best possible service. With this approach, it is not necessary to implement logic that switches between the congestion control algorithms, since they are all being used in parallel at all times. This approach was selected for implementation.

One possible design for the parallel congestion control algorithm approach involves each SLIQ CAT setting up multiple SLIQ connections with its peer, with each SLIQ connection using a different congestion control algorithm. When one SLIQ connection is having trouble coping with a network event, another SLIQ connection (within the same SLIQ CAT) that can handle the event can take over. While this is very easy to implement in the SLIQ CAT, a shortcoming of this design is that packets sent to one SLIQ connection stay within that SLIQ connection, even when it is having trouble moving packets. This prevents packets originally transmitted on a degraded SLIQ connection from being retransmitted on another SLIQ connection when it makes sense to do so.

An alternative design for the parallel congestion control algorithm approach is to move the multiple congestion control algorithms down into the SLIQ connection itself. The SLIQ connection-level is where the congestion control algorithm currently resides, and this design would be a somewhat natural extension of the existing implementation. Having the multiple congestion control algorithms within the SLIQ connection makes it easier for packets to be transmitted or retransmitted using any of the congestion control algorithms. This would allow a packet originally transmitted using CUBIC to be retransmitted using Copa if it makes sense to do so. However, the congestion control algorithms were not designed to allow this behavior, so the algorithms must be augmented to support it. Given the natural fit of having multiple congestion control algorithms at the SLIQ connection-level, as well as the possibility of retransmitting a packet using a different congestion control algorithm, this design was chosen for implementation.

7.3.4.1 Combined Congestion Control Design

With the selected connection-level parallel congestion control algorithm design, each pair of connected IRON nodes continues to use a single SLIQ CAT instance for each end of the tunnel, with a single SLIQ connection between SLIQ CAT pairs. The SLIQ CAT's SLIQ connection continues to have five streams for the different groupings of IRON packets to be exchanged. However, instead of the SLIQ connection requiring just a single congestion control algorithm, the SLIQ connection can optionally use multiple congestion control algorithms in parallel at the same time.

The challenge of this design is to get the multiple congestion control algorithms to work with the SLIQ Connection, Stream, and SentPktManager class instances at run time, while not requiring any significant changes to the congestion control algorithm implementations themselves. The following subsections outline how this is accomplished.

7.3.4.1.1 Connection Setup

Each SLIQ connection may be configured with multiple congestion control algorithms. The existing SLIQ APIs for specifying settings for a single congestion control algorithm are extended to accept an array of settings. Internally, each algorithm is referenced using an integer identifier (the congestion control identifier, or CCID), which is equal to the congestion control algorithm settings array index specified by the application (the SLIQ CAT). For instance, if the SLIQ CAT configuration specifies two congestion control algorithms as "Cubic,Copa", then the SLIQ connection that is created will have two congestion control instances at run time, with the CUBIC congestion control algorithm instance assigned CCID 0 and the Copa congestion control algorithm instance assigned CCID 1.

The SLIQ Connection Handshake Header, which is exchanged when setting up the SLIQ connection, is extended to support multiple congestion control algorithms. The order of the algorithms in the header is by CCID, starting with CCID 0, then CCID 1, etc. Both SLIQ endpoints will use the same CCID numbering since the same SLIQ Connection Handshake Header is sent back and forth during connection establishment.

The SLIQ Connection is updated to be able to create, initialize, and store multiple congestion control algorithms along with their associated state information. To do this, new *CcAlg* and *CcAlgs* structures are used in the SLIQ Connection class in order to store an array of congestion control algorithms, with the CCID used as the index into the array.

Congestion Control Storage Structure

```
struct CcAlg
{
    CongCtrlInterface* cc_alg;
    iron::Timer::Handle send_timer;
    iron::Time next_send_time;
    bool in_ack_proc;
    bool use_rexmit_pacing;
    bool use_una_pkt_reporting;
};

struct CcAlgs
{
    bool use_una_pkt_reporting;
    CapacityEstimator cap_est;
    double chan_cap_est_bps;
    double trans_cap_est_bps;
    double ccl_time_sec;
    double send_rate_est_bps;
    size_t num_cc_alg;
    CongCtrl cc_settings[kMaxCcAlgPerConn];
    CcAlg cc_alg[kMaxCcAlgPerConn];
};
```

Within the CcAlg structure, the members are:

- cc_alg - A pointer to the actual congestion control algorithm object.
- send_timer - A timer handle object for the congestion control algorithm's send pacing timer.
- next_send_time - A time value object that stores the next available send time for the congestion control algorithm's send pacing.
- in_ack_proc - A flag for recording when the congestion control algorithm is within a series of method calls that start with OnAckPktProcessingStart and end with OnAckPktProcessingDone. This is needed since each ACK packet may or may not call into any of the congestion control algorithms.
- use_rexmit_pacing - A flag for recording the congestion control algorithm's UseRexmit-Pacing setting.
- use_una_pkt_reporting - A flag for recording the congestion control algorithm's UseUnaPktReporting setting.

Within the CcAlgs structure, the members are:

- use_una_pkt_reporting - A flag for recording if any of the congestion control algorithms use UNA packet reporting (i.e., UseUnaPktReporting returns true). If at least one of the algorithms require UNA packet reporting, then this flag is set to true. Otherwise, it is set to false. This flag is useful in order to completely skip code blocks related to UNA packet reporting if no algorithms require it.
- cap_est - The capacity estimator object for all of the congestion control algorithms. Using a single capacity estimator that utilizes the same sampling intervals for all congestion control algorithms was found to provide the best possible capacity estimates.
- chan_cap_est_bps - The latest channel capacity estimate (including packet overhead) for all of the congestion control algorithms.
- trans_cap_est_bps - The latest transport capacity estimate (excluding SLIQ, UDP, and IP header overheads) for all of the congestion control algorithms.
- ccl_time_sec - The amount of time, in seconds, since a congestion control limit event (when the congestion control algorithm actively changed a parameter that affects the send rate) changed the capacity estimate.
- send_rate_est_bps - The current send rate estimate in bits/second for all of the congestion control algorithms.
- num_cc_alg - The number of congestion control algorithms in use. Must be between 1 and kMaxCcAlgPerConn.
- cc_settings - An array of the congestion control algorithm settings as specified by the application (the SLIQ CAT). Indexed by CCID, which ranges from 0 to (num_cc_alg - 1). This is stored in its own array in order to be able to pass the entire array around to various methods by itself.
- cc_alg - An array of the congestion control structures. Indexed by CCID, which ranges from 0 to (num_cc_alg - 1).

The SLIQ Stream and SentPktManager classes are updated to be able to access the Connection's *CcAlgs* structure.

7.3.4.1.2 Original Packet Transmission

When a packet is passed to SLIQ for transmission on a Stream, the congestion control algorithms are checked in order to determine if one will allow the packet to be sent. If one is found, then the packet is sent immediately with its CCID recorded in the data header, and its CCID is stored as the associated congestion control algorithm in the SentPktManager's packet state information for the packet. If one is not found, then the packet is added to the Stream's transmit queue as normal.

Each congestion control algorithm optionally supplies a time until the next transmission can occur in order to implement send pacing. A send timer is maintained for each congestion control algorithm in order to implement send pacing correctly. When the send timer expires for an algorithm, a packet is sent from a Stream's transmit queue as normal with its CCID recorded in the data header, and its CCID is stored as the associated congestion control algorithm in the SentPktManager's packet state information for the packet.

7.3.4.1.3 Packet Retransmission

When a Stream's SentPktManager processes a received ACK packet, it decides what packets need to be retransmitted and calls back into the Stream to add those packets to the tail of the Stream's fast retransmit candidate list. When sending of retransmissions is allowed by the Connection (either a send time expires, or another event performs a check and a retransmission is allowed), it calls into the Stream to send one retransmission. Packets transmitted using a given congestion control algorithm can be retransmitted using a different congestion control algorithm in order to make the best use of the channel given the current networking conditions. Doing so requires special handling when a packet is retransmitted using a different congestion control algorithm.

- If the packet's associated CCID matches the CCID of the algorithm allowing the retransmission, then the CanResend and OnPacketResent congestion control method calls are made with the *associated_cc* flag set to true and the algorithm processes the calls as normal.
- If the packet's associated CCID does not match the CCID of the algorithm allowing the retransmission, then the CanResend and OnPacketResent congestion control method calls are made with the *associated_cc* flag set to false. This informs the algorithm that it will not have any state for the packet. The congestion control algorithm must handle these cases appropriately given that it did not send the original packet.

In either case, the CCID of the algorithm that allowed the retransmission is placed in the packet's data header.

7.3.4.1.4 ACK of Transmitted Packet

When a Stream's SentPktManager processes a received ACK packet, it uses the packet's associated CCID, which is recorded in the packet's state, to call the correct congestion control algorithm for the required OnPacketAcked method call. The congestion control algorithm handles the call as normal.

7.3.4.1.5 Transmitted Packet Missing

When a Stream's SentPktManager processes a received ACK packet, it uses the packet's associated CCID, which is recorded in the packet's state, to call the correct congestion control algorithm for the required OnPacketLost method call. The congestion control algorithm handles the call as normal.

7.3.4.1.6 RTT Measurements

When a Stream's SentPktManager processes a received ACK packet, it uses the packet's associated CCID, which is recorded in the packet's state, to call the correct congestion control algorithm for the required OnRttUpdate method call. The congestion control algorithm handles the call as normal.

7.3.4.1.7 RTO Timers

A single RTO timer is used for all congestion control algorithms within a Connection. When the RTO timer expires, the Connection calls into all of the congestion control algorithm OnRto methods. Each congestion control algorithm handles the call as normal.

8 Admission Planner

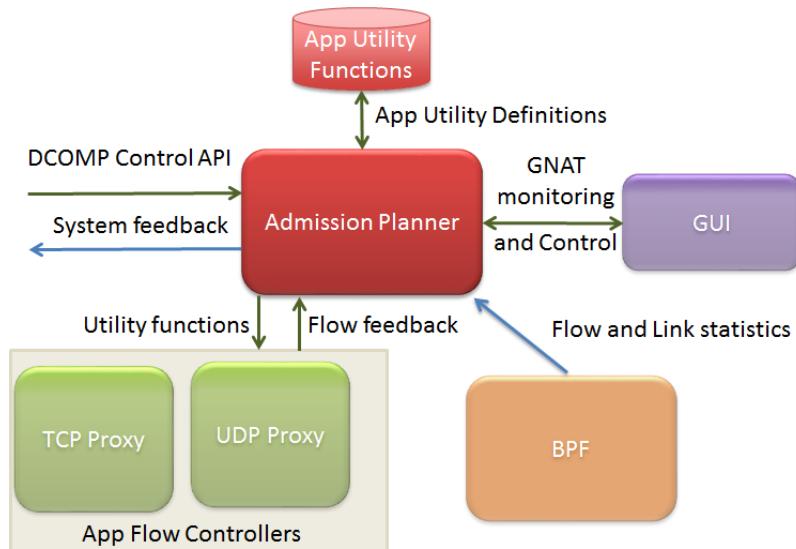
8.1 Summary of Module Goals and Objectives

The goal of the Admission Planner (AMP) is to assign and manage utility functions for individual flows and to coordinate across multiple flows. On each GNAT node, the Proxies and BPF all connect to a local AMP instance that is responsible for planning the flows for that node based on user input and network conditions. AMP also acts as a relay for statistics and queries between the GUI and the Proxies/BPF in both direction.

8.2 High Level Design

8.2.1 Adjacent Modules and Components

AMP obtains application utility function definitions from a configuration file at startup, with runtime dynamic control provided through the DCOMP Control API and the Graphical User Interface (GUI). It also acts as a relay between the proxies/BPF and the DCOMP/GUI control interfaces using a TCP connection on the remote control interface of these components. AMP acts as a TCP client and creates connections to specified Proxy/BPF and as a TCP server which accepts connections from the DCOMP controller and GUI components. The Admission Planner's output is fed into the App Flow Controllers (i.e. UDP and TCP proxies) over the remote control interface. Changes due to control actions from either the DCOMP controller or the GUI are echoed back to the GUI for monitoring purposes.



8.2.2 Use Cases

- If a new App Utility Function definition is configured through the external API, the Admission Planner (AMP) adds the information to its internal state and provides that information to the appropriate flow controller

- As the BPF reports network capacities to AMP, the AMP may modify flow utility functions or triage flows and amend the flow schedule
- As proxies report loss rates associated with flows, AMP decides which flow, if any, should be triaged in order to reduce loss rate
- As proxies report the end of a flow, AMP removes the flow from the schedule
- AMP takes request from either the GUI or the DCOMP Control API to modify App Utility Functions
- AMP relays push requests for statistics from the GUI to the proxies or BPF.
- AMP relays statistics (queue lengths, admission rate, link characteristics, utilities, etc.) from the proxies and the BPF to the GUI for visualizing network state

8.2.3 Input Sources and Information Needed

8.2.3.1 Control Plane Inputs

Source	Input	Description
App Utility Functions configuration file	Application utility data, including function shape, parameters, etc.	Utility function definitions for both classes of flows (based on service ports e.g., SCP sessions) and individual flows when the five tuple for a given flow is known in advance
GUI	Push Request	The GUI can request that statistics be periodically sent to it for visualization and specifies the period. AMP relays these messages to the specified target (either an App Flow Controller or the BPF).
DCOMP Control API or GUI	App Utility and Control commands	Both the DCOMP Control API and the GUI may modify utility function definitions via AMP.
App Flow Controller (proxies)	Flow feedback information	The App Flow Controllers update AMP with the state and conditions of the ongoing flows so as to give AMP the opportunity to adjust the schedule and utility of current and future flows. The App Flow Controller may also report the end of a flow, which the Admission Planner will remove from current and future schedules.
App Flow Controller (proxies)	Push Reply	Once statistics have been requested, they will be sent periodically to the GUI via AMP.

8.2.3.2 Events and Associated Timescales

[ConfigTime]

App Utility Functions and AMP behavioral characteristics are populated. AMP reads a config file which populates the App utility functions with service definitions and flow definitions. Service definitions map port ranges to utility functions, while flow definitions map a specific 5-tuple (protocol, source IP address, source port, destination IP address, destination port) to a utility function. Flow definitions take priority over service definitions which takes priority over a default utility function that is applied to any flow without a more specific mapping. The UDP and TCP proxies are configured with only a default utility function, with additional utility functions added to individual proxies via AMP.

[RunTime]

When AMP starts up, it attempts to establish a TCP connection to the remote control port of the configured proxies and the BPF. Once it connects to a proxy, it sends messages with the allowed service and flow definitions to the proxy. At startup, AMP also sets up a TCP server to accept connections from a GUI on a predefined port.

Either the DCOMP Control API or the GUI may modify utility functions through AMP. A five-tuple is used to identify the flow being modified and this is used to obtain the current utility function for the flow. Upon receiving such a request, the App Utility Functions is updated, and the message is relayed to the relevant proxy.

Once a GUI is connected to AMP, the GUI will send a request message to the proxies and BPF, via AMP, for periodic statistics report. These message a relayed without modification to the target. Subsequently, the proxies and BPF will send periodic statistics to the GUI via AMP. These are also relayed without processing or modification.

8.2.4 Output Destinations and Information Provided

8.2.4.1 Control Plane Outputs

Destination	Output	Description
App Flow Controllers	Utility and schedule data associated with a flow over the remote control interface.	After determining the utility of a flow and its schedule, the Admission Planner sends the 4-tuple and its utility to the App Flow Controllers, along with a command indicating whether to reject packets, start the flow, stand it by (for instance by starting a persistent timer), modify the current utility function or terminate it.
GUI	Schedule, utility and admission information messages, statistics, link characteristics	The Admission Planner may send the flow schedule for a requested period of time. This will include the flow start time and end time, priority and related application and utility functions. The GUI may also request the manifest of rejected flows during a period of time, including the identifying 4-tuple and time of rejection. [Ideally, we would want a reason why the flow was rejected, but it is not clear how we get it]

8.2.4.2 Events and Associated Timescales

[RunTime]

AMP establishes a connection to each of the proxies at startup. It then updates the respective proxy with current service definitions and flow definitions.

AMP receives a flow update message from the DCOMP Control API or the GUI, based on user input. AMP updates the utility function in its App Utility Functions internal state store and sends the updated flow definition to the relevant proxy.

AMP receives a push request from the GUI. Amp caches the message and relays the request, without modification to the target (either the BPF or one of the proxies).

AMP receives a push reply from the BPF or a proxy. AMP maps the message ID to the origin of the message and forwards it. This will happen periodically (on the order of seconds) once the GUI has sent a push request.

8.2.5 Performance Monitoring Statistics

Currently, AMP monitors flow statistics, and relays this information to the GUI for display purposes. It uses some of this information to make admission control decisions and update utility functions on the fly to maximize cumulative network utility. These statistics include:

- The number and percentage of flows that could not be admitted
- The number of flows that, while admitted, failed to be scheduled
- The total expected utility in a queried upcoming period
- The number and percentage of flows of a given type (including the total number of flows processed)
- The per-flow network feedback, including bytes delivered on time, packets delivered on time and average loss rate.

8.3 Detailed Design

When there is insufficient capacity to support all concurrent inelastic flows, competition between the flows can easily lead to a situation none of the flows receives sufficient capacity to achieve non-zero utility. To make better use of the available resources, GNAT will prevent admission of some of the flows into the network in order to free up sufficient resources so that other flows can achieve non-zero utility through a process called *triage*.

Triage responsibilities are shared between the proxies and AMP. Notionally, admission controllers within the proxies have purview only over individual flows and act independently for each flow over very short time scales (on the order of packet inter-send times). AMP has purview over all of the flows and will consider triage actions for the set of flows as a whole, acting at longer timescales.

In a typical over-subscribed scenario, the local backpressure queues will start building up, which has the effect of reducing the send rates calculated by each flow's admission controller. When the send rates drop below the minimum rate to achieve non-zero utility for a specified amount of time, the admission controllers within the proxy (using trapezoidal (TRAP) utility functions) will begin triaging flows. Reducing the number of flows reduces the arrival rates at the local backpressure queue, and allows the queues to drain -- which in turn may cause previously triaged flows to restart, causing the queues to build once more. The process will repeat and is suboptimal since capacity can be consumed that does not increase utility. Here, we use supervisory control in AMP to prevent this from happening.

There are currently three mechanisms that can cause a flow to be triaged, and as described above operate with different scopes and timescales.

- **Triage at the source by the UDP proxy, based on queue lengths at the BPF.**

- For flows with TRAP utility functions, if the queues are too large ($Q/k > p/m$) then the cost of sending a bit is higher than the utility gained by admitting that bit and the admission rate for the flow is reduced.
 - Over an interval, we calculate the maximum number of packets that could be admitted, given the instantaneous admission rates. If this value is less than the specified threshold rate (determined by the nominal send rate and the loss threshold delta) the proxy will turn the flow off for a period controlled by the *restart_interval* variable in the utility function definition.
 - We use the admission rate in these calculation rather than the actual instantaneous flow rate to handle variable bitrate traffic where the source can send below the nominal rate for periods of time.
 - After waiting the length of time specified by the *restart_interval*, the admission control will attempt to restart the flow and will be triaged again if it still cannot be supported – such thrashing wastes network resources
 - AMP's supervisory control function attempts to avoid such thrashing by modifying the admission control parameters for that flow – e.g., by setting *restart_interval* to something very long.
 - The use of a trapezoidal utility function with multiple steps biases the triage decision towards turning off new flows instead of existing flows as new flows will need to ramp up quickly to reach the minimum admission threshold (delta).
 - This uses purely local information and happens in the order of the averaging interval (configurable, fraction of a second)
- **Triage at the source by AMP, based on capacity estimates and queue size calculations.**
 - AMP knows about all concurrent flows at the source proxy, and their corresponding utility functions.
 - AMP knows about the total egress bandwidth at the BPF, based on CAT capacity estimates.
 - Assuming that all elastic traffic will be admitted, and that their rates would adapt based on queue sizes, AMP considers the impact of admitting each inelastic flow. We choose not to admit an inelastic flow if it would result in thrashing.
 - We iterate over all current inelastic flows, in decreasing order of utility per bit (based on the nominal rate of the flow).
 - At each step, the bandwidth available for the elastic traffic is calculated by subtracting the sum of the nominal rates of the admitted inelastic flows from the total reported egress capacity.
 - Given this bandwidth, we can calculate the resulting queues so that the elastic traffic will converge to the above calculated rate.
 - If this queue would result in the inelastic flow (currently being evaluated) stepping down ($Q > Kp/m$), then we choose to not admit this flow.
 - Finally, we notify the proxy of changes in the ON/OFF states of current inelastic flows.
 - There is no restart timer associated with this triage. Flows can only be turned back on if there is a change in capacity or on-going flows.

- This mechanism does not consider cross-traffic that would reduce the available egress capacity. Instead it uses the capacity as an upper-bound, therefore, there can still be thrashing.
- This uses purely local information (from the local proxies and BPF) and happens in the order of a supervisory control cycle (configurable, currently two seconds).
- Elastic flows can be triaged if they would yield less utility than inelastic flows that would now fit.
- The algorithm for finding the best set of flows is given below.

```
ComputeFit(flows, total_egress_capacities)
// Flows are sorted in decreasing order:
// p/m for inelastic flows
// p for elastic flows
// Notation: m(flow) = nominal rate of 'flow'
remaining_capacity = total_egress_capacity
elastic_flows    = GetSortedListOfElasticFlows()
For flow in flows:
if flow is inelastic:
if m(flow) < remaining_capacity:
if min_inelastic_queue_threshold > current_required_queue:
Admit(flow)
remaining_capacity = remaining_capacity - m(flow)
Update(min_inelastic_queue_threshold)
else:
tentative_elastic_flows = elastic_flows
current_utility = Utility(admitted inelastic traffic) + Utility(elastic_flows)
while (min_inelastic_queue_threshold < current_required_queue) && (!tentative_elastic_flows.empty()):
remove lowest priority flow from tentative_elastic_flows
if (Utility(admitted inelastic traffic) + Utility(flow) + Utility(tentative_elastic_flows) > current_utility):
Admit(flow)
elastic_flows = tentative_elastic_flows
// else there is more utility is we do not admit this flow
// else there is not enough capacity for this flow
// else it is an elastic flow. All elastic flows are admitted, unless triaged to accommodate and inelastic flow.
UpdateFlowStates()
// The inelastic flows that are admitted should be ON, and others should be OFF.
// The elastic flows in elastic_flows should be ON and all others should be OFF.
```

- **Triage at the source by AMP, based on feedback from the destination proxy.**
 - The destination proxy monitors loss rate and notifies the source proxy if the loss rate for a flow exceeds a configured threshold (delta from the TRAP utility function).
 - The source UDP proxy includes a sequence number and the total bytes sent thus far in each original packet.
 - The destination UDP proxy keeps track of how many original packets were reconstructed and how many bytes were released to the application.
 - The destination proxy keeps an Exponentially Weighted Moving Average (EWMA) of the loss rate, in terms of bytes, and triggers a Release Record Message (RRM) if the current average loss rate exceeds a threshold.

- These RRMs are triggered immediately when the threshold is crossed and periodically from then on as long as the loss rate remains above the threshold.
 - This is done on a per-flow basis. A proxy can send multiple RRMs in a short time if they are for different flows.
 - The RRMs contain current loss statistics for each current flow to the target source proxy.
 - Subsequent RRMs are periodic to avoid sending one for each packet received once we are above the acceptable loss rate.
 - Subsequent RRMs are necessary as a single RRM will not necessarily result in triage (as described below).
- RRMs use the control packet queue in the BPF and takes priority over data traffic.
- When an RRM is received at the source proxy, it updates state for the flows specified in the RRM and immediately forwards it to AMP in a push message (along with current flow statistics).
- AMP decides which, if any, flow should be triaged.
 - Only one flow can be triaged per triage_interval.
 - This gives the system time to adjust before making further triage decision.
 - This errs on the side of being less disruptive. E.g. if four flows are experiencing loss because we can only support 2, then we do not want to triage all 4 even if they all cross the loss threshold. Instead, we triage one at a time and stop when the remaining flows no longer cross the loss threshold.
 - If multiple flows are currently experiencing loss (above their threshold), AMP chooses to turn off the flow with smallest priority.
 - Flows are restarted after being off for a configurable number of supervisory control cycles. This is meant to be longer than the restart interval of the flow.
 - In the future, there can be other event-driven triggers to turn a flow back on – such as the termination of a flow, or increased capacity.
 - The first flow to trigger an RRM is very likely to be triaged.
 - At the source AMP, there will be only one candidate for loss triage.
- We apply a weight to the current measurement within the EWMA as a way of controlling the flow's sensitivity to loss.
 - A larger weight on the current measurement makes the flow more sensitive to loss and quicker to trigger an RRM.
 - This weight is currently set to be inversely proportional to the priority of the flow, making higher priority flows less sensitive.
 - This approach supports, to some extent, uncoordinated distributed triage.
 - If there are competing flows we want to triage incrementally (rather than triage all) to optimize utility.
 - If the flows are from the same source, AMP has a mechanism to triage multiple flows; however, if the flows are from different

- sources then each AMP will triage exactly one flow per triage interval.
- Instead, by scaling the sensitivity of a flow with its priority, lower priority flows will trigger an RRM faster than higher priority flows and this gives the higher priority flows some time to recover before triggering an RRM.

9 Multicast Group Management Sniffer (MGMS)

9.1 Summary of Goals and Objectives

The four main goals in creating a Multicast Group Management Sniffer (mgms) utility are: 1) eliminate the need to pre-configure multicast group memberships, 2) ensure that multicast applications work without modification, 3) eliminate the need to install and run utilities (e.g., helper apps) on application nodes, and 4) work with both IGMP-Adjacent and IGMP Non-adjacent multicast hosts.

9.2 High Level Design

As shown in Figure 26 below, the mgms application runs on the GNAT node and monitors and responds to group management signaling from multicast applications, including both IGMP-Adjacent and IGMP Non-adjacent hosts. mgms listens on the LAN-facing interface, eno1, for IGMP join/leave packets from the IGMP-Adjacent Host and PIM join/prune packets from the IGMP Non-adjacent Host. mgms tracks group membership state for individual LAN application hosts, combines the information with other local multicast receivers, and provides this information to the BPF. The BPF floods the local group membership information to all other GNAT nodes via GRAMs, and provides aggregated group membership information from all BPFs to the UDP proxy in the form of destination vectors. Destination vectors are provided to the UDP Proxy via the bin map in shared memory.

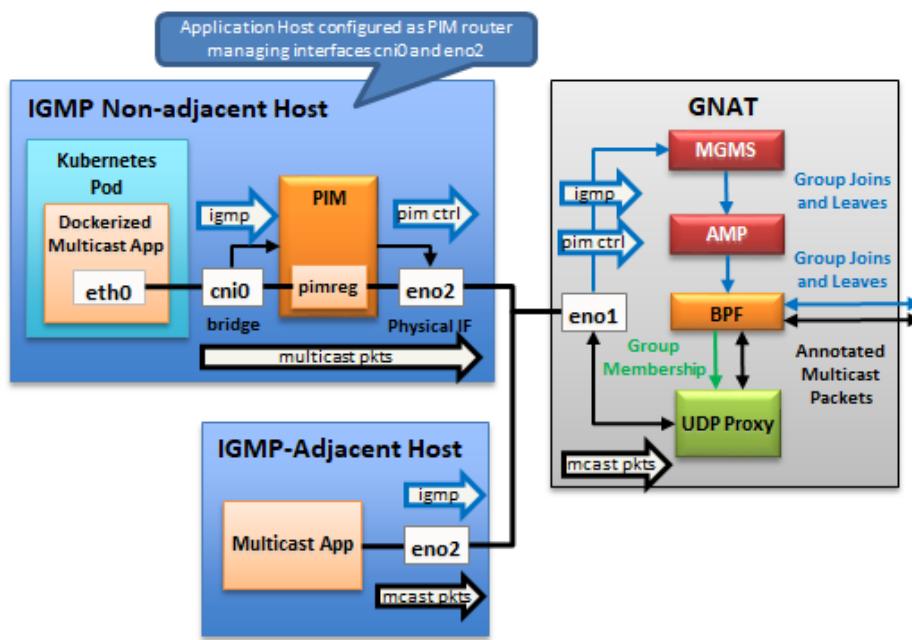


Figure 26. mgms monitors group management signaling from multicast applications to provide group join and leave information to the BPF. The BPF combines this information with group membership information from other BPFs to provide aggregated group membership information to the GNAT UDP Proxy.

9.3 Detailed Design

9.3.1 Receiving Multicast Group Management Packets

mgms receives multicast group management packets, IGMP join/leave packets and PIM join/prune packets, on the GNAT node's LAN-facing interface via a raw socket. A Berkeley Packet Filter (BPF) is attached to the raw socket that is configured to receive all IGMP packets and PIM prune/join packets on the LAN-facing interface. In Figure 26, this is interface eno1 on the GNAT node

9.3.2 Processing Received Multicast Group Management Packets

9.3.2.1 Processing IGMP Packets

mgms currently only processes IGMPv3 Membership Report messages. When an IGMPv3 Membership Report message is received, the group records in the received message are processed according to the group record type as described below.

- IGMPV3_CHANGE_TO_EXCLUDE or IGMPV3_MODE_IS_EXCLUDE group record type is interpreted as a group join. When there are no unicast addresses in the group record it is interpreted as a message to "join all sources". If there are addresses in the group record, it is interpreted as a message to "join the group but ignore the provided sources". In either case, mgms interprets the group record as a group join.
- IGMPV3_CHANGE_TO_INCLUDE group record type may be interpreted as a leave. When there are no unicast addresses in the report it is interpreted as a message to "leave all sources". If there are addresses, it is interpreted as a message to "leave the group for the provided sources". Only for the case of no unicast addresses does mgms interpret the group record as a group leave.
- IGMPV3_MODE_IS_INCLUDE, IGMPV3_ALLOW_NEW_SOURCES, and IGMPV3_BLOCK_OLD_SOURCES group record types are currently not supported.

9.3.2.2 Processing PIM Packets

mgms also processes received PIM join/leave packets, which identify the joined source addresses and pruned source addresses for each of the multicast groups in the packet. Note that mgms currently only supports addresses in the IPv4 address family in the received message. Each joined source address corresponds to a group join and each pruned source address corresponds to a group leave.

9.3.3 Multicast Group Membership Cache

mgms maintains a multicast group membership cache that is updated as IGMP join/leave and PIM join/prune packets are received and processed. The cache stores the adjacent host information for each of the multicast groups. When a new multicast group member is discovered, it is added to the cache. When the first member is added to a group in the cache, mgms notifies the BPF (via AMP) of interest in the multicast group. Subsequent joins for known multicast groups result in no information being provided to the BPF as it is already aware of the group interest. As

members leave a multicast group, the group membership cache is modified to reflect the leaving member. Only when the last member from the local (LAN-side) network has expressed a desire to leave the group does mgms notify the BPF that there is no longer any interest in the multicast group. Aggregation of this information by mgms reduces the amount of interaction between mgms and the BPF and simplifies the BPF as it does not need to track group membership information for each local multicast receiver.

10 Analysis Supporting GNAT Design Points

10.1 MatLab Model for Assessing Latency-constrained Error Control Trades

10.1.1 Problem Formulation

Consider a single link with channel packet loss probability p and deterministic one-way latency L . We wish to transfer packets across this link within a deadline D_{max} and with a source loss rate of less than ϵ_{max} . We assume that time is slotted and each slot can accommodate one packet. We can characterize each algorithm by:

- Its feasible region R which is the set of $(p, D_{max}, \epsilon_{max})$ for which the algorithm can successfully operate
- For operation within R , we can further characterize each algorithm by its achieved loss ϵ and its *efficiency* E which we define as the expected number of packet transmissions required to transfer one packet across the link.

10.1.2 Coded ARQ

We consider a class of algorithms called *Coded ARQ*. These algorithms operate as follows:

1. K source packets are collected and transmitted in slots $1 \dots K$. Then N coded packets are transmitted in slots $K+1 \dots K+N$. The coded packets are random linear combinations of the original K source packets. This is a systematic FEC code.
2. The packets arrive at the destination in slots $L+1 \dots L+K+N$. Packet losses occur and so in some of these slots there will be an “erasure signal”.
3. The destination sends feedback to the source indicating which source packets have been received and the number of addition “degrees of freedom” received from coded packet receptions.
4. The feedback is received at the source in L slots later at time $L+K+N+L$
5. The source then determines if it can send more packets to the destination
 - It uses the following simple rule: it will send more packets if $L+K+N$ (the time required to send the maximum number of packets) is the remainder of the latency budget.
6. If yes, then the source sends $T_i(s, c)$ packets to the destination, s is the total number of source packets received and c is the number of coded packets received and i is the round of transmission. The source will send $K_i = \min(T_i(s, c), K-s)$ systematic packets (source packets not yet received and $T_i(s, c)-K_i$ coded packets).
7. Increment i and go to 3.

Basically, this algorithm uses multiple rounds of systematic FEC to transfer packets from the source to the destination.

10.1.3 Analysis

10.1.3.1 Feasible Region

Let M be the maximum number of transmit cycles available to the source. That is, M is the largest integer satisfying

$$M(K + N + L) + (M - 1)L \leq D_{max}.$$

There are a number of cases.

10.1.3.2 Infeasible

If $L \leq D_{max}$, then $M=0$ and the problem is infeasible: the latency exceeds the deadline.

10.1.3.3 FEC

If $3L > D_{max}$, then $M=1$ and the source has enough time to get some packets to the destination, but not enough time to act on any feedback. Let J

$=D_{max}-L$. If $p^J \leq \epsilon_{max}$, then there exists a feasible FEC code which will achieve the required loss within the deadline.

10.1.3.4 ARQ

If $p_M \leq \epsilon_{max}$, then ARQ is a feasible algorithm.

10.1.3.5 Coded ARQ

If $M > 1$ and $p^M > \epsilon_{max}$, then ARQ cannot achieve the desired loss rate ϵ_{max} and it may be possible to improve on the performance of FEC by using feedback from the destination.

10.1.3.6 Efficiency

10.1.3.6.1 ARQ Without a Latency Constraint

If there is no latency constraint, the efficiency or expected number of transmissions to transfer a single packet (using feedback) is simply $1/(1 - p)$.

10.1.3.6.2 ARQ With a Latency Constraint

If ARQ is feasible given the experiment parameters, the probability of successfully transferring a packet in M rounds is simply $1 - p^M$ while the expected number of transmissions is $(1 - p^M)/(1 - p)$ and thus the efficiency is also $1/(1 - p)$. Thus the imposition of a latency constraint does not affect the efficiency of the algorithm! Note however, that our definition of efficiency is number of transmissions per packet delivered. If instead we defined it as the number of transmissions required by the algorithm per arriving packet (so we get no additional value for packets delivered above the requirement), then there would be a penalty associated with the latency constraint.

10.1.3.6.3 Normalized Efficiency

To make it easier to compare algorithms across different scenarios, we'll use a normalized efficiency $\tilde{E} = E/(1 - p)$, which is 1 when the latency constrained efficiency is equal to the unconstrained efficiency.

10.1.3.7 Loss and Efficiency Computation FEC

To compute the loss associated with using a systematic FEC code we first compute the probability of receiving exactly k packets. When $k < K$ this is the probability that we received exactly k systematic packets but received fewer than $K-k$ coded packets (and thus were unable to decode any additional packets). If $k = K$ then the probability is just that of receiving at least K packets (systematic or coded). Thus, we have:

$$p_k = \begin{cases} \binom{K}{k} (1-p)^k p^{K-k} \sum_{j=0}^{K-k-1} \binom{N}{j} (1-p)^j p^{N-j} & \text{if } k < K. \\ \sum_{j=K}^{K+N} \binom{K+N}{j} (1-p)^j p^{K+N-j}, & \text{if } k = K. \end{cases}$$

The probability of receiving a given packet is simply the expected number of packets received divided by K , so the loss rate is one minus that probability or

$$\epsilon = 1 - \sum_{n=0}^K kp_k/K$$

The number of packets transmitted is simply $K+N$, so the efficiency is

$$\sum_{n=0}^K kp_k/(K + N).$$

10.1.3.8 ARQ

If $P^M \leq e_{max}$, then ARQ is a feasible algorithm. There are (at least) two possible algorithms:

1. Keep sending until you either deliver the packet or run out of time.
2. Send up to M' times where . This value ensures that the probability of loss is less than e_{max} .

In either case, the normalized efficiency is $1 - p$ i.e., there is no penalty associated with the latency constraint compared to no latency constraint!

10.1.3.9 Coded ARQ

The Coded ARQ algorithms can be parameterized by the number of packets that are sent in each round, as a function of the receiver's state. The receiver's state is (j,k) which indicates j systematic packets received and k coded packets received. We will send more packets if time allows and if $j+k < K$. So there are roughly $MK(K+1)/2$ parameters each of which can take on (say) T values. The number of possible “feedback laws” for CodedARQ is enormous ($T^{\frac{1}{2}MK(K+1)}$) and thus an exhaustive search for the optimal CodedARQ algorithm is computationally intractable.

However, given a feedback law, it is relatively straightforward to compute the probability that k

of the original K packets have been received by the end of M rounds of transmissions (though not in closed form). Given a receiver state (j,k) we send an additional $T_i(j,k)$ packets which leads to a probability distribution over new receiver states (j,k) . By propagating this distribution over multiple rounds of transmissions, we can determine if/when we have met the ϵ_{max} loss constraint and the expected number of transmissions (and hence efficiency).

Figure 27 illustrates whether FEC, ARQ or CodedARQ is best as a function of p , ϵ_{max} and D_{max} . In the following, CodedARQ uses a simple (suboptimal) feedback law. In the first round it sends K systematic packets and N coded packets. In subsequent rounds it sends where $DOFN$ is the number of degrees of freedom needed (i.e., $K-j-k$). This feedback law basically uses the same ratio of total-packets-to-DOFs-needed for each round. For each data point, we optimize over K and N to find the most efficient combination.

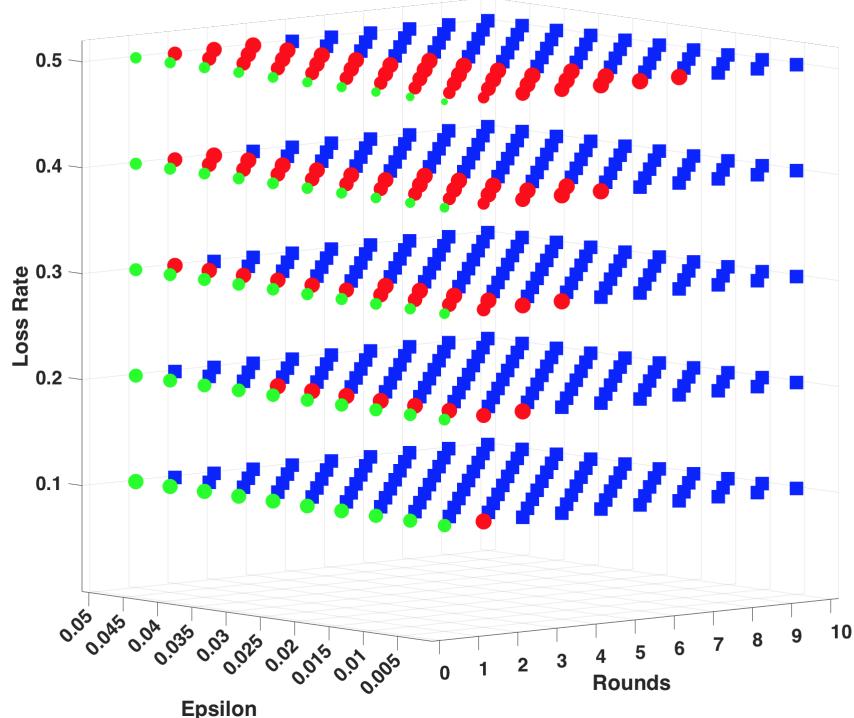


Figure 27. Color indicates which algorithm is best: FEC (green), CodedARQ (red) or ARQ (blue). The size of the dot indicates the normalized efficiency of the algorithm. Larger is more efficient. Note that ARQ has optimal normalized efficiency (1.0).

We also assume that K, N are constrained so that $K \leq 20$ and $N \leq 30$ and that $L \gg M$ so that the choice of K and N do not affect the number of possible rounds. Here we see that CodedARQ is best when the number of rounds (M) is limited and the probability of channel loss is large compared to the target source loss rate.

10.2 Analytical Basis for GNAT's Latency-Constrained Adaptive Error Control (AEC) Algorithm

10.2.1 Overview

The goal for developing AEC is creating an error control algorithm that can, whenever possible, achieve a target probability of packet delivery P_{tgt} over a given data path with a bounded amount of delay. The algorithm must 1) estimate the packet loss rates, bounds on the one-way and round trip delays, and the raw bit rate of the channel (path), and 2) choose an appropriate error control strategy that can achieve the target delivery probability within the allotted delay that also maximizes the efficiency with which the channel is used.

AEC supports three implicit modes of operation – pure Automatic Repeat Request (ARQ), pure Forward Error Correction (FEC), and a hybrid ARQ+FEC mode referred to as Coded ARQ (CARQ). Both ARQ and CARQ modes leverage feedback from the receiver and, depending on the one-way and round trip time delays compared to the delivery delay bound, may retransmit any missing packets one or more times. If sufficient time to close the feedback loop the required number of times (e.g., using a Bernoulli loss model and the estimated packet loss rate), AEC will preferably use pure ARQ since pure ARQ is 100% efficient in terms of channel usage. Unlike schemes incorporating any amount of FEC, with pure ARQ all packets received by the receiver are usable, and no extra or redundant packets are received.

In contrast, if there is insufficient time to close a feedback loop even once, pure FEC becomes necessary. Although this is the least efficient of the three modes, it is the only viable option: in fact, even pure FEC may not even be able to achieve the target delivery constraints.

When there is insufficient time to use pure ARQ, but enough time to close the feedback loop at least once, Coded ARQ allows using a smaller amount of FEC coding and hence is more efficient than pure FEC. With Coded ARQ, the amount of FEC coding required decreases with the amount of feedback allowed, and as a result makes more efficient use of the channel.

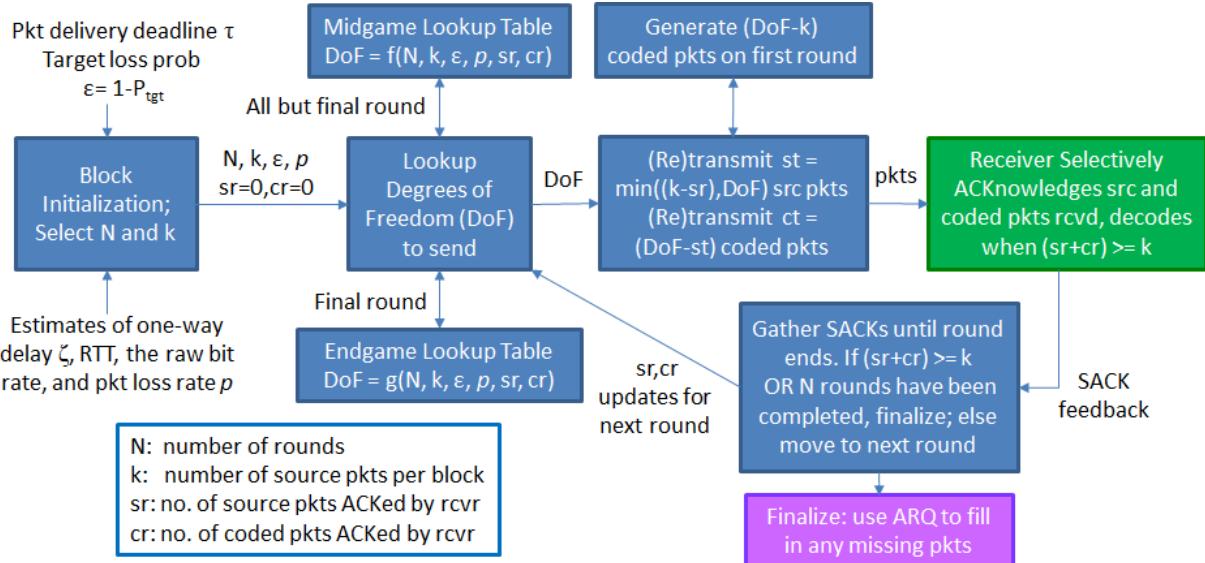
One of our design tenets in developing AEC is leveraging systematic codes for both pure FEC and CARQ. Systematic codes involve first sending all source packets in an encoding group in an unencoded form before sending any additional coded packets as repair packets. This has several attractive properties for use as a tunnel error control mechanism. First, any source packet received can immediately be forwarded downstream without needing to wait for a sufficient number of packets to arrive before decoding can occur. This allows keeping downstream channels fuller and mitigates queuing delays otherwise caused by injecting a large number of packets into the transmit queue following a decoding operation with non-systematic codes. Moreover, in situations where an insufficient number of packets are received (and hence decoding cannot occur), any source packets received can nonetheless be forwarded, increasing the overall packet receive probability.

10.2.2 Basic Error Control Model

A second design tenet is creating a capability where encoding modes and parameters can be adapted quickly, essentially allowing different encoding configurations for each encoding group. This in turn motivates an approach where decisions on mode and coding levels can be made on-the-fly and extremely quickly. Towards this end we have elected to use a pair of precomputed lookup tables: a mid-game table that is used for all but the final set of transmissions, and an end-game table that is used for the final set of transmissions. Each table lookup returns a "Degrees of Freedom" – DoF – value that specifies how many packets should be sent following the completion of a round of feedback. This, in addition to acknowledgements from the receiver in terms of what has already been received determines how many and which packets to retransmit in the next transmission round. This dual table approach has proven exceptionally fast, and is able to achieve performance levels very near the "optimal" performance levels predicted by our dynamic-programming-based offline optimization model described in Section 3.

AEC's error control process that leverages the dual table approach is shown in Figure 28. A new encoding block starts with the arrival of a new source packet, at which point AEC chooses a number of (re)transmission rounds N and a number of source packets per block k , based on: the target probability of receive P_{tgt} , estimates of the one-way delay from the sender to the receiver; the round trip time (RTT); and the packet loss probability P . The selected values of N and k , along with the target receive probability (actually, the maximum loss target $\epsilon = 1 - P_{tgt}$), the channel packet loss probability P , and the number of source and coded repair packets acknowledged by the receiver (sr and cr , respectively, with both set to zero initially) are then used to lookup the DoFs to send using one of the two tables. If $N = 1$ (implying pure FEC) the endgame table is used, otherwise the midgame table is used. Based on the returned DoF value, the sender will transmit $st = k$ source packets, then generate and send $ct = DoF - k$ coded repair packets.

Upon receipt of either a source or a repair packet, the receiver transmits an acknowledgement back to the sender using a cumulative ACKnowledgement (ACK) scheme (e.g., TCP ACKs) along with additional acknowledgement information in the form of Selective ACKnowledgements (e.g., TCP SACKs) of packets beyond the cumulative ACK to help identify both received and missing packets. Upon receipt of an ACK/SACK information covering the last packet transmitted from the encoding block, the transmission round is determined to be complete. If the number of transmission rounds is identically 1 (i.e., $N = 1$), the sender considers the AEC processing complete; if $N > 1$ the sender updates its knowledge of sr and cr based on the feedback from the receiver and performs another table lookup to determine the (hopefully reduced) number of DoFs to send. It then retransmits $st = \min((k - sr), DoF)$ source packets and $ct = DoF - st$ coded packets in the second round. This transmit/acknowledgement process repeats until the receiver has indicated that it has received enough packets to decode the group OR N rounds have been completed. If N rounds have been completed and the receiver still does not have enough packets to decode the encoding group, AEC drops into a pure ARQ mode to fill in any missing source packets which it does at a reduced priority. Packets transmitted in this "cleanup" mode are not expected to arrive within the target delay window, but provide increased reliability for the tunnel.



10.2.3 Mathematical Basis for AEC

10.2.3.1 Estimating Worst Case Delivery Delay

In order to select an (implicit) error control method (ARQ, CARQ, or FEC) and the associated encoding parameters N and k that can meet the delay constraints, we need to estimate worst case delivery delays.

To begin, we note that the worst case number of packets transmitted across all N rounds occurs when all packets through the first $N - 1$ rounds are lost. In this case, the (maximum) number of packets transmitted in each of the first $N - 1$ rounds is given by the midgame table with no source or encoded packets received – i.e., $\eta = f(N, k, \epsilon, p, 0, 0)$ – and the (maximum) number of packets transmitted in the final round is given by the endgame table with no source or encoded packets received – i.e., $\mu = g(N, k, \epsilon, p, 0, 0)$. The maximum number of packets transmitted is then $(N - 1)\eta + \mu$.

For a maximum size packet of σ bits and a channel bit rate of ρ bits per second, a bound on the forward packet serialization delay δ is given by

$$\delta = \frac{\sigma}{\rho}$$

In terms of feedback serialization delay, as SACKS are smaller than data packets we add one additional δ for feedback delay following receipt of the final packet sent in the sequence.

The worst case delay $\Delta_{N,k}$ for an N round configuration with k source packets is then given by

$$\Delta_{N,k} = (N - 1)((\eta + 1)\delta + RTT_{max}) + (\mu\delta + \zeta_{max})$$

where RTT_{max} and ζ_{max} are the maximum observed values of Round Trip Time and one-way delay, respectively. Using the maximum values accounts for the impact of any queueing delays in the system, which is important since queuing delays can be affected by congestion control actions taken by the tunnel itself.

10.2.3.2 AEC Operating as a Pure ARQ Error Controller

Assuming a Bernoulli loss model with packet loss probability p and target maximum loss rate $\epsilon = 1 - P_{tgt}$, the effective loss rate P_{loss} after N' transmissions using an ARQ based scheme is given by

$$P_{loss} = p^{N'} \leq \epsilon \implies N' \geq \text{ceil}\left(\frac{\log(\epsilon)}{\log(p)}\right)$$

Hence if the delivery deadline τ is such that $\tau > \Delta_{N'}$, then there is sufficient time to close the feedback loop to achieve the target receive probability using only ARQ (Note: ARQ uses source packets individually, hence $k=1$ here). Since ARQ is 100% efficient, scanning the tables for the most efficient encoding that can meet the delay constraint τ and target receive probability constraint P_{tgt} will yield N' with a k of 1, indicating a pure ARQ mode of operation.

Note that there may be multiple different table entries that can meet the delay and target receive probability constraints, i.e., there may be other viable values of N and k where $N \geq N'$ and $k \geq 1$. In this case we choose the combination with the smallest delivery delay, which are identically $N = N'$ and $k = 1$; i.e.

$$\min_{N,k}(\Delta_{N,k}) = \Delta_{N',1}$$

10.2.3.3 AEC Operating as a Pure FEC Error Controller

The effective receive probability P_{recv} for a length n systematic code with k source packets and $m = n - k$ coded repair packets is given by

$$P_{recv} = \sum_{i=k}^n \binom{n}{k} p^{n-i} (1-p)^i + \frac{1}{k} \sum_{i=0}^{k-1} i \binom{k}{i} p^{k-i} (1-p)^i \left(\sum_{j=0}^{\min(m,k-i-1)} \binom{m}{j} p^{m-j} (1-p)^j \right)$$

The first summation term is the packet delivery success that can be achieved when k or more packets are received in an encoding block, which means that the receiver can reconstruct all of the source packets. The second double summation term is the additional packet delivery success that occurs when less than k packets are received, yet some of the packets are source packets and so can still be used as they are unencoded using a systematic coding strategy

For FEC, $N=1$; hence the worst case delay for FEC using the model described earlier is identically $\Delta_{1,k} = \mu\delta + \zeta_{max}$. A notional strategy for choosing FEC parameters then would be to first choose the number of DoF to send n such that

$$\Delta_{1,k} = n\delta + \zeta_{max} \leq \tau \implies n \leq \frac{(\tau - \zeta_{max})}{\delta}$$

and then choose k by looking through the endgame table for values $\mu = g(1, k, \epsilon, p, 0, 0)$ such that $\mu \leq n$. Note that there may be multiple values k and associated values μ that meet these conditions, in which case we would choose the most efficient.

The question of which FEC parameters yield the most efficient solution hints at a bigger problem however. Unlike the ARQ case which is 100% efficient – and so it is the clear choice if the constraints are met – FEC may or may not be more efficient than CARQ. Specifically, depending on the values τ , ζ_{max} , and RTT_{max} , it may be the case that there is sufficient time to close the loop and use CARQ (albeit with shorter codes) to achieve the same target packet receive probability and delay bounds. In this situation, it is likely that a CARQ solution is more efficient than pure FEC, and so is preferable. Hence we must first determine which configurations of FEC and CARQ can be used to achieve the target delivery probabilities and delay constraints, and then compare the various alternatives to determine which is the most efficient solution and hence the preferred technique. We defer discussion of efficiency measures until after the following discussion on CARQ.

10.2.3.4 AEC Operating as a CARQ Error Controller

A mathematical model of CARQ performance can be constructed as follows. First, assume we have decided to use N rounds of CARQ with a number of source packets k . Let $S_{x,y,z}$ be the state where y source packets and z coded packets have been ACKed by the end of round x , with $0 \leq x \leq N$. Initially $x = 0$ no packets have been transmitted hence the starting state $S_{0,0,0}$ has probability 1 (i.e., $P(S_{0,0,0}) = 1$) and the probability of all other starting states is 0 (i.e. $P(S_{0,y,z}) = 0 \forall \{x, y\} \neq \{0, 0\}$).

Next, let $d_{x,y,z}$ be the DoF transmitted (or retransmitted) when the sender is in state $S_{x,y,z}$. Trivially $d_{x,y,z} = 0$ when $(y+z) \geq k$ as the receiver already has sufficient DoF to decode.

Since we (re)transmit any missing source packets ahead of any missing coded packets, the number of source packets (re)transmitted $st_{x,y,z}$ at the beginning of round x is then $st_{x,y,z} = \min(d_{x,y,z}, k - y)$ and the number of coded packets (re)transmitted $ct_{x,y,z}$ at the beginning of round x is then $ct_{x,y,z} = d_{x,y,z} - st_{x,y,z}$. Upon receipt and ACKnowledgement of packets (re)transmitted from state $S_{x,y,z}$, the set of possible next states is $S_{x+1,y',z'}$ where $y \leq y' \leq y + st_{x,y,z}$ and $z \leq z' \leq z + ct_{x,y,z}$.

The probability of transitioning from state $S_{x,y,z}$ to state $S_{x+1,y',z'}$ is then

$$P(S_{x,y,z}, S_{x+1,y',z'}) = \binom{st_{x,y,z}}{y'-y} p^{st_{x,y,z} - (y'-y)} (1-p)^{y'-y} \binom{ct_{x,y,z}}{z'-z} p^{ct_{x,y,z} - (z'-z)} (1-p)^{z'-z}$$

For completeness, we note the following implications on state transition probabilities given DoF values $d_{x,y,z}$:

$$d_{x,y,z} \geq 0 \implies P(S_{x,y,z}, S_{x+1,y',z'}) = 0 \text{ when } y' \leq y \text{ or } z' \leq z$$

$$d_{x,y,z} = 0 \implies P(S_{x,y,z}, S_{x+1,y',z'}) = 1 \text{ when } y+z \geq k \text{ and } (y' = y \text{ and } z' = z)$$

$$d_{x,y,z} < 0 \implies P(S_{x,y,z}, S_{x+1,y',z'}) = 0 \text{ when } y+z \geq k \text{ and } (y' \neq y \text{ or } z' \neq z)$$

The probability of state $S_{x+1,y',z'}$ is given by

$$S_{x+1,y',z'} = \sum_{i=y}^{y'} \sum_{j=z}^{z'} P(S_{x,i,j}) P(S_{x,i,j}, S_{x+1,y',z'})$$

The packet receive probability P_{recv} is then

$$P_{recv} = \sum_{i=0}^k \sum_{j=k-1}^{max(j)} P(S_{N,i,j}) + \frac{1}{k} \sum_{i=0}^{k-1} i \sum_{j=0}^{k-i-1} P(S_{N,i,j})$$

where $max(j)$ is the maximum number of coded packets that may be delivered to the receiver for a given encoding block. For AEC this is $max(d_{0,0,0} - k, d_{N-1,0,0} - k)$.

The parameter selection process is then to search through the midgame and endgame tables and choose the values of N and k that maximize the efficiency and have the property that $\Delta_{N,k} \leq \tau$.

We note here that this set of expressions works for the case $N = 1$ and so this strategy covers the FEC case as a proper subset. Now we just need a method for computing efficiency.

10.2.3.5 Encoding Efficiency For FEC and CARQ

To complete the selection algorithm for deciding between FEC and CARQ, we need a measure of efficiency. Here we define encoding efficiency as the number of *usable* packets received divided by the *total* packets received. The difference between the number of packets received and the number of usable packets received is the number of *unusable* packets received.

There are two ways that a received packet may be unusable. The first is if there are *excess* packets received – i.e., *more* than k packets are received for any given encoding block. The second is if there are *too few* packets received, – i.e., *less* than k packets are received for a given encoding block – and some of those are encoded packets (vice source packets). In this second case, any

encoded packets will be unusable, since insufficient DoFs are available to perform the decoding function. Any unencoded source packets received are of course usable.

We now define the *block delivery efficiency* as the expected number of usable packets received divided by the total packets received in the encoded block: i.e.,

$$\text{efficiency} = \frac{E(\text{usable})}{E(\text{total})} = \frac{E(\text{usable})}{E(\text{usable}) + E(\text{unusable})} = \frac{kP_{\text{recv}}}{kP_{\text{recv}} + E(\text{unusable})}$$

Using the notation developed in the previous section, $E(\text{unusable})$ for systematic codes is given by

$$E(\text{unusable}) = k \left[\sum_{i=0}^k \sum_{j=k+1-i}^{\max(j)} P(S_N, i, j) \right] + \sum_{i=0}^{k-1} \sum_{j=0}^{k-i-1} j P(S_N, i, j)$$

Note again that for the FEC case, $N = 1$.

10.2.3.6 Constructing the Midgame and Endgame Tables: Choosing DoF-To-Send Values

A key question now is how to choose the degrees-of-freedom to send in each state at run time. We *could* try to use the Dynamic Programming model from our Markov Decision Process (MDP) used to derive the (near) optimal solution for constrained rounds, however it is computationally expensive. Instead we use the (approximate) pre-computed two-table model described above to turn run-time decision making into a series of simple table lookups.

The motivation for AEC's two table configuration originates with an analogy of ARQ. Specifically, with ARQ, the loss probability is reduced by the same multiplicative factor in each (re)transmission round. Using the same midgame table at each round (except the last) mimics this core retransmission strategy, effectively reducing the expected number of missing packets with each round and thereby minimizing the number of extra (coded) packets to maximize efficiency. However, reaching the final round indicates that the required number of DoFs have not yet been received despite repeated attempts, and since the last round is the last chance to achieve the target packet receive probability, the endgame table in general is more aggressive.

In creating the two lookup tables we start with the following mathematical analysis. Let p_{D_R} be the probability of reaching decodable state D_R at the completion of round R , and p_{U_R} be the probability of reaching undecodable state U_R by the end of round R . Further, let p_{S_R} be the probability that the total number of packets received by the completion of round R is $\geq k$ given that an insufficient number of packets were received by the completion of the previous round $R - 1$. Then we may express $p_{D_R} = p_{S_R} (1 - p_{D_{R-1}})$, and by expanding the recursion, $p_{D_R} = p_{S_R} \sum_{i=1}^{R-1} (1 - p_{S_i})$. Similarly, $p_{U_R} = \sum_{i=1}^R (1 - p_{S_i})$. Using these expressions, the total probability of successfully decoding (which may complete at any round) is then $p_D = \sum_{i=1}^N p_{D_i}$, and the probability of being unable to decode after N rounds is given by $p_U = p_{U_N} = \sum_{i=1}^N (1 - p_{S_i})$.

These relationships are illustrated in Figure 29 below for an $N = 3$ round example. Initially we start in an undecodable state U_0 , and use the midgame table to look up the DoF to send. At the end of the round, we may be in decodable state D_1 and we terminate the algorithm, or we are in undecodable state U_1 and we move to the next round. If we move to the second round, we again use the midgame table, along with information about which packets have been received to lookup the DoF to send. At the end of the second round we may now be in decodable state D_2 and we terminate, or we are in undecodable state U_2 and we move to the next (and final) round. On the final round we instead use the endgame table, and end up either in decodable state D_3 or undecodable state U_3 .

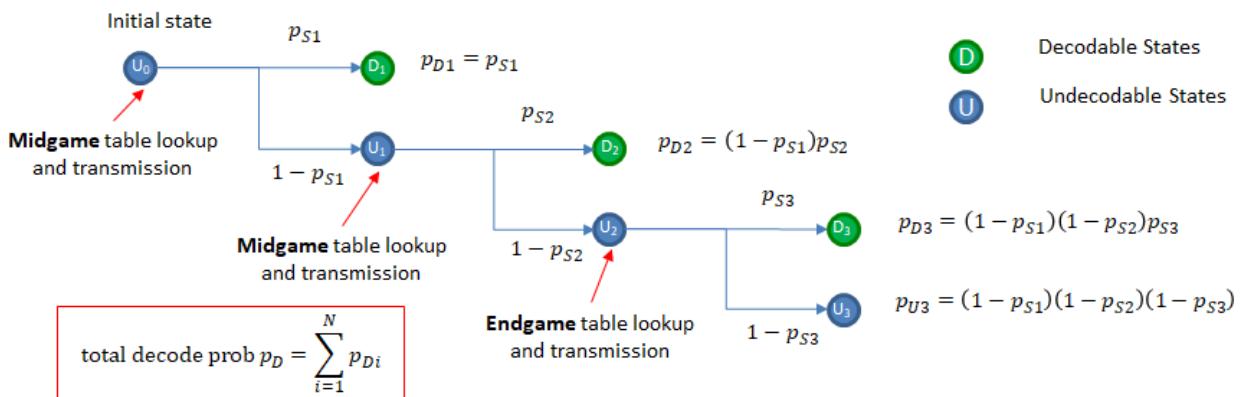


Figure 29. Calculation of decoding probabilities after each transmission round.

We now present the AEC table construction procedure. The notation here follows the model and notations developed in Section [4.3.3](#)

AEC Table Construction Procedure:

For each of a given set of encoding parameters N, k, ϵ , and P , we execute the following search loop.

For candidate target midgame probabilities of receive P_{mid} over the range $0 < P_{mid} < 1$:

Compute the DoF-to-send values $d_{0,y,z}$ for the midgame table as follows: (note: with our two table model)

For each value y of received and ACKnowledged source packets $0 < y < k - 1$:

For each value z of received and ACKnowledged coded packets $0 < z < k - 1 - y$:

Find $d_{0,y,z}$ for table position y, z such that the conditional packet probability of receive $P_{0,y,z} > P_{mid}$, with $P_{0,y,z}$ given by

$$P_{0,y,z} = \sum_{i=k-(y+z)}^{d_{0,y,z}} \binom{d_{0,y,z}}{i} p^{d_{0,y,z}-i} (1-p)^i \geq P_{mid}$$

Using the Markov model from the CARQ section and the midgame table of $d_{0,y,z}$ values, compute the state probabilities $P(S_{N-1,y,z})$ for the next-to-final round

Compute the packet receive probability P_{ntfr} at the completion of the next-to-final round, with P_{ntfr} given by

$$P_{ntfr} = \sum_{i=0}^k \sum_{j=k-i}^{\max(j)} P(S_{N-1}, i, j)$$

Compute an endgame target probability of receive $P_{end} = \frac{P_{tgt} - P_{ntfr}}{1 - P_{ntfr}}$

Compute the DoF-to-send values $d_{N,y,z}$ for the endgame table as follows:

For each value y of received and ACKnowledged source packets $0 < y < k - 1$:

For each value z of received and ACKnowledged coded packets $0 < z < k - 1 - y$:

Find $d_{N,y,z}$ for table position y, z such that the conditional packet probability of receive $P_{N,y,z} > P_{end}$, with $P_{N,y,z}$ given by:

$$P_{N,y,z} = \sum_{i=k-(y+z)}^{d_{N,y,z}} \binom{d_{N,y,z}}{i} p^{d_{N,y,z}-i} (1-p)^i + \frac{1}{k} \sum_{i=0}^{ubi} (i+y) \binom{yts}{i} p^{yts-i} (1-p)^i \left(\sum_{j=0}^{ubj} \binom{zts}{j} p^{zts-j} (1-p)^j \right)$$

where

$$yts = \min(d_{N,y,z}, k - y)$$

$$zts = \max(d_{N,y,z} - yts, 0)$$

$$ubi = \min(d_{N,y,z}, k - (y + z) - 1)$$

$$ubj = \min(zts, k - (y + z) - 1 - i)$$

Using the Markov model from the CARQ section and the endgame table of $d_{N,y,z}$ values, compute the state probabilities $P(S_{N,y,z})$ for the final round

Compute the efficiency for this {midgame, endgame} table pair using the final state probabilities

Choose the {midgame,endgame} table pair with the greatest efficiency, and store the corresponding midgame target P_{mid} as the best solution for the given set of parameters N, k, ϵ , and p .

End AEC Table Construction Procedure

Note that the parameter P_{mid} is sufficient to regenerate the $d_{0,y,z}$ and $d_{N,y,z}$ values for the mid-game and endgame tables. For convenience, we also store the endgame target P_{end} .

10.2.4 Midgame and Endgame Run-time/Implementation Details

The above table construction procedure process is implemented in the utility *aectablegen* located in the iron/util/aectablegen directory. This is used to generate an include file that is referenced by SLIQ at compile time. Then, for both speed and convenience, we pre-compute and store the DoF-to-send tables and associated efficiency values so that the table search and decision making processes are as fast as possible.

10.2.5 Simple Example Showing AEC Processing

We now present a simple example to help visualize AEC run-time processing. Assume that the number of source packets in an encoding block $k=4$. Note that the tables are used to lookup DoF-to-send values based on the current number of ACKnowledged source and coded packets, so we never need to lookup values when the total DoFs ACKnowledged (total of the source and encoded packets) is greater than or equal to 4; this is because when the DoF at the receiver is equal to k the encoding group is decodable and so further transmissions are not needed. Hence both the midgame and endgame tables will be lower triangular with dimensions of 4x4.

For this example we use a packet loss rate of $p=0.3$, a target packet receive probability $P_{tgt}=0.995$, and the number of round $N=3$. The midgame and endgame tables are shown in Figure 30 below.

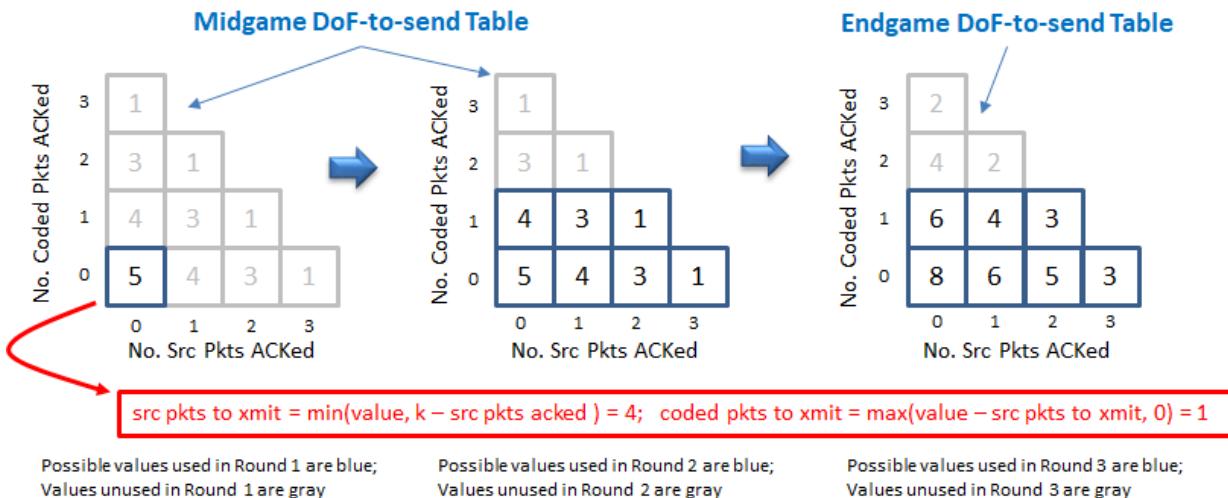


Figure 30. Simple example showing possible DoF-to-send values for each of N rounds based on the midgame and endgame lookup tables.

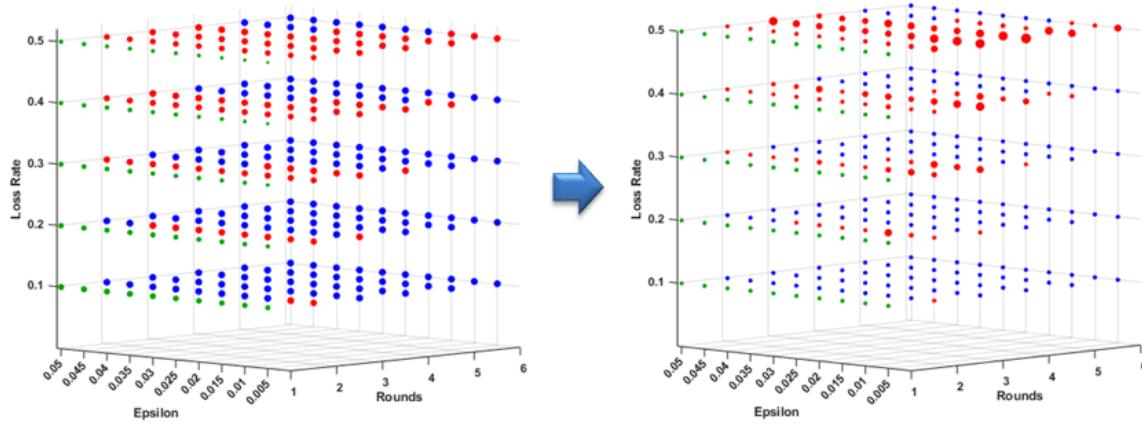
At the start, both the number of ACKnowledged source and ACKnowledged encoded packets are of course 0, hence the only possible DoF-to-send value for the first round is $f(3, 4, 0.05, 0.3, 0, 0) = 5$ as shown in the lower left of the leftmost figure. This implies that $k=4$ source packets will be sent, and that $5-4=1$ coded packets will be sent in the first transmission.

In the second round, the possible number of source packets that may be ACKnowledged ranges from 0 to 3 (if 4 are ACKnowledged then the algorithm terminates after the first round) and the possible number of coded packets that may be ACKnowledged ranges from 0 to 1. Hence anywhere from 1 to 5 packets may be transmitted in the second round. The collective region is shown highlighted in blue in the middle figure, along with the corresponding DoF-to-send values.

In the final round, the possible number of source packets that may be ACKnowledged again ranges from 0 to 3, and the possible number of coded packets that may be ACKnowledged again ranges from 0 to 1. As a result the DoF-to-send values may range from 3 to 8 as highlighted in the rightmost figure. For those cases where no coded packets have been received, the DoF-to-send values exceed $k - sr$, and so will result in transmissions of coded packets. We see from the rightmost figure that the endgame table is more aggressive than the midgame table (i.e., the endgame table will specify sending a greater number of coded packets for the same number of ACKnowledged source and coded packets) as described earlier. We also see that for some cases DoF-to-send values will require generating more than the single encoded packet required in the first round.

10.2.6 Analytical Comparison of AEC Performance with the (near) Optimal Solution

Here we provide a visual comparison of AEC performance with the achievable performance predicted by the Dynamic Programming (DP)-based optimizer. The choice of error control algorithm selected by the DP-based optimizer is shown on the left of Figure 31, with colors used to indicate the algorithm choice, and circle sizes used to indicate the relative efficiency values. The corresponding performance of AEC for these different operating points is shown on the right of Figure 31, with colors again used to indicate the algorithm choice, and circle sizes now used to indicate the difference between AEC and the DP-derived solution. As indicated in the Figure annotations, the largest difference between AEC and the DP-derived solution is 1.5%, with the average difference being less than 0.6%.



Most efficient algorithm as a function of number of rounds, packet loss rate (PER) and ϵ (target loss rate) using a ***Markov Decision Process (MDP)** model with a Dynamic Program-based optimization

- FEC (**green**)
- Coded ARQ (**red**)
- ARQ (**blue**)

Difference in efficiency between AEC and MDP solutions.
Larger circles indicate greater differences.

- Largest difference is 1.5%
- Smallest difference is 0%
- Average difference is < 0.6%.

AEC efficiency ranges from 60% (FEC) to 100% (ARQ)

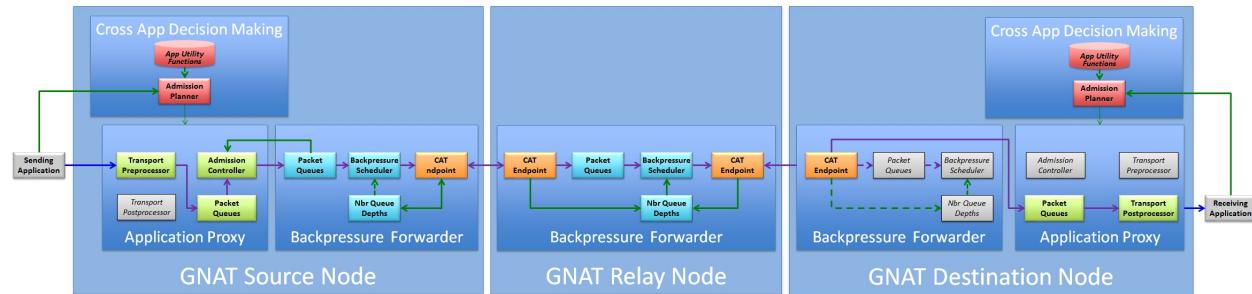
Figure 31. Analytical comparison of AEC performance with that obtained from a DP-optimized solution over a range of packet loss rates, target packet loss rates, and the allowed number of rounds.

Measured performance of the AEC implementation is provided in Performance Assessment Document.

11 Use Cases

11.1 A Day in the Life of a Packet

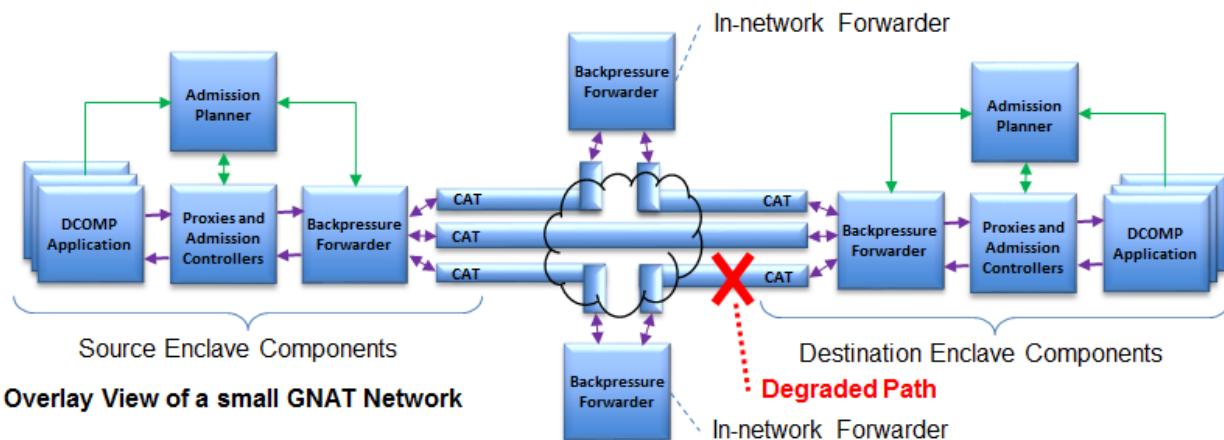
In this section we describe a simple scenario with the goal of laying out the steps involved in relaying a packet from a source application to a destination application.



1. Working from Left to Right, the sending application generates an application packet and sends it to the GNAT node.
2. The Application Proxy receives the application packet and performs any performance enhancement preprocessing (e.g., Forward Error Correction) to create proxied packets, which are then place in a local packet queue.
3. The Admission Controller within the Application Proxy computes the *flow ID* for the proxied packet and uses it to determine the associated Utility Function. It then uses the Utility Function and queue length associated with the *Bin ID* to determine the rate at which proxied packets from this flow should be accepted.
4. Once the proxied packet is accepted, it is forwarded to the Packet Queues within the Backpressure Forwarder where it is placed in queue associated with its *Bin ID*
5. The Backpressure Scheduler removes proxied packets from the Packet Queues when transmit opportunities occur on the Tunnel Interfaces. These decisions are based on the local queue lengths as well as the queue lengths reported by the neighboring GNAT nodes.
6. The proxied packet is then transmitted to the Relay GNAT node.
7. The Relay node receives the proxied packet from a Tunnel Interface and, since the packet has already been accepted into the system, places it directly in the Packet Queues.
8. The Backpressure Forwarder eventually de-queues the proxied packet and gives it to a Tunnel Interface.
9. The proxied packet is then transmitted to the Destination node.
10. The Destination node Tunnel places the proxied packet directly into the Packet Queues of the Application Proxy for any performance enhancing post-processing (e.g., FEC decoding, packet reordering, jitter reduction) to recover original application packets.
11. The Application Proxy sends recovered application packets to the receiving application.

11.2 Response to a Link Failure

In this section we provide an example of how GNAT responds to network faults and attacks. Referring to the simple topology in the figure below, a source enclave on the left is connected to a destination enclave on the right. There are three overlay paths between the source and destination enclaves. One of the overlay paths directly connects the two enclaves over a single CAT. The two other overlay paths indirectly connect the source and destination through intermediate backpressure forwarders located within the network, each path involving a pair of CATs.



Initially there are multiple concurrent application flows from the source to the destination that the backpressure forwarder distributes over the three different paths. A segment of the path through the bottom-most forwarder then fails as indicated by the red X. At this point the CAT associated with that segment no longer receives acknowledgements and stops accepting packets. This causes packets to start accumulating in queues at the bottom forwarder. When sufficient packets have built up, there is no longer a positive queue differential between the source forwarder and the bottom forwarder. At this point, the source forwarder will no longer provide packets to the interconnecting CAT, and will instead attempt to deliver packets at a faster rate over the two remaining paths.

If the increased rates cannot be supported by the remaining paths, packets start accumulating in the queues at the source forwarder itself. This causes the packet-level admission control in the proxies at the source to slow down delivery of packets to the source forwarder in proportion to their respective utility functions. Initially this reduces the transmission rate on the elastic flows as the inelastic flows will try to maintain their minimum rate. If any lower utility inelastic flows fall below their critical rates, they self-regulate to a zero rate, and any elastic flows may actually increase delivery rates to fill the gap created from inelastic flows dropping out. Any new application flows that attempt to start during this period may be disallowed by the admission planner; alternately they may be allowed to start if they are of sufficiently high utility, resulting in the reduction or termination of other ongoing flows. Once the failed link is restored, the associated CAT resumes accepting packets and the queues in the bottom forwarder begin to drain. This restores the queue differential, allowing the source forwarder to resume sending packets along the bottom-most path.

11.3 Distributed Dynamic Load Balancing of Multicast Traffic

A key goal for GNAT is being able to local sense and dynamically respond to downstream congestion in order to deliver multicast packets to multiple receivers. A simple example is shown below, where the source node (Node 0) needs to send to 3 destination nodes (nodes 5, 6, and 7) over the network topology represented by nodes 1, 2, 3, and 4. The capacities of each interconnecting link are labeled, and change over time.

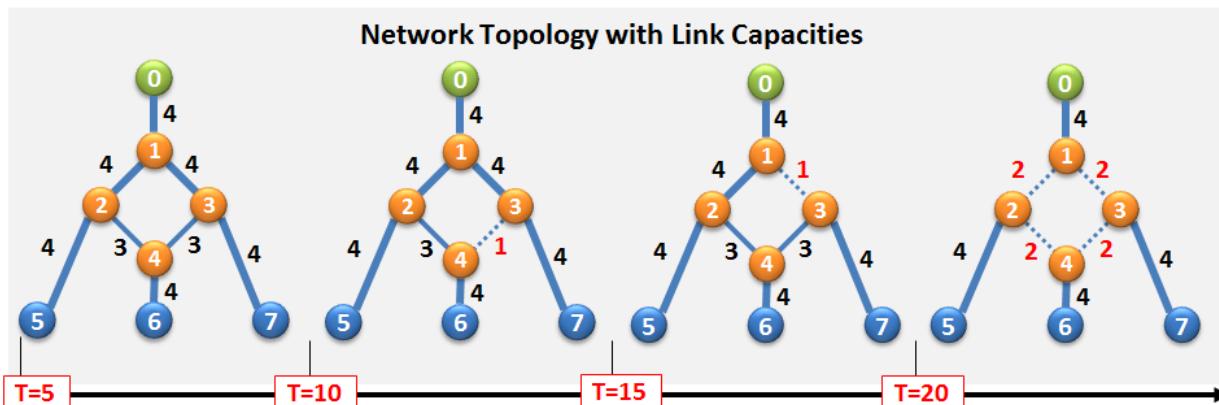
To get started, we note that traditional multicast routing chooses a single path from a given source to each destination in the multicast receiver group. Efficient multicast tree formation attempts to do so in a way that uses the least amount of resources, for example by choosing link where a single multicast packet can be used to service multiple downstream nodes. Nonetheless, traditional multicast routing does not attempt to use multiple paths from any given source to reach any given destination, and so from the perspective of a given source destination pair looks very much like standard single path routing and forwarding for unicast applications.

Using a simple multicast tree as a baseline, we see that at time T=5 the topology is such that the maximum capacity that can be sustained to all multicast receivers is 3. To see this, note that node 6 is only reachable via network node 4, and node 4 can only be reached from node 0 via a link with capacity 3. Although the other two nodes in the multicast receiver group (nodes 5 and 7) can be reached with rates as high as 4, getting the same data to all members in the receiver group forces the group rate to be the minimum of all individual rates, and hence the maximum rate that can be used is 3.

At time T=10 we see that the capacity on the link from node 3 to node 4 has been reduced to 1. Hence if the multicast tree construction inadvertently choose this path, the maximum sustained rate for the multicast group would be reduced to 1.

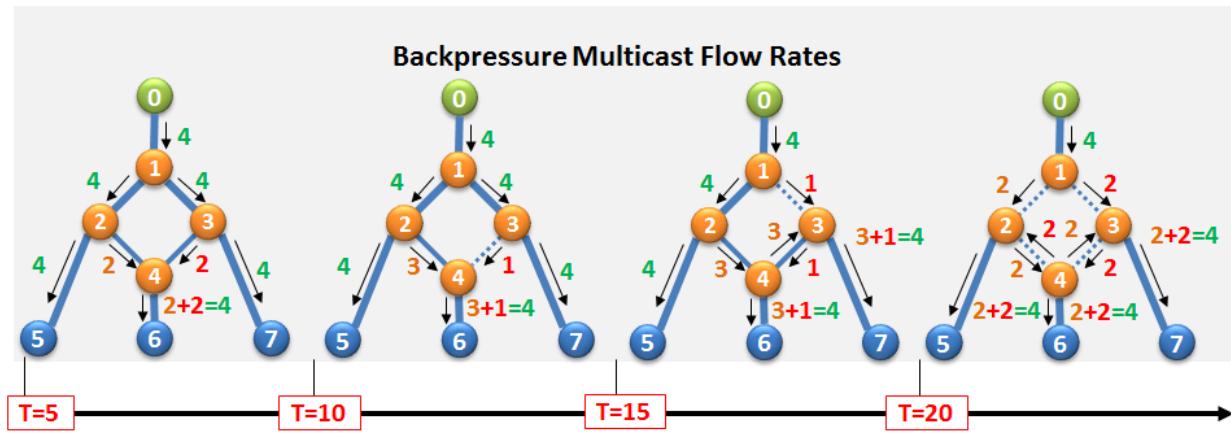
At time T=15, we see that the capacity on the link between node 3 and node 1 has been restored, however the capacity from node 1 to node 2 is now reduced to 1. As with the previous set of link speeds, if the multicast tree construction inadvertently chose the link from 1 to 3 as path of the tree, the maximum sustained rate for the multicast group would be reduced to 1.

At time T=20, we see that the capacity on all of the network nodes has been reduced to 2. Hence the maximum sustained rate for the multicast group would be reduced to 2, regardless of how the multicast tree is constructed.



In contrast, GNAT's use of backpressure-based forwarding allows the use of multiple paths to reach any given destination. Hence GNAT is able to sustain a rate of 4 through all of the above

network topologies by using different links to carry different amounts of traffic, even from a single source to a single destination as shown below.



12 Support Tools

12.1 LinkEm

Our LinkEm implementation is (very loosely) derived from Lincoln Laboratory's publicly released link emulator by the same name. At a high level, both LinkEm implementations operate as ethernet bridges between a pair of Linux interfaces, and use a busy-wait loop to control timing accuracy. Both implementations model the effects of queuing with limited buffer sizes to mimic bottleneck routers, with the ability to explicitly set the amount of buffering allowed. Both implementations model the effects of serialization and propagation delays. Both implementations support modeling various types of packet loss, including losses based on both bit error rates and packet error rates. Both support dynamic control over modeled characteristics using a separate application called LinkEmClient.

That said, the LinkEm capability included with GNAT is a complete reimplementation, and includes several new features. Our implementation supports jitter modeling, as well as a Gilbert-Elliott burst error model. To support modeling of more complex networks (vice simple links/paths), our LinkEm implementation supports:

- Modeling of concurrent, parallel links/paths, so that side-by-side comparisons of application behaviors with and without GNAT can be made in real-time
- Modeling of multiple different network paths with separately settable path characteristics based on subnets. This allows modeling the effects of different paths from a given source to different destinations independently using a single emulator located at the edge of the network.
- Modeling of access links where the multiple different paths to different networks pass through a single, common link/path emulator with separately settable set of parameters.

These latter capabilities are described in later sections.

We note here that our reworked design has focused on extreme performance levels. It avoids allocations and deallocation by using pre-allocated packet pools, and avoids all memory copies except into and out of the kernel on read and write calls. It is very accurate and responsive. It uses the x86 rdtsc processor instruction (a single clock cycle instruction) to obtain timing information. Since the rdtsc instruction is called directly from the LinkEm executable, it is extremely low overhead; unlike the more familiar gettimeofday system utility, no kernel calls are involved. Moreover the entire LinkEm process runs in a busy loop and is typically assigned very high processor priority. As such there are no lags caused by LinkEm giving up its processor core. Although this makes LinkEm CPU intensive for the processor it is assigned, it is extremely well regulated and highly accurate.

12.1.1 LinkEm Basics

LinkEm runs on a Linux host nominally located in-line between 1) an application host, service host, or LAN, and 2) another application host, service host, LAN or WAN.

In the discussions below we'll consider configurations where LinkEm is being used as a bridge between a LAN and WAN; however this is not a requirement. At its core, LinkEm is a dynamically controllable software bridge that regulates the flow of traffic between a pair of interfaces. Hence LinkEm can be used wherever a bridge (i.e., an ethernet switch) can be used.

As a simple example, if we want to regulate the flow between two interfaces em2 and em3, the LinkEm command used to start LinkEm looks like

./LinkEm -1 em2 -2 em3 &

All remaining configuration can be performed while LinkEm is running using LinkEmClient.

Note 1: In the example command above, em2 is designated as interface 1, and em3 is designated as interface 2. By convention, interface 1 is the WAN facing interface, and interface 2 is the LAN facing interface. This distinction is important when modeling complex networks where we want to model multiple different paths within a larger WAN; however for simpler environments where modeling only a single link or path is needed (say, bridging two LANs) the distinction of which interface is designated as interface 1 and which is designated as interface 2 does not matter and is irrelevant.

Note 2: LinkEm assumes that when it is launched that the interfaces it will control are in the "up" state and that they are not otherwise in use as bridge or router interfaces.

Note 3: LinkEm opens raw sockets in order to send and receive traffic on the designated interfaces. Doing so in general requires root permissions. Hence we typically run LinkEm using *sudo*.

Note 4: LinkEm need not be placed in the background as shown; for illustration purposes, we do so here to allow running the control application LinkEmClient in the same session.

12.1.2 Controlling LinkEm Using LinkEmClient

Typically we control LinkEm's behavior by modifying capacities, delays (including jitter), and loss rates to model network dynamics. This is accomplished using LinkEmClient. For example, once LinkEm is running, we can **throttle** the rate to 20 Mbps (using **-t**), the one-way **delay** to 1ms (using **-d**), and amount of **buffering** used to 50kbytes in order to model a bottleneck router (using **-b**) by running the command shown below on the LinkEm host

./LinkEmClient -t 20000 -d 1 -b 50000

The path capacity can subsequently be throttled back to 10 Mbps, keeping all other parameters the same, using the command

./LinkEmClient -t 10000

Note that using LinkEmClient does **not** require root privileges.

Note also that LinkEmClient can be used to control a remote LinkEm (i.e., a LinkEm running on a different machine), and that there are a number of other options including querying and statistics reporting available. These as described in the section entitled "Controlling LinkEm at Run-Time" below.

12.1.3 Modeling Concurrent Parallel Networks

LinkEm supports a dual-queue model that enables side-by-side comparison with and without GNAT over the same physical network. With this enhancement, whenever throttling is used, LinkEm manages any packets with ECN bits set to a designated "bypass" value separately using a separate credit accounting system and "front end" queue. Hence as long as the physical interface is much faster than the throttled rate, LinkEm can concurrently rate limit two independent streams. The assigned front-end buffer size, link speed (i.e., throttle value), delay, loss model and loss rate assigned either at startup or on-the-fly using LinkEmClient apply to both the ECN-marked and the non-ECN-marked streams separately.

12.1.4 Modeling Multiple Independent Paths

For testing and demonstration environments where we are unable to insert LinkEm inline between routers within the interior of a WAN, for example when testing over operational networks, impairments must instead be applied at the edges in a way that allows modeling impairments that would otherwise occur within the network interior. To support this model of operation, LinkEm has been extended so that impairments along paths within the interior of a black core network can be emulated with LinkEm instances located at the edges of the network, while retaining its original ability to emulate impairments by locating LinkEm within the network.

In order to emulate path impairments for multiple different paths using an impairment device located at the edge of a network, LinkEm has been enhanced to 1) allow specifying multiple different (emulated) Paths and 2) assign and apply a different model to each (emulated) Path. Up to 15 user configured Paths can be added to LinkEm, each Path associated with a collection of up to 8 IP subnets.

An example configuration for a 4 enclave experiment is shown in Figure 32. Here, the LinkEm instance running on iron-le1 is configured with three different Paths, one Path to each destination enclave: Path1 will service all packets destined to/received from enclave 2; Path 2 will service all packets destined to/received from enclave 3; and Path 3 will service all packets destined to/received from enclave 4. Each Path is configured with the remote enclave's IRON node subnet and application node subnet. The depicted configuration supports an IRON versus Baseline side-by-side comparison. The IRON traffic is destined to the 192.168.35.0, 192.168.36.0, and 192.168.33.0 subnets and the Baseline (bypass) traffic is destined to the 192.168.103.0, 192.168.104.0, and 192.168.101.0 subnets. The specification of two subnets for each of the Paths ensures identical impairments will be applied to the IRON and Baseline traffic when a Path's model is modified.

Similar Paths exist for each of the other LinkEm nodes, iron-le2, iron-le3, and iron-le4. For simplicity, we don't show the other LinkEm node's Paths.

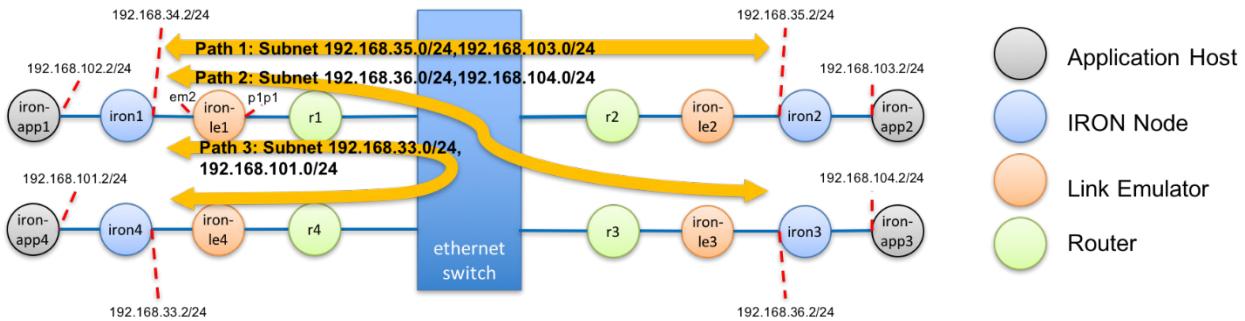


Figure 32. LinkEm Edge Impairment Example

The following items are configurable for each LinkEm Path:

- *Subnet specification* – up to 8 subnet address and prefix length pairs. The subnet specifications are used to find a matching Path when packets are received on one of the LinkEm interfaces. At start-up, LinkEm is provided the two interfaces that are to be bridged via the *-1 and -2* command-line arguments. Packets received on interface 1 are treated as IN-BOUND packets by LinkEm and packets received on interface 2 are treated as OUT-BOUND packets by LinkEm: in other words interface 1 (designated above as interface p1p1 for iron-le1) is assumed to be facing the WAN side of an enclave configuration, and interface 2 (designated above as interface em2 for iron-le1) is assumed to be facing the LAN side (the side closest to the IRON node) of an enclave. The assignment of an interface to the *-1* option and the *-2* option is dynamically determined when LinkEm is launched utilizing information provided in the experiment's *exp.cfg* file.

When an INBOUND packet from the WAN is received by LinkEm, the *source* address from the packet is used to lookup a matching Path. When an OUTBOUND packet from the LAN is received by LinkEm, the *destination* address from the packet is used to lookup a matching Path.

The subnet values for a Path should be the subnets that are reachable via interface 1. In Figure 32, the subnets for the Paths are the subnets that are reachable via the iron-le1 p1p1 interface.

- *Throttle* – channel rate in Kbps
- *Delay* – one-way propagation delay in ms
- *Buffer* – buffer size, in bytes
- *Model* – either SPER (simple packet error rate model), SBER (simple bit error rate model), or None (no model)
- *Model specific parameters* – Model parameters include bit error rates for the SBER model, packet error rates for the SPER model

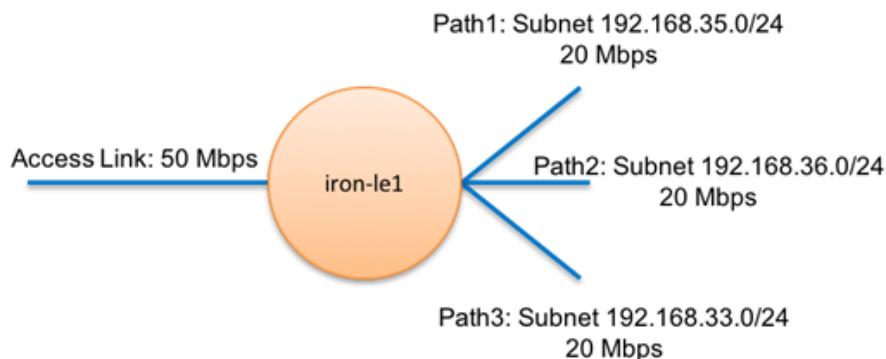
Once a Path is configured, it controls the flow of packets between the interfaces that are bridged together. Since each Path has its own configuration, different impairments can be applied to each Path independently.

12.1.4.1 Default Path

Path 0 is the default Path in LinkEm and is the match for 1) all non-IP packets and 2) all packets that do not match any other configured LinkEm Paths. The Path 0 subnet is not configurable and has a value of 0.0.0.0/0, which matches everything. The existence of this default Path enables **all** existing experiments to utilize the enhanced LinkEm without modification.

12.1.5 Modeling Access Links

The enhancements to LinkEm described above introduced the concept of modeling different paths in a single LinkEm instance. The modifications, however, did not support for access link rate limitations and Path rate limitations at the same time. Consider, as an example, a LinkEm supporting an enclave that is configured with Paths to 3 other enclaves, depicted below. The link modeling for packets destined to the 192.168.35.0 subnet is controlled by the LinkEm Path 1 configuration (rate limit of 20 Mbps), the modeling for packets destined to the 192.168.36.0 subnet is controlled by the LinkEm Path 2 configuration (rate limit of 20 Mbps), and the modeling for packets destined to the 192.168.33.0 subnet is controlled by the LinkEm Path 3 configuration (rate limit of 20 Mbps). If the Path configuration is symmetric for the links, it is possible that we allow 60 Mbps to enter the local enclave, which is more than the desired 50 Mbps.



To accurately model the above example, LinkEm has been modified to take into account both access link rate limitations and Path rate limitations at the same time. The serialization delay of a received packet is a function of both the access link rate limit and the packet's corresponding Path rate limit.

12.1.6 Configuring LinkEm

LinkEm can be configured by: 1) providing a configuration file as a command line argument or 2) using the LinkEmClient. The LinkEmClient is also used to modify the behavior of LinkEm post start-up. Internally, LinkEm has been modified so that the commands received from the LinkEmClient are processed the same as the commands that are read from a configuration file. The simplification of the internal LinkEm logic makes LinkEm more maintainable. For GNAT, LinkEm is typically configured with the LinkEmClient using the information that is provided an experiment's *lem_init.cfg* and *lem.cfg* files.

12.1.6.1 LinkEm Configuration File

Following is an example LinkEm configuration file. It provides a description of the supported LinkEm commands and their syntax. Additionally, it provides some example commands.

```
# An example LinkEm configuration file. This file may be provided to
# LinkEm at startup and the included "commands" will be used to
# initialize LinkEm. Additionally, the LinkEmClient will generate
# "commands" of the format in this file to remotely modify the
# behavior of LinkEm.
#
# # Supported "commands":
#
# - Pathx.y:<path parameters>
#
# Provides the LinkEm with a supported path on a bridged
# interface. 'x' represents the path identifier and must be
# between 1 and 15. 'y' represents the interface the path applies
# to and can be 0, 1, or 2. A value of 1 indicates that the Path
# command is applied to the LinkEm command-line '-1' interface option.
# A value of 2 indicates that the Path command is applied to the LinkEm
# command-line '-2' interface option. By convention, interface 1 is the
# WAN side of the LinkEm node and interface 2 is the LAN side. A value
# of 0 indicates that the Path command is applied to both the LinkEm
# command line '-1' and '-2' interface options.
#
# The <path parameters> can include the following:
#
# o s=address/prefix,... : This is the subnet specification for the
# Path. This must be provided with a 'y'
# value of 0, which indicates it applies
# to both interfaces. Up to 8 subnets can be
# provided for each Path. Packets are matched
# by either using the source or destination
# address from the received packets.
# Default value: 0.0.0.0/0,
# which will match all packets.
# o E=model : The error model type to use. Either SPER (Simple
# Packet Error Rate model), SBER (Simple Bit
# Error Rate model), or NONE (no model).
# Default value: SPER
# o e=name=value : Specify an error model parameter as a name
# value pair.
# o J=model : The Jitter model type to use.
# Default value: None
# o j=name=value : Specify a Jitter model parameter as a name
# value pair.
# o t=throttle (in Kbps) : The rate, in Kpbs.
# Default value: no throttling
# o d=delay (in ms) : The delay, in milliseconds.
# Default value: 0
# o b=buffer : The size of the buffer, in bytes.
# Default value: 65536.
# o ber=bit error rate : The bit error rate. This is a model
# specific parameter that only applies
# to the SBER Model.
# o per=packet error rate : The packet error rate. This is a model
# specific parameter that only applies
# to the SPER Model.
#
# Each Path command can include more than one path parameter. If
```

```
#      multiple are provided, they must be separated by the ';' character. Some examples follow:  
#  
#      Subnet address setup:  
#          Path1.0:s=172.24.2.0/24,172.24.3.0/24  
#  
#      Set throttle and delay symmetrically (50 Mbps, 20 ms):  
#          Path1.0:t=50000;d=20  
#  
#      Set delay asymmetrically (10 ms interface 2->1):  
#          Path1.2:d=10  
#  
# - Bypass=<TOS value>  
#  
#      Specifies the bypass TOS value used to support dual baseline link.  
#      A value of 0 disables bypass processing. Default value is 0.  
#  
# - Query  
#  
#      Queries the LinkEm for the state of the bridged interfaces. This command only makes sense when used from the LinkEmClient.  
  
# Specification for Path 1 with symmetric behavior.  
# Path1.0:s=172.24.1.0/24;m=1;t=1000;d=100;b=65536  
  
# Specification for Path 2 interface 1. An example that changes one subnet parameter at a time.  
# Path2.0:s=172.24.2.0/24  
# Path2.1:m=1  
# Path2.1:t=1000  
# Path2.1:d=100  
# Path2.1:b=128000  
  
# Specification for Path 2 interface 2.  
# Path2.2:m=1;d=1000  
  
# Specification for Path 8.  
# Path8.0:s=172.24.8.0/24;m=1;d=10  
  
# Specification for path 8 interface 4. This will generate an error in LinkEm.  
# Path8.4:m=1;d=10  
  
# Specification for path 20. This will generate an error in LinkEm.  
# Path20.0:s=172.24.2.0/24;m=1;d=10  
  
# Set the bypass TOS value.  
# Bypass=4
```

12.1.6.2 Controlling LinkEm with LinkEmClient

The LinkEmClient is used to modify the configuration of LinkEm and to query LinkEm for its internal state. Following is the usage message output from the LinkEmClient.

```
Usage: LinkEmClient [OPTION...]  
-h=<host>                                LinkEm host.  
-p=<port, default=3456>                      LinkEm management listen port.  
-w                                              TOS bypass value. 0 disables bypass.  
-q                                              Query the LinkEm state.  
-S                                              Query the operation status of the LinkEm.
```

<code>-R=<interval, in milliseconds></code>	Periodic statistics logging interval, in milliseconds. 0 disables periodic logging.
<code>-A</code>	Access Link modification.
<code>-P=<path></code>	Identifier of the Path to which the command applies. This must be between 1 and 15. This is required to modify the behavior of one of the LinkEm Paths.
<code>-I=<interface, default=0></code>	Identifier of the interface to which the command applies. A value of 0 indicates the command applies to both interfaces for the specified Path Identifier.
<code>-s=<ipaddress/prefix length,...></code>	Path subnet specifications. Up to 8 subnets specifications can be provided for each Path. If more than 1 subnet specification is provided, they must be separated by commas.
<code>-d=<delay></code>	Propagation delay, in ms.
<code>-t=<throttle></code>	Throttle value, in Kbps.
<code>-E=<error model name></code>	The error model name. One of SPER, SBER, or None.
<code>-e=<key>=<val> <type></code>	An error model specific parameter.
<code>-J=<jitter model name></code>	The jitter model name. One of GMM, DMM, or None.
<code>-j=<key>=<val></code>	A jitter model specific parameter.
<code>-b</code>	Buffer size, in bytes.
Help options:	
<code>-?, --help</code>	Show this help message
<code>--usage</code>	Display brief usage message

The `-h` option is optional. It defaults to localhost if not provided. The `-p` option is used if LinkEm is started with a management port that is different from the default management port. The `-w`, and `-q` options are **not** Path specific and are provided without the `-P` option. The `-d`, `-t`, `-E`, `-e`, `-J`, `-j`, and `-b` options **require** the `-P` option and may optionally include the `-I` option.

To set up a symmetric one-way delay of 50 ms for Path 1 in LinkEm executing on the local host, issue the following command:

LinkEmClient -P 1 -d 50

To set up an asymmetric delay, issue the following command next:

LinkEmClient -P 1 -I 2 -d 100

The combination of the above 2 commands will result in a 50 ms delay in the 1→2 direction and a 100 ms delay in the 2→1 direction, for a round-trip delay of 150 ms.

LinkEmClient may also be utilized to query the internal LinkEm state by issuing the following command:

LinkEmClient -q

which results in the following output:

```
iron@iron96:~$ LinkEmClient -h localhost -q
Path0.1:s=0.0.0.0/0;m=SPER Model;PER=0.0;t=0.000000;d=0;b=12800
Path0.2:s=0.0.0.0/0;m=SPER Model;PER=0.0;t=0.000000;d=0;b=12800
```

12.1.6.3 Configuring Access Links Using LinkEmClient

Access Link throttling is controlled via the LinkEmClient with the following command:

```
LinkEmClient -A [-I interface] -t throttle
```

The **-A** option is used to identify that the Access Link is being configured. The optional **-I** option is used to identify the Access Link interface. The semantics of using this option are the same as those used to configure the LinkEm Paths. Typically, this won't be provided, which means the throttling is symmetric for both interfaces. However, LinkEm does support different Access Link throttles for each of the interfaces. An asymmetric Access Link configuration could be employed to model different uplink and downlink rates, for example. The **-t** option is used to provide the Access Link throttle value, in Kbps.

The output from the LinkEmClient query command has been modified to include the newly added Access Link information as seen below:

```
iron@iron96:~$ LinkEmClient -q
AccessLink.1:t=50000.000000
AccessLink.2:t=50000.000000
Path0.1:s=0.0.0.0/0;E=SPER;PER=0.000000;J=None;t=0.000000;d=0;b=12800
Path0.2:s=0.0.0.0/0;E=SPER;PER=0.000000;J=None;t=0.000000;d=0;b=12800
Path1.1:s=192.168.35.0/24;E=None;J=None;t=0.000000;d=0;b=12800
Path1.2:s=192.168.35.0/24;E=SPER;J=None;t=20000.000000;d=0;b=12800
Path2.1:s=192.168.36.0/24;E=None;J=None;t=0.000000;d=0;b=12800
Path2.2:s=192.168.36.0/24;E=SPER;J=None;t=20000.000000;d=0;b=12800
Path3.1:s=192.168.33.0/24;E=None;J=None;t=0.000000;d=0;b=12800
Path3.2:s=192.168.33.0/24;E=SPER;J=None;t=20000.000000;d=0;b=12800
```

Lines 2 and 3 above show the Access Link throttling configuration, 50 Mbps, and lines 7, 9, and 11, show that the throttling configuration for the 3 Paths, each 20 Mbps. This LinkEm configuration aligns with the example depicted above.

12.1.7 Starting LinkEm

The format of the *LINKEM_NODES* entry in the GNAT experiment configuration file, *exp.cfg*, has been changed to the following:

```
LINKEM_NODES=(node6:nodeA:linkB node7:nodeC:linkD node8:nodeE:linkF)
```

The new information, **nodeA:linkB**, **nodeC:linkD**, and **nodeE:linkF**, identifies an interface on node6, node7, and node8, respectively. This information is resolved, during an experiment's configuration stage, to *Reference IP Addresses* using the information contained in the testbed topology file. Each *Reference IP Address* is provided to the IRON LinkEm startup script and is used

to dynamically determine which interface on the LinkEm node is the -1 command-line option and which interface is the -2 command-line option. Recall that the -1 and -2 command-line options identify the interfaces that are to be bridged by the LinkEm. The *Reference IP Address* allows for a generic LinkEm startup script to be used on all LinkEm nodes, irrespective of the IP Addresses assigned to the interfaces that are to be bridged together.

12.1.8 Choosing LinkEm Buffer Sizes

Buffer sizes can affect the behavior and performance of the various different algorithms used by the Capacity Adaptive Tunnels (CATs) for hop-by-hop congestion control.

The current default congestion control algorithm used by the SliqCats is **CUBIC,Copa**, meaning that CUBIC and Copa are working in tandem as a "Nuisance".

Other options including CUBIC by itself, Copa by itself, and several older versions of Copa including DetCopaM, DetCopa with a fixed delta value, and Copa2. Configurations for these variants are discussed below.

12.1.8.1 CUBIC,Copa (Default SliqCat Algorithm)

If GNAT is the only source of traffic on the testbed, a good buffer size to use is 25Kbytes. In general this is too small for CUBIC (and so will incur losses and back off to rates lower than the nominal link capacity), but is more than sufficient for Copa and the two algorithms together will keep the link full, irrespective of link speed.

When GNAT is **not** the only source of traffic on the testbed network, and competing traffic is running CUBIC, then the buffer size should be increased, using the rule for CUBIC described below

12.1.8.2 CUBIC

CUBIC uses packet losses as a sign of congestion, so it is able to adapt to the amount of buffer space at the bottleneck link. When using CUBIC, the LinkEm buffer sizes should generally be set to between 1 to 2 times the bandwidth-delay product for optimal behavior. This follows general best-practices for TCP, which is where CUBIC originated. However, CUBIC will work with other buffer sizes, although its performance may not be optimal in these cases. The specific calculation is shown below:

- The RTT for a link is twice the specified LinkEm delay; If the LinkEm delay is specified as 0, then assume an RTT of 1 millisecond.
- Set the buffer size, in bytes, to two times the bandwidth-delay product for the link in units of bytes. This is equal to $(2 * \text{rate} * \text{rtt} / 8)$ where rate is in units of bits per second and rtt is in units of seconds.
- If the computed buffer size is less than 20,000 bytes, then use a buffer size of 20,000 bytes

Note that CUBIC will work reasonably well with smaller buffers (e.g., 15kBytes) but experimentally at least does not make full use of the channel.

12.1.8.3 Copa

Copa3 tries to keep an average of around 4 packets in the bottleneck queue, oscillating between zero packets (briefly!) and about twice this value. Assuming 1500 byte packets yields a peak value of 12Kbytes; to provide extra headroom, we round this number up to 20Kbytes.

12.1.8.4 Older Copa variants

12.1.8.4.1 DetCopaM

The policy controller used in DetCopaM adjusts the Copa delta value dynamically, and Copa attempts to keep exactly 1/delta packets enqueued at the bottleneck link, where each packet is assumed to be 1,000 bytes. The current policy controller used in this algorithm is not able to dynamically detect the amount of buffer space at the bottleneck link. Thus, when using this algorithm, it is very important to never under-buffer the LinkEm nodes. If a LinkEm node is under-buffered, then the DetCopaM send rate over that link will likely go to infinity since it does not use losses as a congestion signal. The general rules for setting the LinkEm buffer size are then:

- The RTT for a link is twice the specified LinkEm delay; If the LinkEm delay is specified as 0, then assume an RTT of 1 millisecond.
- Set the buffer size, in bytes, to two times the bandwidth-delay product for the link. This is equal to $(2 * \text{rate} * \text{rtt} / 8)$ in order to get the buffer size in bytes, where rate is in bits per second and rtt is in seconds.
- If the computed buffer size is less than 20,000 bytes, then use a buffer size of 20,000 bytes. This is because Copa limits the delta value to 0.1 (the maximum value), which means that the minimum number of packets that Copa attempts to keep enqueued at the bottleneck link is 10 packets, where each packet is assumed to be 1,000 bytes long, or 10,000 bytes total. This is then doubled in order to allow Copa some headroom for send rate oscillations.
- There is no need to set the buffer size to more than 500,000 bytes unless you are purposefully experimenting with buffer-bloat. This is because Copa limits the delta value to 0.004 (the minimum value), which means that the maximum number of packets that Copa attempts to keep enqueued at the bottleneck link is 250 packets, where each packet is assumed to be 1,000 bytes long, or 250,000 bytes total. This is then doubled in order to allow Copa some headroom for send rate oscillations.

For example, the buffer size for a LinkEm link with 10,000 kbps throttling and 10 milliseconds of delay would be:

$$(1) \text{size} = 2 * 10,000,000 * (2 * 0.010) / 8 = 50,000$$

12.1.8.4.2 DetCopa With Fixed Delta Value

When using DetCopa with a fixed delta value (e.g., DetCopa_0.04, where the delta value is equal to 0.04), Copa will attempt to keep exactly 1/delta packets enqueued at the bottleneck link, where each packet is assumed to be 1,000 bytes. With this algorithm, it is very important to never under-buffer the LinkEm nodes. If a LinkEm node is under-buffered, then the DetCopa send rate over that link will likely go to infinity since it does not use losses as a congestion signal. The general rules for setting the LinkEm buffer size are:

- Set the buffer size to **(2,000 / delta)** bytes. The factor of 2 is included to allow for some headroom for send rate oscillations,

12.1.8.4.3 Copa2

The Copa2 congestion control algorithm is also an early version of Copa, and is considered deprecated. Users should use any of the other algorithms listed above.

However, since it is very similar to DetCopa and uses a delta value, the rules given above for DetCopa With Fixed Delta Value apply to this algorithm as well. The default delta value for Copa2 is currently 0.1, although it may be configured with other delta values as needed.

12.2 NACK-Oriented Reliable Multicast (NORM)-based File Transfer Protocol (*nftp*)

12.2.1 Summary of Goals and Objectives

The main goals of *nftp* are:

- Providing a reliable file transfer capability to a group of receivers, using the [NORM](#) protocol as a basis.
- Providing a reliable multicast file transfer capability that controls multicast group membership at the source. This is a deviation from traditional multicast which is receiver driven.
- Providing a single multicast group for control plane and data plane traffic. This eliminates 1) multicast route convergence times and 2) having to choose a network-wide available multicast group for new transfers.
- Providing an abstract interface to coordinate with the underlying network. This provides the flexibility to develop different "flavors" of *nftp* that are able to work with different types of underlying networks. For example, a GNAT-provided implementation of this interface provides the destination list for a file transfer to AMP, which in turn directs the GNAT UDP Proxy how to mark the flow's packets with the appropriate destination list.
- Providing an interface that separates control plane traffic from data plane traffic, similar to how *ftp* operates.
- Providing an interface that is similar to the well-known *scp* Linux utility. A list of destination address and output paths are provided for each destination, allowing for the creation of different files in different locations at each destination, if desired.

12.2.2 High Level Design

The *nftp* utility is composed of the following two components:

- *nftpd*: Daemon that runs on all nodes in the network that may be file transfer participants. *nftpd* listens for control packets transmitted by *nftp* sources advertising a file transfer, acknowledges the file transfer advertisement packet if the local node is in the destination list for the transfer, and forks off an *nftp* receiver for the advertised file transfer.
- *nftp*: Acts as either a source or receiver for a multicast file transfer. *nftp* sources multicast control packets containing information about the upcoming file transfer, including a list of destinations and information used by the source to uniquely identify the flow. *nftp* sources wait until the identified receivers have acknowledged receipt of the control packet, indicating that the *nftp* receivers are ready to receive the file, before starting the file transfer. *nftp* receivers receive the file transfer packets and write the file to the appropriate location on the local disk.

Figure 33, depicted below, illustrates the operational overview of utilizing *nftp* to transfer a file from Node1 to Node2 and Node4. Note: *nftpd* is running on all network nodes prior to the start of the desired file transfer.

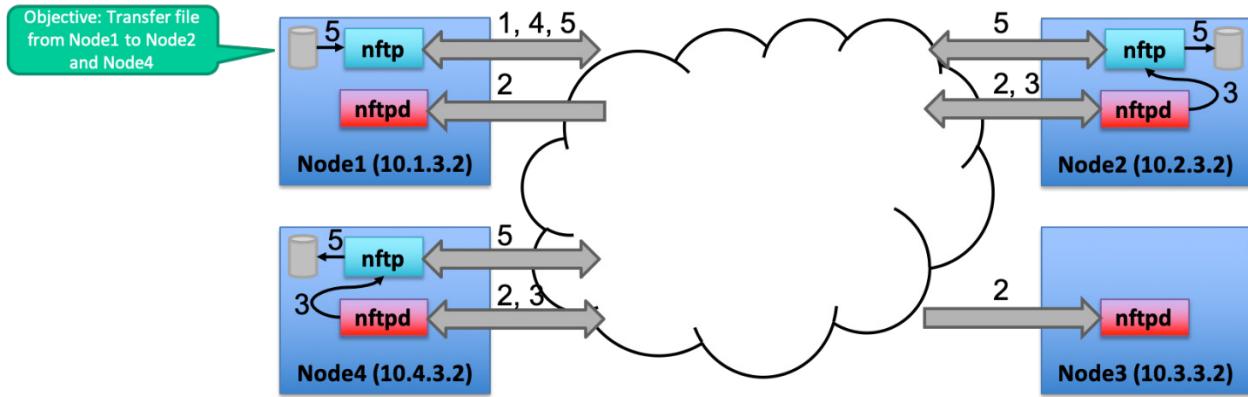


Figure 33. Using nftp to transfer a file from Node1 to Node2 and Node4

The following sequence of events occur during the file transfer process. The identifier for each of the events is shown in Figure 33.

1. An *nftp* source starts one Node1, which sends a File Transfer Announcement (FTA) packet to the multicast group that all *nftpd* instances have joined. The FTA identifies Node2 and Node4 as destinations.
2. All *nftpd* instances receive the *nftp* FTA packet.
3. The *nftpd* instances running on Node2 and Node4 generate and transmit a File Transfer Announcement Acknowledgement (FTAA) packet and fork off an *nftp* receiver instance. Node1 and Node3 do not acknowledge the received FTA control message as they are not in the file transfer destination list.
4. The *nftp* source on Node1 receives the FTAA packets.
5. When all FTAA packets are received by the *nftp* source on Node 1, it reads the file from disk and starts the file transfer. Node2 and Node4 receive the file and write it to disk, transmitting the appropriate NORM feedback packets to Node1, as necessary.
6. Once the file transfer has completed, the *nftp* source on Node1 and the *nftp* receivers on Node2 and Node4 terminate.

As illustrated by the above sequence of events for a file transfer, *nftp* has a control plane that is separate from the data plane, similar to how *ftp* operates. The control plane traffic occurs between an *nftp* source and the *nftpd* instances and consists of FTA packets sent by the *nftp* source and FTAA packets sent by *nftpd* instances that are in the list of destinations for the file transfer. The data plane traffic occurs between the *nftp* source and the set of *nftp* receivers that are started to support the file transfer and consists of the file transfer packets and the NORM feedback packets that drive the repair process.

12.2.3 Detailed Design

12.2.3.1 *nftp* Control Plane Packets

File Transfer Announcement (FTA) Packets

When an *nftp* source has a file to transfer, it first generates and transmits a File Transfer Advertisement (FTA) packet that contains the source address and source port for the file transfer and a

list of destination information, including the destination IP Address and the destination output path for the received file.

The format of the FTA packet is shown below.

File Transfer Announcement (FTA) Packet Format

The FTA packet fields are:

- **Msg Len** (2 bytes) - The total length of the FTA packet, in network byte order
 - **Msg Type** (1 byte) - Message type, set to 0x1
 - **Unused** (1 byte) - Unused, set to zero
 - **Src IP Address** (4 bytes) - The source IP Address for the file transfer, in network byte order
 - **Src Port** (2 bytes) - The source port for the file transfer, in network byte order
 - **Num Dsts** (1 byte) - The number of destinations for the file transfer
 - **Unused** (1 byte) - Unused, set to zero

For each destination parameter set included, the required fields are:

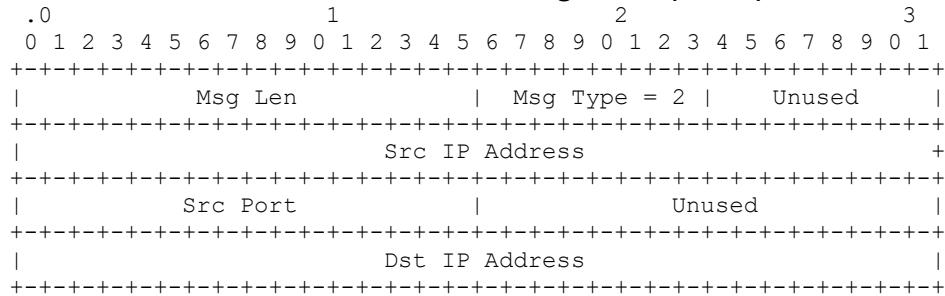
- **Dst IP Address** (4 bytes) - The destination IP Address for the file transfer, in network byte order
 - **Dst Path Len** (1 byte) - The length of the destination output path
 - **Dst Output Path** (Dst Path Len bytes) - The output path for the received file

12.2.3.2 File Transfer Announcement Acknowledgement (FTAA) Packets

When *nftpd* receives an FTA packet that is for the local node, it generates and transmits an FTAA packet that contains the source address, source port, and destination address.

The format of the FTAA packet is shown below.

File Transfer Announcement Acknowledgement (FTAA) Packet Format



The FTAA packet fields are:

- **Msg Len** (2 bytes) - The total length of the FTAA packet, in network byte order
- **Msg Type** (1 byte) - Message type, set to 0x2
- **Unused** (1 byte) - Unused, set to zero
- **Src Ip Address** (4 bytes) - The source IP Address for the file transfer, in network byte order
- **Src Port** (2 bytes) - The source port for the file transfer, in network byte order
- **Dst IP Address** (4 bytes) - The destination address that matches the local node, in network byte order

12.2.3.3 *nftp* Data Plane Packets

Refer to the NORM [Developer Guide](#) and [RFC](#) for the details pertaining to the file transfer and repair feedback packet formats.

12.2.4 *nftp* Receiver Packet Filtering

As described previously, control plane and data plane traffic is sent to the single multicast group used by *nftp*. The *nftp* receivers that are started to support a file transfer must have a way to filter out unwanted packets. This is accomplished by the filtering that is established at the receivers. Recall that *nftp* receivers are started by *nftpd* when *nftpd* determines that the local node is a destination for an announced file transfer. The received FTA packet contains the information necessary to set up the *nftp* receiver filtering.

The Source IP Address and Source Port contained in the FTA packet is the information that uniquely identifies a file transfer. This information is provided to the *nftp* receiver when it is started. The receiver takes the following actions to ensure that it receives only packets for the file transfer for which it was started.

1. The *nftp* receiver joins the multicast group specifying the source address from which to accept data, known as [Source Specific Multicast](#) (SSM). This portion of the receive filter ensures that the *nftp* receiver only receives packets from the file transfer source address.
2. The *nftp* receiver attaches a Berkeley Packet Filter (BPF) to the receive socket that identifies the source port that must be matched to receive packets, referred to as Source Specific Port (SSP) filtering.

The filter combination of SSM and SSP ensure that the *nftp* receiver only receives packets for the announced file transfer, even though there may be other file transfers to the single multicast group utilized by the *nftpd* and *nftp* applications.

12.2.5 NORM Enhancements

A number of enhancements have been made to the publicly available NORM library in direct support of *nftp*, as described below.

12.2.5.1 Addition of Source Specific Port (SSP) Filtering to NORM

12.2.5.1.1 SSP Filtering Goal

This feature makes it possible to use a single multicast group for all file transfers in a GNAT network.

12.2.5.1.2 SSP Filtering Approach

An API call was added to NORM to support the identification of a source port filter, *nftp*'s SSP filtering that enables an *nftp* receiver to only receive packets for the file transfer that it is participating in. This call dynamically generates an appropriate BPF and attaches the BPF to the *nftp* receive socket.

12.2.5.1.3 SSP Filtering Impact

SSP Filtering enables *nftp* to deconflict the packets for concurrent *nftp* file transfers originating from a common source node. It makes it possible for *nftp* to use a single multicast group in GNAT experiments for all file transfers.

12.2.5.2 Addition of Window-Based Flow Control to NORM

12.2.5.2.1 Window-Based Flow Control Goal

A window-based flow control feature was added to NORM to leverage GNAT-assisted flow control. This significantly improved throughput in GNAT's multi-path environment, and in particular allows coordination between NORM and GNAT's Admission Control functions

12.2.5.2.2 Window-Based Flow Control Approach

The approach uses existing NORM sequence numbers, which are included in all NORM transmissions. Each time that a packet is transmitted at the source, the highest sent sequence number

is updated. The source receives feedback from the GNAT network, which it uses to compute the size of the flow control window.

Two types of feedback packets are provided by the GNAT network, described below.

Flow Control Window Size Update Packet Format

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+-----+			
version type=7	hdr_len	sequence	
+-----+			
	source_id		
+-----+			
subtype=1	unused	window_size	
+-----+			

The Flow Control Window Size Update Packet fields are:

- **version** (4 bits) - Set to 0x1
- **type** (4 bits) - Set to 0x7
- **hdr_len** (1 byte) - Indicates the number of 32-bit words that comprise the message's header portion, set to 0x2
- **sequence** (2 bytes) - Sequence number set by the message originator, in network byte order
- **source_id** (4 bytes) - Uniquely identifies the node that sent the message within the context of a single NORM Session, in network byte order
- **subtype** (1 byte) - Flow Control message subtype, set to 0x1
- **unused** (1 byte) - Unused, set to zero
- **window_size** (2 bytes) - The size of the flow control window, in packets, in network byte order

Flow Control Window Update Packet Format

.0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+-----+			
version type=7	hdr_len	sequence	
+-----+			
	source_id		
+-----+			
subtype=2	unused	window_size	
+-----+			
rcv_seq_num	sent_seq_num		
+-----+			

The Flow Control Window Update Packet fields are:

- **version** (4 bits) - Set to 0x1
- **type** (4 bits) - Set to 0x7
- **hdr_len** (1 byte) - Indicates the number of 32-bit words that comprise the message's header portion, set to 0x2
- **sequence** (2 bytes) - Sequence number set by the message originator, in network byte order

- **source_id** (4 bytes) - Uniquely identifies the node that sent the message within the context of a single NORM Session, in network byte order
- **subtype** (1 byte) - Flow Control message subtype, set to 0x2
- **unused** (1 byte) - Unused, set to zero
- **window_size** (2 bytes) - The size of the flow control window, in packets, in network byte order
- **rcv_seq_num** (2 bytes) - The sequence number of the last received packet from the NORM source, in network byte order
- **sent_seq_num** (2 bytes) - The sequence number of the packet most recently admitted to the GNAT network, in network byte order

The available flow control window size, X, is computed at the source by the following:

$$X = W - (S - A)$$

where,

- **X** is the available window size, in packets, computed at the source
- **W** is the window size, in packets, reported by GNAT in the Flow Control Window Size and Flow Control Window Update packets
- **S** is the highest sequence number sent by the *nftp* source
- **A** is the last sequence number admitted to the network by GNAT, reported in the Flow Control Window Update packets

The main event logic inside of NORM asks the NORM Flow Controller, if enabled, for the computed available window size for the NORM Session. If there is available room in the computed window, the pending packet is transmitted. Otherwise, no transmission occurs.

Currently, this feature is only useful if using *nftp* in a GNAT experiment, where flow control window information is provided by the network. Otherwise, the *nftp* source doesn't receive the required feedback pertaining to the flow control window leading to an *nftp* source that is eternally flow control blocked.

12.2.5.2.3 Window-Based Flow Control Impact

The addition of window-based flow control allows NORM enables to take full advantage of GNAT's performance and resilience. It enables for the development of a NORM-based file transfer application that can set the NORM TX Rate to a value that is greater than the network can provide. Once the network buffers fill up, the NORM source calculates that it is window limited and stops additional packet transfers until it receives updates "opening" the flow control window indicating packet transmission can resume. This approach is more efficient than trying to dynamically adjust the NORM TX Rate to match the network rate. Using this approach, packets are available when the network is ready until a time that the source has no additional packets to send.

12.2.6 Configurable Parameters

The following are common configurable *nftp* parameters:

- **Multicast Interface:** Controls the interface that is used to transmit/receive multicast packets, controlled by the '-i' option.
- **Multicast Address:** Identifies the single multicast group that is used for control plane and data plane traffic, controlled by the '-m' option. **Note:** this must match the multicast group that *nftpd* is using.
- **Multicast Destination Port:** Identifies the multicast destination port for control plane and data plane traffic, controlled by the 'p' option. **Note:** this must match the multicast destination port that *nftpd* is using.

The following are configurable *nftp* source parameters for a GNAT network:

- **AMP IP Address:** The IP Address of the GNAT AMP process, controlled by the '-A' option.
- **Window-Based Flow Control:** Enables window-based flow control, controlled by the '-f' option.
- **NORM TCP-Friendly Congestion Control:** Enables the NORM provided TCP-friendly Congestion Control, controlled by the '-c' option.
- **File Name:** Identifies the fully qualified name of the file to transfer, controlled by the '-S' option.

The following are configurable *nftp* destination parameters:

- **Source Specific Multicast IP Address:** The IP Address of the source of the file transfer, controlled by the '-a' option.
- **Output File Name:** Identifies the name of the output file, controlled by the '-o' option.
- **Receive Output Directory:** The fully qualified path to the destination directory, controlled by the '-R' option.
- **Source Port:** The source port, provided by the *nftp* source, for a file transfer, controlled by the '-s' option.

The following are configurable *nftpd* parameters:

- **Multicast Interface:** Controls the interface that is used to transmit/receive multicast packets, controlled by the '-i' option.
- **Multicast Address:** Identifies the single multicast group that is used for control plane traffic, controlled by the '-m' option. **Note:** this must match the multicast group that *nftp* is using.
- **Multicast Destination Port:** Identifies the multicast destination port for control plane traffic, controlled by the '-p' option. **Note:** this must match the multicast destination port that *nftp* is using.
- **nftp Binary Directory:** Identifies the location of the *nftp* binaries, controlled by the '-B' option.

For more detailed usage instructions, including default values for user-configurable parameters, execute one of the following commands:

nftp/nftpd Usage Commands

`nftpg -h`

`nftpd -h`

12.3 GNAT-enabled Secure Copy (scp) Application

We use the *scpa.sh* script to demonstrate how file transfer applications can interface with GNAT. This script wraps the *scp* command and allows the use to assign deadlines and priorities to file transfers, which will be configured in GNAT in real time.

12.3.1 Assumptions:

It is assumed that the underlying *scp* command will run without a prompt (i.e. keys should be in place, to allow ssh access without password or passphrase).

12.3.2 Usage:

The *scpa.sh* script can be found in IRON/experiments/scripts and should be executed on the source application node. It uses the following syntax:

```
scpa.sh [-12346BCpqrv] [-c cipher] [-F ssh_config] [-i identity_file] [-l limit] [-o ssh_option] [-P port] [-S program]"[[user@]host1:]file1 ... [[user@]host2:]file2 <deadline> <priority> <AMP_addr>
```

All but the last three parameters are simply *scp* syntax, and these parameters will be passed to *scp* in this order. Hence *scpa.sh* can support anything *scp* can support.

The <deadline> is a soft deadline for the file transfer, in seconds.

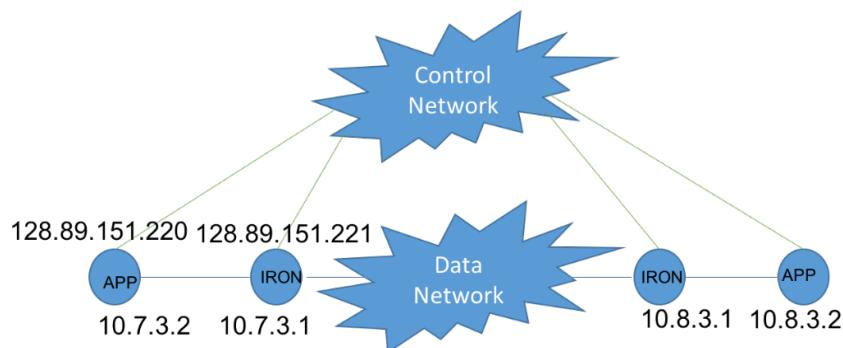
The <priority> is the priority to be assigned to the file transfer.

The <AMP_addr> is the control-plane address of IRON node attached to the source application.

It is important to note that the address of the destination should be on the data plane and not the control plane as we want the file transfer to flow through the IRON network.

The command below is an example command, with network diagram to show which IP addresses are used:

```
./scpa.sh -i ~/.ssh/id_rsa_iron ..../oop.pcap iron@10.8.3.2:~/a.pcap 90 10  
128.89.151.221
```



13 Creating and Running GNAT Experiments

13.1 Preface

To facilitate development and testing of GNAT, we created a set of automation scripts and tools that we use to configure and run experiments, as well as gather results from experiments and perform various analyses. This document describes:

- How we create a flexible and composable network emulation testbed
- What is needed to configure and run a GNAT experiment using the testbed
- The various scripts and tools supporting this process, and
- How these scripts and tools are used, including a simple example

Please note: GNAT is derived from an overlay-based backpressure forwarding capability referred to as IRON (for Intrinsically Resilient Overlay Network) developed by us under the earlier DARPA EdgeCT program. Since GNAT is largely an enhanced version of IRON, and to avoid confusing outside organizations already familiar with IRON, we retain the name "IRON" for the enhanced software capabilities developed under GNAT. Hence references to IRON directories and users appearing in the discussion below reflect our choice to retain the name IRON for our GNAT software.

13.2 Test Network Emulation

As described in the project overview and illustrated in Figure 34 below, GNAT functions as a overlay network that interconnects two or more user sites using a public network infrastructure as the underlay. The user sites (black clouds, representing LAN environments) are separated from the public network (gray cloud, representing WAN environments) using GNAT gateways (green squares) located at the edges of the public network. GNAT nodes can also be placed at strategic locations within the WAN to further improve performance. GNAT nodes use tunnels to steer traffic along alternate underlay paths when one or more of the direct underlay paths are impaired or otherwise unusable.

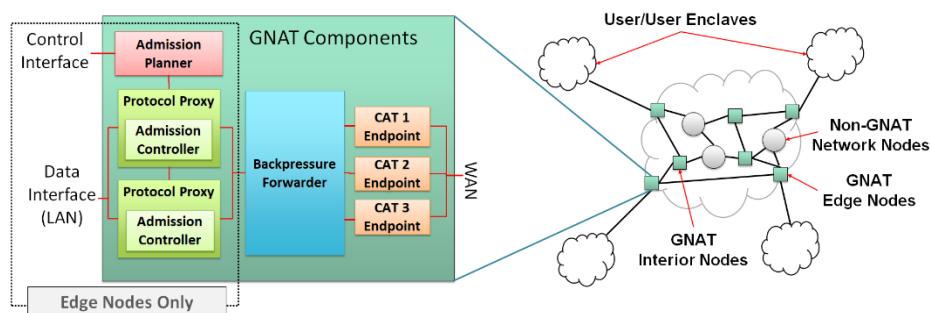


Figure 34. The network model for the DCOMP program consists of distributed user sites – enclaves – interconnected by a large public network (e.g., the Internet). In-line devices (shown here as GNAT nodes) situated between enclaves improve performance and resilience to congestion, misconfigurations and attacks within the public network.

Emulating this environment for development and testing purposes consists of three main networking components:

1. Sources and sinks of application traffic
2. GNAT nodes, which are functionally equivalent to routers
3. Network emulation to model base path characteristics and to dynamically inject or remove impairments along those paths.

In our testbed we organize a set of three hosts into an enclave emulation unit; interconnecting multiple enclave emulation units via an ethernet switch allows us to easily compose test networks with varying numbers of enclaves. As shown in Figure 35, each enclave emulation unit consists of:

1. An App Node running on the first of the three hosts that is used to source and sink all application traffic for the enclave.
2. A GNAT node running on the second of the three hosts that provides the DCOMP services. Each GNAT node is configured with two separate WAN-facing interfaces to support dual-homed operation as a means for providing further resilience. This allows a GNAT node to be connected to two distinct networks (e.g., Comcast and Fios).
3. A WAN emulation node running on the third of the three hosts. The WAN emulation host consists of two separate path emulators, one for modeling the network paths for each network, and a support router. The support router provides the means for routing traffic between different enclaves.

Note that each of the three hosts is connected to a common experiment control network that is used for loading software, starting and stopping experiments, and retrieving experimental artifacts. The control interfaces to each of the enclave hosts shown here are not shown in subsequent drawings for simplicity.

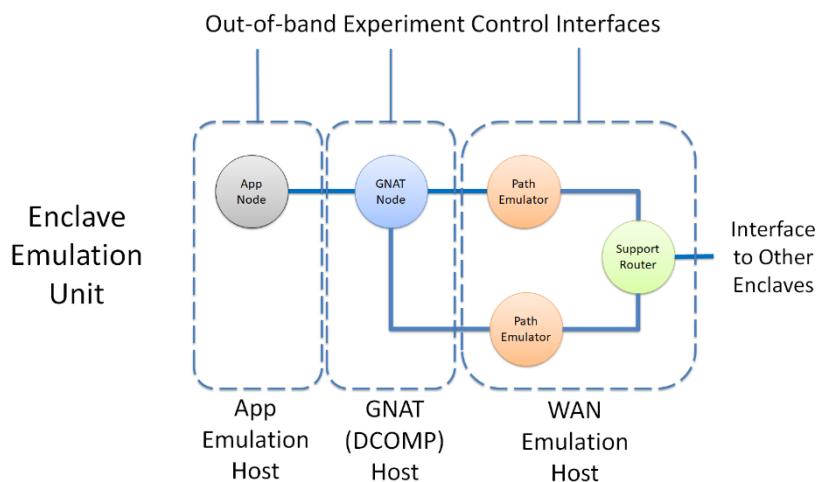


Figure 35. Our test network infrastructure is organized into enclave emulation units. Interconnecting two or more of these units via an ethernet switch allows creating test networks with varying numbers of enclaves as needed.

Note also that we use a separate physical interface for each of the connections shown in Figure 35. Hence the App Emulation Host requires two physical interfaces, the GNAT Host uses four physical interfaces, and the WAN emulation host uses eight physical interfaces.

This in turn requires that routing in each of the three enclave emulation hosts be configured so that the underlay network being modeled works correctly without any GNAT software or without any link/path emulation software running. Hence the app node in one enclave can communicate with the app node in any other enclave. Moreover, since the path emulators are functionally equivalent to ethernet bridges, the physical interfaces associated with each path emulator are typically bridged together when the path emulation software is not running.

Our experiment control scripts assign the four primary GNAT components (BPF, UDP Proxy, TCP Proxy, and AMP) to separate CPU cores whenever available. Each path emulator will be assigned to a separate core as well (our path emulators use busy wait loops to provide highly accurate timing). Hence it is recommended that physical hosts used as GNAT hosts and WAN emulation hosts each have at least four cores.

13.3 Testbeds

Multiple enclave emulation units are interconnected using a large Ethernet switch to create an experiment testbed, an example of which is shown in Figure 36 below:

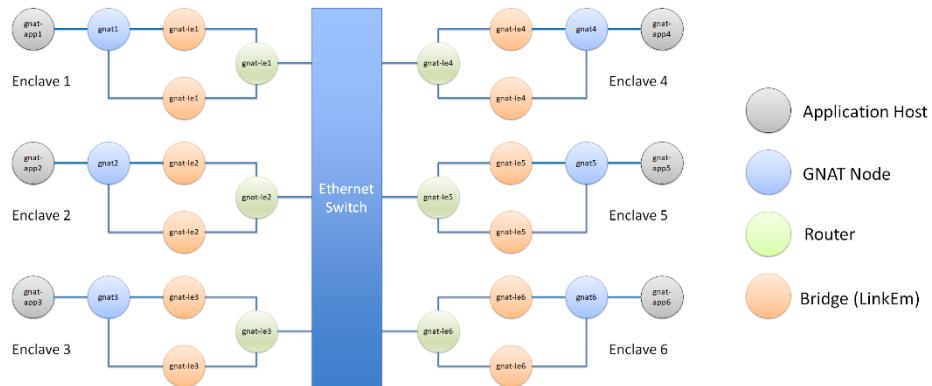


Figure 36. Enclave emulation units are hosted on real physical nodes. Enclaves are interconnected using a large Ethernet switch.

Organizing the testbed in this fashion enables different combinations of enclaves to support an experiment. For example, consider a 3 enclave experiment that is run multiple times. Run 1 might occur on Enclaves 1, 2, and 3 and subsequent runs might occur on Enclaves 1, 5, and 6, Enclaves 2, 3, and 4, and Enclaves 1, 3, and 5. The physical organization of the testbed permits the use of *any* 3 enclaves to support our hypothetical 3 enclave experiment. If an experimenter wishes to run on the same enclaves (for example, to do some regression testing on a fixed set of hardware), this can be accomplished. When an experiment is run, the experimenter can request either the explicit enclaves desired or the number of required enclaves. In either case, the real

physical enclaves reserved for the experiment are provided to the automated experiment execution scripts. Creating and running experiments will be described in the *Creating A New Experiment* and *Experiment Execution* sections below.

Note that while dual homing is supported by the above testbed, it need not be used for all experiments.

13.3.1 Configuring Testbed Nodes

Once organized into enclave units, our testbed nodes require some configuration before they are ready for hosting a GNAT experiment. The interfaces on the testbed nodes must be configured, **sudo** must be properly configured to work with the automated experiment execution scripts, and a minimum set of applications need to be installed on the application nodes.

Note: The GNAT software has been tested on Ubuntu 14.04, Ubuntu 16.04, and Ubuntu 18.04, and we fully expect GNAT to work correctly with other Linux distributions and versions. One caveat is that Linux kernel versions between 4.5 and 4.18 have a UDP port reuse bug that requires setting up separate ports for each CAT in the BPF configurations for these kernel releases. We were able to get the Linux networking maintainers to fix this particular bug, and kernel release 4.19 and later have been shown to work correctly.

13.3.1.1 Interface Configuration

As previously described, the routing in the testbed nodes is configured so that the underlay network being modeled works correctly without any IRON software or without any link/path emulation software running. Hence the application node in one enclave can communicate with the application nodes in all other enclaves. Figure 37 depicts our example physical testbed with all of the interface addresses identified. Note that there are many ways that the addressing could be assigned and that this only serves as an example of how we have configured our testbed.

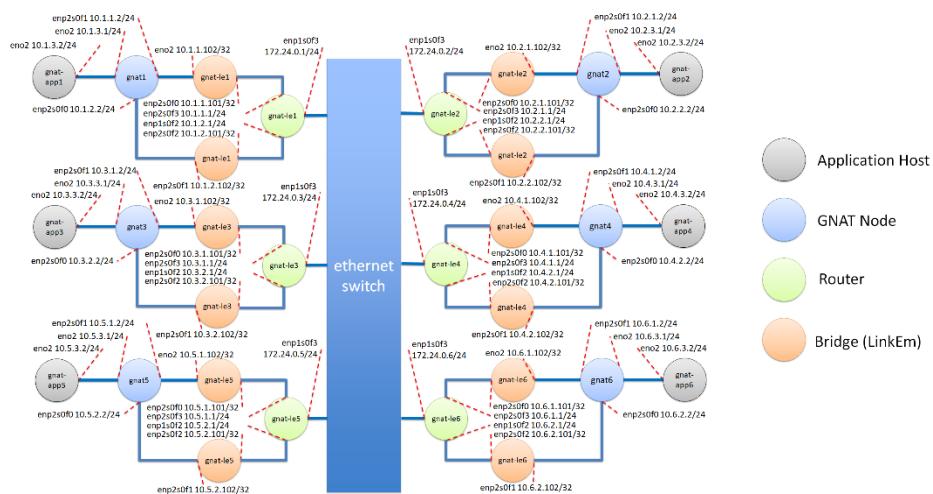


Figure 37. Physical testbed showing interface addresses

In addition to configuring the interface addresses, we set up the necessary static routes in the */etc/network/interfaces* file for each of the testbed nodes. The configuration for the App Emulation Host, **gnat-app1**, is depicted below:

gnat-app1 Interface File

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eno1
iface eno1 inet static
    address 128.89.151.170
    gateway 128.89.151.1
    netmask 255.255.254.0

# The application network interface
auto eno2
iface eno2 inet static
    address 10.1.3.2
    netmask 255.255.255.0
    mtu 1350
    up route add -net 10.0.0.0/8 gw 10.1.3.1 dev eno2
    up route add -net 172.24.0.0/16 gw 10.1.3.1 dev eno2
```

The configuration for the GNAT Host, **gnat1**, is depicted below:

gnat1 Interface File

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eno1
iface eno1 inet static
    address 128.89.151.201
    gateway 128.89.151.1
    netmask 255.255.254.0

auto eno2
iface eno2 inet static
    address 10.1.3.1
    netmask 255.255.255.0
#    up route add -net 10.1.3.0/24 gw 10.1.3.2 dev eno2

auto enp2s0f1
iface enp2s0f1 inet static
    address 10.1.1.2
    netmask 255.255.255.0
    up route add -net 10.0.0.0/8 gw 10.1.1.1 dev enp2s0f1
    up route add -net 172.24.0.0/24 gw 10.1.1.1 dev enp2s0f1

auto enp2s0f0
iface enp2s0f0 inet static
    address 10.1.2.2
    netmask 255.255.255.0
    post-up /sbin/ip route flush table 2
    post-up /sbin/ip route add default via 10.1.2.1 dev enp2s0f0 table 2
    post-up /sbin/ip rule add from 10.1.2.2 table 2
    post-down /sbin/ip rule del from 10.1.2.2 table 2
```

The configuration for the WAN Emulation Host, **gnat-le1**, is depicted below:

gnat-le1 Interface File

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eno1
iface eno1 inet static
    address 128.89.151.211
    gateway 128.89.151.1
    netmask 255.255.254.0

auto br_2s0f0_eno2
iface br_2s0f0_eno2 inet static
    address 10.115.11.1/32
    bridge_ports eno2 enp2s0f0
    post-up ip address add 10.1.1.101/32 dev enp2s0f0
    post-up ip address add 10.1.1.102/32 dev eno2

auto br_2s0f2_2s0f1
iface br_2s0f2_2s0f1 inet static
    address 10.115.12.1/32
    bridge_ports enp2s0f1 enp2s0f2
    post-up ip address add 10.1.2.101/32 dev enp2s0f2
    post-up ip address add 10.1.2.102/32 dev enp2s0f1

# The following are taken care of in the above bridge config post-up commands
#
#auto enp2s0f0
#iface enp2s0f0 inet static
#    address 10.1.1.101
#    netmask 255.255.255.255

#auto eno2
#iface eno2 inet static
#    address 10.1.1.102
#    netmask 255.255.255.255

#auto enp2s0f2
#iface enp2s0f2 inet static
#    address 10.1.2.101
#    netmask 255.255.255.255

#auto enp2s0f1
#iface enp2s0f1 inet static
#    address 10.1.2.102
#    netmask 255.255.255.255

auto enp2s0f3
iface enp2s0f3 inet static
    address 10.1.1.1
    netmask 255.255.255.0
#    up route add -net 10.1.1.0/24 gw 10.1.1.2 dev enp2s0f3
#    up route add -net 10.1.3.0/24 gw 10.1.1.2 dev enp2s0f3

auto enp1s0f2
iface enp1s0f2 inet static
    address 10.1.2.1
    netmask 255.255.255.0
```

```
#      up route add -net 10.1.2.0/24 gw 10.1.2.2 dev enp1s0f2

auto enp1s0f3
iface enp1s0f3 inet static
    address 172.24.0.1
    netmask 255.255.255.0
#      up route add -net 10.1.0.0/16 gw 172.24.0.1 dev enp1s0f3
#      up route add -net 10.2.0.0/16 gw 172.24.0.2 dev enp1s0f
#      up route add -net 10.3.0.0/16 gw 172.24.0.3 dev enp1s0f3
#      up route add -net 10.4.0.0/16 gw 172.24.0.4 dev enp1s0f3
#      up route add -net 10.5.0.0/16 gw 172.24.0.5 dev enp1s0f3
#      up route add -net 10.6.0.0/16 gw 172.24.0.6 dev enp1s0f3
#      up route add -net 10.7.0.0/16 gw 172.24.0.7 dev enp1s0f3
#      up route add -net 10.8.0.0/16 gw 172.24.0.8 dev enp1s0f3
#      up route add -net 10.9.0.0/16 gw 172.24.0.9 dev enp1s0f3
#      up route add -net 10.10.0.0/16 gw 172.24.0.10 dev enp1s0f3
#      up route add -net 10.11.0.0/16 gw 172.24.0.11 dev enp1s0f3
#      up route add -net 10.12.0.0/16 gw 172.24.0.12 dev enp1s0f3
#      up route add -net 10.19.0.0/16 gw 172.24.0.19 dev enp1s0f3
#      up route add -net 10.20.0.0/16 gw 172.24.0.20 dev enp1s0f3
#      up route add -net 10.21.0.0/16 gw 172.24.0.21 dev enp1s0f3
#      up route add -net 10.22.0.0/16 gw 172.24.0.22 dev enp1s0f3
#      up route add -net 10.23.0.0/16 gw 172.24.0.23 dev enp1s0f3
#      up route add -net 10.24.0.0/16 gw 172.24.0.24 dev enp1s0f3
```

Recall that since the path emulators in the WAN Emulation Host are functionally equivalent to ethernet bridges, the physical interfaces associated with each path emulator are typically bridged together when the path emulation software is not running. The configuration for interface `br_2s0f0_eno2`, which bridges interfaces `enp2s0f0` and `eno2`, and interface `br_2s0f2_2s0f1`, which bridges interfaces `enp2s0f2` and `enp2s0f1`, accomplishes this.

When the automated experiment execution scripts start an experiment, the scripts need to bring down the bridge interfaces before running the path emulation software. When the experiment terminates, the path emulation software terminates and the bridge interfaces need to be brought back up. In order to do this, the automated experiment execution scripts must be able to identify the interfaces impacted by the experiment. Note that not all experiments will impact both bridge interfaces. For example, any experiment with a single-homed enclave will only impact one of the bridge interfaces. The automated experiment execution scripts require that the bridge interfaces be named in accordance to the following convention:

br_WANIF_LANIF

where *WANIF* is the name of the WAN-facing interface to be bridged and *LANIF* is the name of the LAN-facing interface to be bridged. If the name of either of these interfaces is longer than 5 characters, then the last 5 characters of the interface name are used, as this is typically all that is necessary to uniquely identify the interface. This is required because there is a limit on the length of the bridge interface names, typically 16 characters. When the link emulation software starts, a reference address is passed into the scripts that enables the scripts to determine the LAN-facing and WAN-facing interfaces that are part of the affected bridge. The reference address is automatically provided when an experiment is created using the experiment creation scripts described in the *Creating A New Experiment* section below and does not need to be provided by the user. For a more detailed explanation of the operation and capabilities of the link emulation software, see [here](#).

13.3.1.2 Configuring sudo

Because of the use of raw sockets for packet capture, running GNAT requires root privileges when executed. This can be accomplished by adding the appropriate users to the */etc/sudoers* file on the testbed nodes. Additionally, the automated experiment execution scripts generally start the GNAT components from a remote execution node via ssh commands. While it is possible, it is generally very inconvenient to have to provide a password for every sudo command that is executed as there are many during the course of starting and stopping an experiment. It is best to set up sudo on the testbed nodes so that no password is required. We typically run all experiments as the **iron** user. The following lines added to the */etc/sudoers* files give the **iron** user sudo privileges and indicate that no password is required when the **iron** user issues the sudo command:

```
# all the other users
iron    ALL=(ALL:ALL) NOPASSWD: ALL
```

13.3.1.3 Installing Applications

13.3.1.3.1 Installing mgen

The automated experiment execution scripts use *mgen* as the source and destination application for the experiment's traffic flows. If not installed on the testbed application nodes, this can be accomplished with the following command:

sudo apt-get install mgen

Following the installation, issuing the command

mgen

on the testbed application node produces the following output:

```
mgen: version 5.02
mgen: starting now ...
13:00:33.522278 START Mgen Version 5.02
13:00:33.522335 STOP
```

13.3.1.3.2 Installing gnuplot

The automated experiment execution scripts provide an option to post process the experiment results into goodput, delay, and loss plots at the destination node for each of the experiment's flows. In order to generate these plots, the *mgen* output files are processed with the *trpr* application and the *trpr* output is then plotted with *gnuplot*. We modified the *trpr* application to support plotting large numbers of flows. As such, the *trpr* application is built with the GNAT executables and is deployed when an experiment is run. However, *gnuplot* is also required to post process the experiment results and is not installed when an experiment runs. If *gnuplot* is not installed on the testbed application nodes, this can be accomplished with the following command:

sudo apt-get install gnuplot5-qt

Following the installation, issuing the command

gnuplot --version

on the testbed application node produces the following output:

```
gnuplot 5.0 patchlevel 3
```

13.4 Testbed Abstraction

13.4.1 Generic Terminology For Testbeds

A design goal for the automated experiment execution scripts is that they can be used without modification to run experiments on different testbeds. The first step to accomplish this is to start thinking about the testbeds in generic terms, i.e., as set of generic nodes connected by links. Consider Figure 38, which is an abstraction of the Example Physical Testbed described in the previous section.

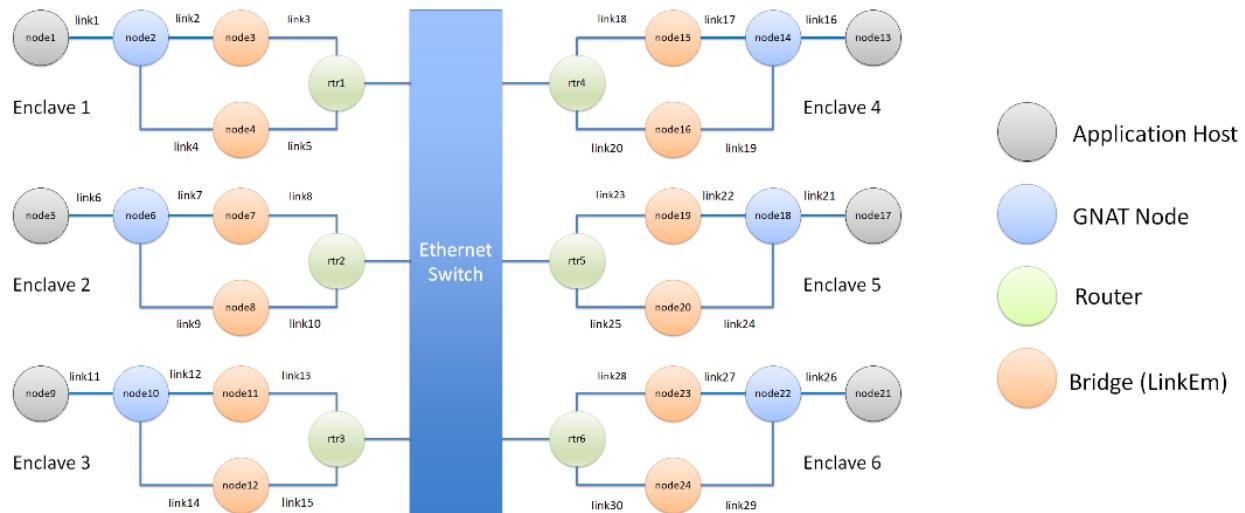


Figure 38. Abstraction of example physical testbed to generic nodes and links

One can now refer to the nodes and the node interfaces in the experiment configuration files in a generic fashion, for example node1 and node1:link1. Thinking about a testbed in this fashion provides an abstraction layer that provides the flexibility to run an experiment configured with these generic names to be run on many different physical instantiations of the topology. We call this the "experiment topology".

Notice that our enclaves have four nodes in them consisting of one application node, one GNAT node, and 2 link emulation nodes and 5 links connecting the nodes. A testbed topology defined in the above fashion has an *enclave node offset* of $((\text{enclave id} - 1) * 4)$ and a *enclave link offset* of $((\text{enclave id} - 1) * 5)$. Our experiment topology assigns generic node identifiers in the following way for each enclave:

- The application nodes are (*enclave node offset* + 1 through N), where N is the number of application nodes in the enclave. In the single application node example above, these are **node1** for **Enclave 1**, **node5** for **Enclave 2**, **node9** for **Enclave 3**, etc.
- The GNAT node is (*enclave node offset* + (N+1)), **node2** for **Enclave 1**, **node6** for **Enclave 2**, **node10** for **Enclave 3**, etc. in the above example.
- The link emulation nodes are (*enclave node offset* + (N+2) and (N+3)), **node3** and **node4** for **Enclave 1**, **node7** and **node8** for **Enclave 2**, **node11** and **node12** for **Enclave 3**, etc. in the above example.

Our experiment topology assigns generic link identifiers in the following way for each enclave:

- The links between the application nodes and the GNAT nodes are (*enclave link offset* + 1 through N), where N is the number of application nodes in the enclave. This is **link1** for **Enclave 1**, **link6** for **Enclave 2**, **link11** for **Enclave**, etc. in the above example.
- The link between the GNAT node and the first link emulation node is (*enclave link offset* + (N+1)), **link2** for **Enclave1**, **link7** for **Enclave 2**, **link12** for **Enclave 3**, etc. in the above example.
- The link between the first link emulation node and the black cloud is (*enclave link offset* + (N+2)), **link3** for **Enclave 1**, **link7** for **Enclave 2**, **link12** for **Enclave 3**, etc. in the above example.
- The link between the GNAT node and the second link emulation node is (*enclave link offset* +(N+3)), **link4** for **Enclave 1**, **link9** for **Enclave 2**, **link14** for **Enclave 3**, etc. in the above example.
- The link between the second link emulation node and the black cloud is (*enclave link offset* +(N+4)), **link5** for **Enclave 1**, **link10** for **Enclave 2**, **link15** for **Enclave 3**, etc. in the above example.

By adhering to the assignment of node and link identifiers described above for the experiment topology, the automated experiment execution scripts can figure out the generic node and link identifiers for any given enclave id.

The next section describes the testbed topology file, which binds the above abstract version of the testbed with a real physical instantiation.

13.4.2 Testbed Topology Files

To run on a real testbed, we need to bind the abstracted experiment topology illustrated above with an instantiated physical topology of a real testbed. The goal is to have a common set of configuration files that can be bound to the desired physical nodes at runtime. The Testbed Topology File is the glue that binds the abstracted testbed description to a set of real physical nodes. Each Testbed Topology File may contain the following lines:

- '#' lines: This is a comment line.
- suffix <host name suffix>: This is the suffix to use when creating the fully qualified host names. This line can include the EXP_NAME keyword which will be substituted with the

provided experiment name at configuration time. This feature is only used for ISI DeterLab experiments because the assigned hostnames for the DeterLab testbed nodes contains the name of the experiment in the fully qualified name.

- **exp_base_dir:** This is the base directory for the experiments on the remote testbed nodes.
- **results_location:** This is the directory that the experiment results will be written to on the local machine (the machine from which the experiment was executed).
- **remote_execution_node:** This is the node that the experiment will be executed from. This is an optional entry and is typically only set for DeterLab experiments, where the value is usually set to users.isi.deterlab.net.
- **num_enclaves:** The total number of enclaves in the testbed.
- **app_nodes_per_enclave:** The number of application nodes in each enclave.
- **le_nodes_per_enclave:** The number of Link Emulator nodes in each enclave. This must be 1 or 2.
- **linkX nodeA nodeB:** This describes a link between 2 nodes. The current scripts do not use this information, however, it is anticipated that this may be useful for automatically creating the DeterLab configuration files as a future task.
- **nodeA hostname linkX=a.b.c.d,linkY=e.f.g.h:** This describes the information for generic nodeA, including its hostname and the IP Addresses of its links.

The Testbed Topology Files reside in the <IRON install dir>/IRON/experiments/testbeds/ directory.

Following is the testbed topology file for the Example Testbed previously described:

Example Testbed Topology File

```
# Topology file for BBN testbed
#
#   Enclave 1:                                     Enclave 4:
#
#       link1      link2      link3          link18      link17      link16
# node1 --- node2 --- node3 ---+      +---+      +--- node15 --- node14 --- node13
#           |          |          |           |          |          |
#           |          rtr1 --|          |-- rtr4          |
#           |          |          |           |          |          |
#           +---+ node4 ---+      |          |           +---+ node16 -----+
#               link4      link5          |          |           link20          link19
#               |          |
#               |          |
#   Enclave 2:                                     Enclave 5:
#
#       link6      link7      link8          link23      link22      link21
# node5 --- node6 --- node7 ---+      | S |      +--- node19 --- node18 --- node17
#           |          |          | W |          |          |
#           |          rtr2 --| I |-- rtr5          |
#           |          |          | T |          |
#           +---+ node8 ---+      | C |      +---+ node20 -----+
#               link9      link10         | H |           link25          link24
#               |          |
#               |          |
#   Enclave 3:                                     Enclave 6:
#
#       link11     link12     link13          link28      link27      link26
# node9 --- node10 -- node11 ---+      | |          +---+ node23 --- node22 --- node21
#           |          |          | |          |          |

```

```
# | rtr3 --| |-- rtr6 |
# | | | | |
# +----+ node12 ---+ +---+ +--- node24 -----+
# link14 link15 link30 link29

suffix bbn.com
exp_base_dir /home/${USER_NAME}
results_location ${HOME}/iron_results

num_enclaves 6
app_nodes_per_enclave 1
le_nodes_per_enclave 2

link1 node1 node2
link2 node2 node3
link3 node3 rtr1
link4 node2 node4
link5 node4 rtr1
link6 node5 node6
link7 node6 node7
link8 node7 rtr2
link9 node6 node8
link10 node8 rtr2
link11 node9 node10
link12 node10 node11
link13 node11 rtr3
link14 node10 node12
link15 node12 rtr3
link16 node13 node14
link17 node14 node15
link18 node15 rtr4
link19 node14 node16
link20 node16 rtr4
link21 node17 node18
link22 node18 node19
link23 node19 rtr5
link24 node18 node20
link25 node20 rtr5
link26 node21 node22
link27 node22 node23
link28 node23 rtr6
link29 node22 node24
link30 node24 rtr6

# Enclave 1
node1 gnat-app1 link1=10.1.3.2
node2 gnat1 link1=10.1.3.1,link2=10.1.1.2,link4=10.1.2.2
node3 gnat-le1 link2=10.1.1.102,link3=10.1.1.101
node4 gnat-le1 link4=10.1.2.102,link5=10.1.2.101

# Enclave 2
node5 gnat-app2 link6=10.2.3.2
node6 gnat2 link6=10.2.3.1,link7=10.2.1.2,link9=10.2.2.2
node7 gnat-le2 link7=10.2.1.102,link8=10.2.1.101
node8 gnat-le2 link9=10.2.2.102,link10=10.2.2.101

# Enclave 3
node9 gnat-app3 link11=10.3.3.2
node10 gnat3 link11=10.3.3.1,link12=10.3.1.2,link14=10.3.2.2
node11 gnat-le3 link12=10.3.1.102,link13=10.3.1.101
node12 gnat-le3 link14=10.3.2.102,link15=10.3.2.101
```

```
# Enclave 4
node13 gnat-app4 link16=10.4.3.2
node14 gnat4 link16=10.4.3.1,link17=10.4.1.2,link19=10.4.2.2
node15 gnat-le4 link17=10.4.1.102,link18=10.4.1.101
node16 gnat-le4 link19=10.4.2.102,link20=10.4.2.101

# Enclave 5
node17 gnat-app5 link21=10.5.3.2
node18 gnat5 link21=10.5.3.1,link22=10.5.1.2,link24=10.5.2.2
node19 gnat-le5 link22=10.5.1.102,link23=10.5.1.101
node20 gnat-le5 link24=10.5.2.102,link25=10.5.2.101

# Enclave 6
node21 gnat-app6 link26=10.6.3.2
node22 gnat6 link26=10.6.3.1,link27=10.6.1.2,link29=10.6.2.2
node23 gnat-le6 link27=10.6.1.102,link28=10.6.1.101
node24 gnat-le6 link29=10.6.2.102,link30=10.6.2.101
```

Please refer to the testbed files in the GNAT distribution, located in the <IRON install dir>/IRON/experiments/testbeds/ directory for more examples, including the testbed files for running on the ISI DeterLab testbed.

13.5 Experiment Configuration Templates

While configuring experiments in terms of the nodes and links in the testbed topology file is possible and ultimately required by the automated experiment execution scripts, another layer of abstraction is provided that simplifies the configuration of an experiment. Configuration file templates are supported by the experiment execution scripts and provide the experimenter with a mechanism to describe configuration items in "notional" enclave terminology. This enables an experiment to be described to run on any combination of real physical enclaves without having to identify the real physical enclaves that will be used. As an example, consider a 3 Enclave experiment. The configuration templates enable the experimenter to describe the experiment in terms of Enclaves 1, 2, and 3, even if the experiment will run on a different set of physical Enclaves (e.g., Enclaves 6, 8, and 10). At runtime, the experiment execution scripts map "notional" enclaves to physical enclaves. The physical enclaves are then used to generate the node specific configuration files that are required to execute the experiment on the set of assigned real testbed nodes.

The experiment execution scripts accomplish this by substituting "notional" enclave replacement strings, strings that are enclosed in '\$' characters in the experiment configuration templates, with the information that is extracted from the testbed topology file. These replacement strings have the following format: *\$replacement_string\$*. The following set of configuration template replacement strings are inserted into the configuration templates when creating an experiment, fully described in the *Creating A New Experiment* section below, and are generally not modified by an experimenter:

- ***\$enclaveX_appY_wan_addr\$***: Refers to the WAN-facing address of application Y in Enclave X.
- ***\$enclaveX_iron_node\$***: Refers to the GNAT node in Enclave X.
- ***\$enclaveX_iron_lan_addr\$***: Refers to the LAN-facing address of the GNAT node in Enclave X.

- **\$enclaveX_iron_lan_link\$**: Refers to the LAN-facing generic link identifier of the GNAT node in Enclave X.
- **\$enclaveX_iron_wanY_link\$**: Refers to the WAN-facing generic link identifiers of the GNAT node in Enclave X. Recall that we typically create experiment testbeds that are dual homed. This directive enables us to refer to the appropriate WAN-facing generic link identifiers.
- **\$enclaveX_iron_wanY_addr\$**: Refers to one of the WAN-facing addresses of the GNAT node in Enclave X. Recall that we typically create experiment testbeds that are dual homed. This directive enables us to refer to the appropriate WAN-facing address.

While the above configuration template replacement strings are not generally modified by the experimenter when configuring an experiment, they can be observed in the configuration template files that are generated when the experiment is created. There are, however, a set of configuration template replacement strings that are typically provided by the experimenter when finalizing the experiment details. These replacement strings are as follows:

- **\$enclaveX_appY_node\$**: Refers to application host Y in Enclave X.
- **\$enclaveX_leY_node\$**: Refers to link emulation node Y in Enclave X.

If it is necessary to collect packet traces to further examine the results of the experiment, described in the *Common Debugging Techniques* section, the following configuration template replacement strings may need to be provided:

- **\$enclaveX_appY_wan_link\$**: Refers to the WAN-facing generic link identifier of application Y in Enclave X.
- **\$enclaveX_leY_lan_link\$**: Refers to LAN-facing generic link identifier of link emulation node Y in Enclave X.
- **\$enclaveX_leY_wan_link\$**: Refers to WAN-facing generic link identifier of link emulation node Y in Enclave X.

13.6 Creating A New Experiment

So far, we have described: physical experiment testbeds, the abstraction of these testbeds to generic node and link identifiers contained in the testbed topology file, and a further abstraction, configuration file templates, which enables the experimenter to configure experiments in "notional" enclave terminology. A number of system experiments exist (in the **<IRON install dir>/IRON/experiments/** directory) and are documented [here](#). Next, we will outline the process of creating a new experiment from scratch.

13.6.1 Define Experiment

Defining a set of objectives is the first step in creating a new experiment. At a high level, these objectives include the following:

- Characterization of "what" the initial GNAT network provides: This includes the "plumbing" required to enable the enclaves to communicate with each other and the initial link configuration between the enclaves.
- Characterization of "how" the GNAT network will be utilized: This includes a specification of the flows, the desired utility functions for the flows, and the set of network impairments.

For the purposes of this tutorial, we will describe how to create the [3-node-system](#) experiment. The high-level details of the experiment are summarized in the Figure 39 below:

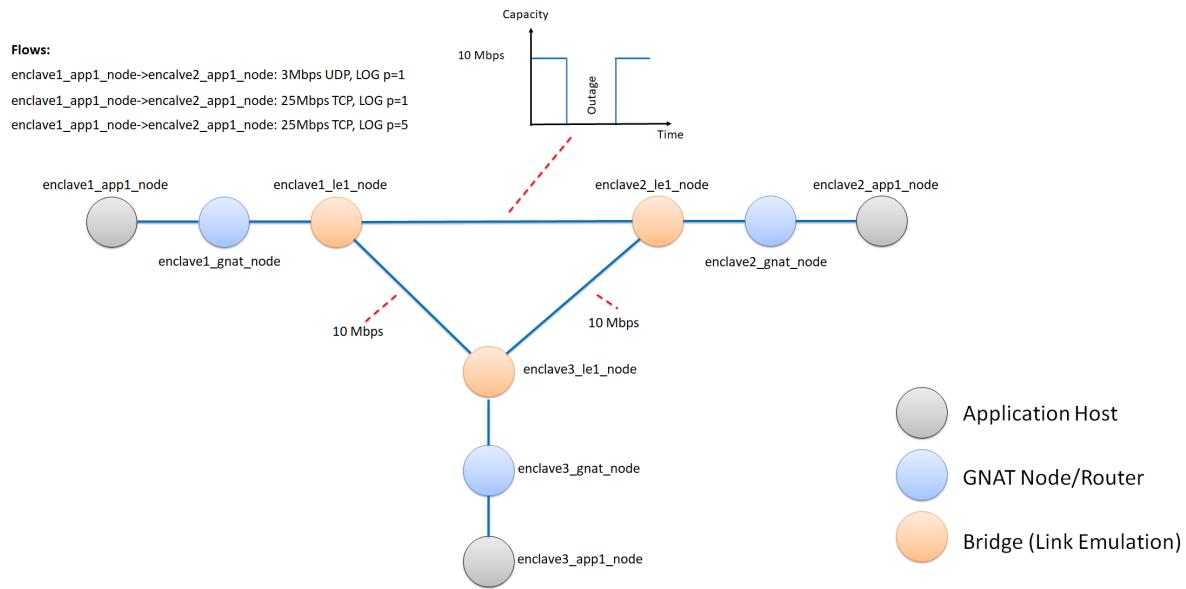


Figure 39. Summary details for 3-node-system experiment

Note that the nodes in the figure above are labelled in "notional" enclave terminology. Recall, for example, that *enclave1_app1_node* refers to application node 1 in Enclave 1. As previously described, thinking about the experiment like this abstracts the experimenter away from the real physical hardware that the experiment will run on and enables the experiment to run on any collection of real, physical enclaves that are available in the target testbed. The above figure contains all of the information that is required to continue with the process of creating our new experiment. It contains the details describing the nature of the desired GNAT network, the "what", and the details of how that network is to be utilized (in terms of the desired flow dynamics and network impairment events), the "how". We can now proceed with gathering up the inputs and generating the experiment configuration files.

13.6.2 Generate Experiment Configuration Files

The *create_exp.sh* script, located in the <IRON install dir>/IRON/experiments/scripts/ directory, generates the configuration files and templates for our experiment. A full description of the script and all of its options and arguments is obtained by running the following command:

create_exp.sh -h

A more detailed description of the most commonly included options and arguments is provided below.

- The base directory option, -B, identifies the location for the generated experiment configuration files and templates.
- The duration option, -l, specifies the duration of the experiment, in seconds. The default value is 60.
- The delay option, -d, identifies the initial delay value, in ms, for the links between the enclaves. The default value is 10.
- The throttle option, -t, identifies the initial rate, in Kbps, for the links between the enclaves. The default value is 100000.
- The size option, -b, identifies the buffer size, in bytes, for the links between the enclaves. The default value is 500000.
- The arguments to the script include the number of experiment enclaves and a name for the experiment.

Currently, the *create_exp.sh* script only supports the creation of a full mesh topology of the enclaves. A future enhancement might include providing a specification configuration file that express a more general connectivity matrix of the enclaves. If a full mesh is not what is desired for the experiment, this can be adjusted following the creation of the experiment. Additionally, the *create_exp.sh* script creates an initial set of links between the enclaves that are homogeneous. If a homogeneous set of links is not what is desired for the experiment, this can be adjusted following the creation of the experiment.

13.6.2.1 Gather Inputs

The next step is to gather the inputs for the experiment creation script. The following list of configuration inputs is sufficient to create our experiment:

- Number of Enclaves: 3
- Experiment Name: 3enclave_example_exp
- Link Characteristics: 10 Mbps, 10 ms delay, 50 Kbyte buffers, Packet Error Rate Error Model, No Jitter Model
- Base Directory: <IRON installation dir>/IRON/experiments
- Flow Definitions: 2 TCP flows and 1 UDP flow from Enclave 1 to Enclave 2
- Service Definitions: 1 high priority and 1 low priority TCP flow, and 1 low priority UDP flow
- Network Impairment: Direct link between Enclaves 1 and 2 severed during run for 10 seconds

13.6.2.2 Script Execution

Now that we have defined the objectives for our experiment and collected the inputs for the experiment creation script, we are ready to generate the experiment's configuration files and templates. For our example, we will provide the destination directory for the generated experiment

configurations, set the link rates to 10 Mbps, 10 ms delay, 50 Kbyte buffers, and provide the required number of enclaves and experiment name to the *create_exp.sh* script. The following commands will generate our experiment:

```
cd <IRON installation dir>/IRON/experiments/scripts/  
  
../create_exp.sh -B <IRON installation dir>/IRON/experiments -t 10000 -d 10 -b 50000  
3 3enclave_example_exp
```

13.6.2.3 Generated Outputs

The experiment configuration files are placed in the directory provided to the experiment creation script, **<IRON installation dir>/IRON/experiments/3enclave_example_exp** in our example. The generated outputs are shown below:

Generated Experiment Configuration Files

<IRON installation dir>/IRON/experiments/3enclave_example_exp

```
└── cfgs
    ├── amp_common.cfg
    ├── amp_services tmpl
    ├── bin_map.tmpl
    ├── bpf_common.cfg
    ├── bpf_enclave1.tmpl
    ├── bpf_enclave2.tmpl
    ├── bpf_enclave3.tmpl
    ├── lem_init.tmpl
    ├── lem tmpl
    ├── process.cfg
    ├── system.cfg
    ├── tcp_proxy_common.cfg
    ├── tcp_proxy_enclave1.tmpl
    ├── tcp_proxy_enclave2.tmpl
    ├── tcp_proxy_enclave3.tmpl
    ├── traffic tmpl
    ├── udp_proxy_common.cfg
    ├── udp_proxy_enclave1.tmpl
    ├── udp_proxy_enclave2.tmpl
    └── udp_proxy_enclave3.tmpl
    └── enclaves.cfg
    └── exp.tmpl
```

At this point, the generated set of configuration files and templates contain the directives to construct a GNAT network enabling our 3 enclaves to communicate with each other. The *.cfg files contain configuration information that is not node specific. The *.tmpl files are the configuration template files containing enclave specific information in the form of configuration template replacement strings, described in the *Experiment Configuration Templates* section above. The *exp.tmpl* file contains the experiment's high-level configuration information including: length of the experiment, application, GNAT, and link emulation nodes in the experiment, and any desired

packet capture nodes in the experiment. The *enclaves.cfg* file contains the default set of testbed enclaves to utilize for the experiment and is only used when there is no testbed reservation server host configured, as described in the *Experiment Execution* section below.

Note: The bolded configuration templates (*amp_services tmpl*, *lem tmpl*, and *traffic tmpl*) **must** be modified after they have been created.

Recall that the experiment creation script provides the "what", of an GNAT network over which the enclaves can communicate. The experimenter needs to configure "how" the GNAT network is to be utilized, which include the network services that will run, the details of how the (emulated) links will operate as well as the traffic profiles. . Modifying these files is the means of accomplishing this. The remaining configuration template files typically do not need to be modified unless the experimenter wishes to deviate from the default GNAT network experiment configuration. The configuration files and templates contain comments that explain each of the configurable items, should the experimenter find it necessary to change any of them.

13.6.3 Optionally Modifying the GNAT Network "Plumbing"

Before proceeding with configuring "how" the generated GNAT network is to be used, we must ensure that the generated network aligns with the experiment objectives. Recall that the *create_exp.sh* script that was used to create our new experiment creates a full mesh of enclaves with homogeneous links between each enclave. If either of these script creation assumptions is not desired for the created experiment, the experiment configuration templates can be modified before proceeding with finalizing our experiment.

13.6.3.1 Modifying Generated Full Mesh

The configuration of the connectivity between the experiment enclaves is contained in the *bpf_enclaveX tmpl* and *lem_init tmpl* configuration template files. The *PathController.X.Type* and *PathController.X.Endpoints* lines in the *bpf_enclaveX tmpl* file can be removed for any links in the created configuration template that are not required for the experiment. If any lines are removed, the *Bpf.NumPathControllers* line **must** also be modified to reflect this change. Note that the *PathController.X.Type* and *PathController.X.Endpoints* entries **must** contain X values between 0 and *Bpf.NumPathControllers*.

Additionally, any Paths in the initial link emulation configuration file template, *lem_init tmpl*, can be removed, if desired. This will eliminate the step of setting up link emulation paths that will not be used.

Note that the modifications described in this section are optional and need only be done if the desired experiment connectivity is not a full mesh of the experiment enclaves.

13.6.3.2 Modifying Initial Link Characteristics

The initial link configuration information is located in the *lem_init tmpl* configuration template file. The configuration information for any link that should deviate from generated configuration

template can be made in this file. See the link and path emulation utility, documented in Section 12.1 for the details explaining how to configure the links to match the desired starting link characteristics. Note that this step in the process is optional and need only be done if the starting states of the links are not homogeneous.

13.6.4 Finalize Experiment Details

Thus far, we have generated a set of configuration files that enable our enclaves to communicate with each other. We now need to configure "how" we will use this network of enclaves to achieve the experiment objectives. This includes configuring the experiment flows, utility function definitions, and network impairments.

13.6.4.1 Flow Definitions

The flow configuration information is entered in the *traffic tmpl* configuration template. Three types of flows can be configured in this file, tcp, udp, or short_tcp. The format of the directive for TCP and UDP flows is as follows:

```
protocol src dest num_flows src_port dst_port start_time end_time packet_size
data_rate
```

The format of the directive for short TCP flows is as follows:

```
protocol src dest num_flows lo_port hi_port start_time end_time flow_size_bytes inter-
flow_duration
```

Refer to the comment section in the *traffic tmpl* file for a complete description of the flow definition directives. Our example experiment has 3 flows, 1 UDP flow and 2 TCP flows. The next step is to configure the 3 flows for our example experiment as follows:

```
udp $enclave1_app1_node$ $enclave2_app1_node$ 1 30777 30777 10.0 50.0 1024 3Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ 1 29778 29778 10.0 50.0 1024 25Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ 1 29779 29779 10.0 50.0 1024 25Mbps
```

Note that the source and destination of each of the flows are specified in configuration template replacement string notation described previously in the *Experiment Configuration Template* section above. Since our enclaves have only 1 application node and our flows are from Enclave 1 to Enclave 2, the above replacement strings define our flows and provide the automated experiment execution scripts with the required information. The automated experiment execution scripts will replace instances of the configuration template replacement strings, i.e., **\$enclave1_app1_node\$**, with the appropriate generic node id extracted from the testbed topology file.

13.6.4.2 Flow Service Definitions

Now that we have defined a set of flows for the experiment, we need to define the Service Definitions for the flows. The format for defining the Service Definitions is fully described in the comment section of the *amp_services tmpl* configuration file.

Our experiment objectives call for 1 high priority Log utility TCP flow, 1 low priority Log utility TCP flow, and 1 low priority Log utility UDP flow. The following entries accomplish this:

```
2 tcp_proxy add_service 29778-29778;type=LOG:a=10:m=25000000:p=1:label=low_prio_tcp;
2 tcp_proxy add_service 29779-29779;type=LOG:a=10:m=25000000:p=5:label=high_prio_tcp;
2 udp_proxy add_service 30750-
30799;1/1;1500;0;0;120;0;type=LOG:a=10:m=25000000:p=1:label=low_prio_udp
```

Note that the port ranges of our utility function definitions **must** match the ports in our flow definitions. It is also important to note that the flow service definition lines should be added to the *amp_services tmpl* file in the section labeled "User defined flow and service definitions should be placed here:". The default service definitions, appearing at the end of the *amp_services tmpl* file, **must not** be deleted and must appear as the last entries in the file.

13.6.4.3 Network Impairment Definitions

The link emulation nodes in our experiment infrastructure handle experimenter defined impairments. The details of the link and path emulation utility is fully documented [here](#). Our experiment calls for a network impairment 20 seconds into the run, lasting for 10 seconds, that severs then restores the direct link between Enclaves 1 and 2. The network impairment information is entered in the *lem tmpl* configuration file. The format of the entries is fully described in the comment section of the file. The following entries will accomplish the desired network impairment event:

```
sleep 20
$enclave1_le1_node$ set path:2 per:1.0
$enclave2_le1_node$ set path:1 per:1.0
sleep 10
$enclave1_le1_node$ set path:2 per:0.0
$enclave2_le1_node$ set path:1 per:0.0
```

The experiment creation script creates an GNAT network that is configured to support egress modeling, meaning that network impairment modeling is employed as packets are leaving an enclave. Two link emulation nodes need to be modified in order to completely sever the link between Enclaves 1 and 2. Line 2 modifies the Enclave 1 link emulation node, setting the Packet Error Rate on Path 2, the path to Enclave 2, to 1.0 (a 100% packet error rate). Similarly, line 3 modifies the Enclave 2 link emulation node, setting the Packet Error Rate on Path 1, the path to Enclave 1, to 1.0. The end result of these link emulation directives drops all packets leaving Enclave 1 destined Enclave 2 on the direct path, and vice versa. During the outage, the only remaining path from Enclave 1 to Enclave 2 is the 2-hop path via Enclave 3. Ten seconds later, the action is undone by resetting the Packet Error Rates back to 0.0, restoring the direct path between Enclaves 1 and 2. Again, note that the entries in this file are specified in configuration template replacement string notation.

13.6.4.4 Optional Additional Configuration Modifications

At this point in the creation of our new experiment, we have the script generated set of configuration files and templates and the experimenter-modified templates describing how the GNAT network is to be used. There are many configurable items in the generated configuration files and templates, any of which can be modified by the experimenter prior to experiment execution. If desired, this is the time to make modifications to the GNAT component configurable items.

The experiment execution scripts also support a feature, referred to as parameterized experiments, which allows the experimenter to run a series of experiments, each having its own unique configuration. Note that this is optional and need not apply to all created experiments.

13.6.4.4.1 Parameterized Experiments

The experiment execution scripts permit an experimenter to run a series of experiments in which one or more configurable items are changed from run to run while maintaining the same infrastructure. This is accomplished by inserting special tags into a configuration file or template and providing a parameter input file, named *params.txt*, which contains the values for the various tags. If a *params.txt* file is created for the experiment, it **must** be placed in the experiment's *cfgs* directory. As an illustration, consider a series of experiments where the experimenter wants to change the number of high priority TCP flows to observe its effect on system behavior. To accomplish this, the experimenter modifies the experiment's *traffic tmpl* configuration template as follows:

Parameterized traffic tmpl

```
udp $enclave1_app1_node$ $enclave2_app1_node$ 1 30777 30777 10.0 50.0 1024 3Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ %NUM_HI_PRIO_TCP_FLOWS% 29778 29778 10.0
50.0 1024 25Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ 1 29779 29779 10.0 50.0 1024 25Mbps
```

The line of interest from the above snippet from the *traffic tmpl* configuration template file is line 2. Notice that the value for the number of flows on this line (previously "1") has been replaced by **%NUM_HI_PRIO_TCP_FLOWS%**, a parameter substitution tag. The names of parameter substitution tags are chosen by the experimenter and adhere to the following format:

`%tag_name%`.

The automated experiment scripts will replace instances of the parameter substitution tags with the values extracted from the parameter input file, *params.txt*, which is illustrated below for our example experiment:

params.txt

```
# Parameter values for the number of high priority TCP flows.
NUM_HI_PRIO_TCP_FLOWS = 1 5 10
```

This *params.txt* input file specifies 3 values (1, 5, and 10) for the parameter substitution tag, **NUM_HI_PRIO_TCP_FLOWS**. Notice that the tag delimiters, '%', are not included in the parameters input file. Each line in the file contains the tag name and the set of values that the tag can be assigned.

Note: All tags defined in this file **must** have the same number of values. When a parameterized experiment is configured, multiple run directories are created, one for each set of configuration options defined in the *params.txt* input file. This will be described below in the *Experiment Execution* section.

13.7 Preparing Experimenter's Run-Time Environment

We have now successfully created our new experiment and are almost ready to execute it. However, we need to ensure that our environment is prepared before we can run our new experiment.

13.7.1 Shell (login environment) Set-up

Prior to running an GNAT experiment, refer to the <IRON install dir>/IRON/iron/RE-ADME.txt file for detailed information about how to set up your shell to build the GNAT software and run the automated experiment execution scripts. The contents of this file are provided below as a convenience.

IRON README.txt file

Quick-Start Guide

1. Make sure you are using bash, tcsh or csh.
2. Set the "IRON_HOME" environment variable to where this file is located

```
export IRON_HOME=<IRON install dir>/IRON/iron"
```

or

```
setenv IRON_HOME <IRON install dir>/IRON/iron
```

where <IRON install dir> is the location of the IRON installation. This may be placed in your .bashrc or .cshrc file for convenience. However, make sure that this file tests if this environment variable has already been set before setting it. A bash example of this is:

```
if [ -z "$IRON_HOME" ] ; then
    export IRON_HOME=<IRON install dir>/IRON/iron
fi
```

A tcsh/csh example of this is:

```
if (! $?IRON_HOME ) then
    setenv IRON_HOME      <IRON install dir>/IRON/iron
endif
```

3. Source the appropriate setup file, e.g., for bash:

```
cd $IRON_HOME
. setup/debug.bash
```

or for tcsh/csh:

```
cd $IRON_HOME
source setup/debug.csh
```

Substitute the correct path and setup file name (currently debug.bash,

optimized.bash, debug.csh, or optimized.csh) depending on your shell and how you would like the software built.

4. If the UNIX platform you are using is something different than Linux with a 3.13 or 3.2 kernel, then create the needed debug and optimized style files for your platform in the iron/build directory using the Linux_3.2_debug and Linux_3.2_optimized style files as templates. The platform name may be determined using the command "uname -s", and the version number (of the form X.Y) may be determined using the command "uname -r" and ignoring everything after the first and second numbers. Replace instances of -DLINUX_3_2 with -DLINUX_X_Y.

5. To build all native code (C/C++) software, perform the following:

```
cd $IRON_HOME  
make clean  
make
```

6. To build the unit test code, perform the following:

```
cd $IRON_HOME  
make -f makefile.unittest clean  
make -f makefile.unittest
```

7. To execute the unit test code, perform the following:

```
cd $IRON_HOME  
.bin/{style-file-name}/testironamp  
.bin/{style-file-name}/testironbpf  
sudo .bin/{style-file-name}/testironcommon  
.bin/{style-file-name}/testirontcpproxy  
.bin/{style-file-name}/testironudpproxy
```

where {style-file-name} is the name of the file described in step 4 - for example Linux_3.2_debug.

The unit tests print out dots while they are executing and a status message.

8. To build the IRON documentation, perform the following:

```
cd $IRON_HOME/doc  
make docs
```

9. To access the IRON documentation, use a web browser to open the file \$IRON_HOME/doc/html/index.html

10. To execute the IRON python unit test code, cd \$IRON_HOME/python and then follow the directions in the README.txt file.

13.7.2 Experimenter ssh Configuration

GNAT experiments are typically run as a shared user so that not every experimenter requires special sudo privileges. We typically create a shared user, **iron** for the purposes of this tutorial, to accomplish this. After creating the **iron** user public/private key pair (named *id_rsa_iron* in our example), the private-key file is distributed to all users that will run GNAT experiments. The experimenter typically puts the private key in their *~/.ssh* directory. The public-key file is added to the */home/iron/.ssh/authorized_keys* file on each testbed machine. Each experimenter can then

modify their ssh configuration so that when an ssh command is made to a testbed machine, the command is executed as the **iron** user with the *id_rsa_iron* key for non-password authentication.

The following lines can be added to your ssh config file, *~/.ssh/config*, to accomplish this for our example experiment testbed:

```
Host gnat-app?
  HostName %h.bbn.com
  Port 22
  User iron
  IdentityFile ~/.ssh/id_rsa_iron
  StrictHostKeyChecking=no
  LogLevel=quiet
Host gnat-app?.bbn.com
  Port 22
  User iron
  IdentityFile ~/.ssh/id_rsa_iron
  StrictHostKeyChecking=no
  LogLevel=quiet

Host gnat?
  HostName %h.bbn.com
  Port 22
  User iron
  IdentityFile ~/.ssh/id_rsa_iron
  StrictHostKeyChecking=no
  LogLevel=quiet
Host gnat?.bbn.com
  Port 22
  User iron
  IdentityFile ~/.ssh/id_rsa_iron
  StrictHostKeyChecking=no
  LogLevel=quiet

Host gnat-le?
  HostName %h.bbn.com
  Port 22
  User iron
  IdentityFile ~/.ssh/id_rsa_iron
  StrictHostKeyChecking=no
  LogLevel=quiet
Host gnat-le?.bbn.com
  Port 22
  User iron
  IdentityFile ~/.ssh/id_rsa_iron
  StrictHostKeyChecking=no
  LogLevel=quiet
```

The IdentityFile entries in the above configuration **must** point to the **iron** user private key that was created. The above configuration ensures that all ssh commands to our testbed machines will occur as the **iron** user. This is the case when either the hostname, i.e., **gnat1**, or the fully qualified domain name, i.e., **gnat1.bbn.com**, is used. Note that there is nothing special about the **iron** user. The shared user account can have any name, however, this name **must** be provided to the automated experiment execution scripts as described in the *Running An Experiment* section below.

13.8 Reserving Testbed Nodes

It is generally true that only one instance of the various GNAT components can be run on a host at a time. An exception to this rule is the link emulation component. For dual homed experiments, we typically run 2 instances of the link emulation executable on a single host. Since a testbed Enclave can support only one experiment at a time, we need an automated mechanism to reserve testbed nodes for our experiments so that we don't conflict with experiments already in progress. The GNAT testbed reservation system accomplishes this.

The automated experiment execution script, *reserve_ctl.sh*, interacts with the testbed reservation script that runs on the testbed reservation server to manage the reservation of the physical nodes in the various experiment testbeds. The testbed reservation system uses standard Linux file locking mechanisms (*lockfile-create* and *lockfile-remove* system calls) to reserve testbed nodes as an atomic operation and works with any testbed topology file, even new ones. The *lockfile-progs* package must be pre-installed on the reservation server. This can be accomplished with the following command:

```
sudo apt-get install lockfile-progs
```

The following high-level steps occur when interacting with the testbed reservation system:

1. Create lockfile for the testbed
2. Modify testbed reservation state
3. Remove lockfile for testbed

The testbed lockfiles are located in the following directory:

/run/lock/iron_testbeds/

The testbed reservation system must support many experiments being run by many experimenters. To satisfy this objective, it runs as a shared user (typically the *iron* user at BBN) on the reservation server so that a global view of the status of all testbeds is available. This common user is typically the user that the experiments run as, described in the *Experiment Execution* Section below. The testbed reservation state for a testbed is kept in the following location on the reservation server:

/home/<shared user>/testbed_reservations/<testbed_name>/

In this directory are enclave lock files that include who locked the testbed enclave when. Note that the files in this directory are only text files and modifying them without grabbing a lockfile is both possible and dangerous as the integrity of the testbed may be compromised. To ensure proper operation, the *reserve_ctl.sh* script should be used as it ensures that the policy of modifying the enclave lock files only occurs when the testbed lock file is acquired.

When interacting with the testbed reservation system for a testbed, the reservation script takes the following detailed actions:

1. The real testbed lock file, located in the `/run/lock/iron_testbeds/` directory, is created with the `lockfile-create` system command. If the lockfile already exists, the script waits until either the lockfile can be created or times out.
2. Once the testbed lock file is created, the `/home/<user>/testbed_reservations/<testbed_name>/` directory is "owned" by the script and can be modified.
3. Enclave lock files for reserved enclaves are created in the `/home/<user>/testbed_reservations/<testbed_name>/` directory. These files identify the owner of the enclave and when it was locked.
4. The real testbed lock file, located in the `/run/lock/iron_testbeds/` directory, is removed. After this happens, testbed enclaves can be reserved or released by other experimenters.

The `reserve_ctl.sh` script is fully integrated with the automated experiment execution scripts and typically is not run stand-alone, however, it can be, if necessary. A full description of the script and all of its options and arguments is obtained by running the following command:

```
reserve_ctl.sh -h
```

A more detailed description of the most commonly included options and arguments is provided below.

- The base directory option, `-b`, identifies the testbed topology file base directory. This is typically the `<IRON install dir>/IRON/experiments/testbeds/` directory.
- The `-l` option identifies the desired physical enclaves requested, as a colon separated list. For example, `-l 1:2:3`, is used to request locking physical Enclaves 1, 2, and 3 (discussed in the *Testbed Topology Files* section). Note that this action should not be done in a stand-alone mode as it will occur automatically when an experiment is run.
- The `-L` option identifies the desired number of physical enclaves requested. This option is used when it doesn't matter which real physical enclaves are used for the experiment. For example, `-L 3`, is used to request 3 physical enclaves. Note that this action should not be done in a stand-alone mode as it will occur automatically when an experiment is run.
- The query option, `-q`, queries the status of the testbed.
- The release option, `-r`, releases the colon separated list of enclaves. For example, `-r 1:2:3`, is used to release physical Enclaves 1, 2, and 3. This can be used by the experimenter if the automated experiment execution scripts fail for any reason to release the enclaves that were locked for the failed experiment.
- The server option, `-s`, identifies the reservation server host.
- The user option, `-u`, identifies the user the script is to run as. The state of the testbed reservation locks will be found in the `/home/<user>/testbed_reservations/` directory on the reservation server. All experimenters should use a shared user here.
- The testbed topology file for the desired testbed is provided as an argument to the script.

The following command is used to query the status of our example testbed:

```
reserve_ctl.sh -b <IRON install dir>/IRON/experiments/testbeds -q bbn_testbed.cfg
```

The following is observed when all enclaves are available:

All enclaves available.

The following is observed when some of the testbed enclaves are reserved:

```
enclave1:  
Locked by: sgriffin  
Reserved on: Tue Oct 30 10:18:11 EDT 2018  
  
enclave2:  
Locked by: sgriffin  
Reserved on: Tue Oct 30 10:18:11 EDT 2018  
  
enclave3:  
Locked by: sgriffin  
Reserved on: Tue Oct 30 10:18:11 EDT 2018
```

For all enclaves that are reserved and currently unavailable, the output above shows who "owns" the enclave, user sgriffin, and when it was locked. This information may be useful if enclaves get "stuck" in a reserved state. It provides a reference to who has the locked enclaves so that additional information including intended duration of usage can be queried.

Note that the operation of the testbed reservation system is fully integrated into the experiment execution process, described in the "Experiment Execution" section below. The *reserve_ctl.sh* script is generally only run manually to query the state of the testbed reservations when not enough testbed nodes are currently available. The *reserve_ctl.sh* script can also be manually run to release testbed nodes that are currently reserved. Note that the user that has the nodes currently locked must execute the script to release them.

13.9 Experiment Execution

13.9.1 Experiment Overview

Typically, a GNAT experiment is executed on the machine where the GNAT distribution is installed. This is usually **not** a testbed machine, but a machine that is able to reach all of the testbed machines. When an experiment runs, a number of the following steps may occur depending on which command-line options are provided to the automated experiment execution script:

1. The GNAT executables are built on the local machine.
2. The experiment is staged on the local machine in the **/home/<user>/iron_exp_staging** directory, referred to as the "local staging directory".
3. The experiment is configured in the local staging directory.
4. The experiment is installed on the remote testbed machines (into the *exp_base_dir* defined in the Testbed Topology File)

5. The experiment is executed on the remote testbed machines.
6. The experiment results are post-processed on the remote testbed machines.
7. The experiment results are collected and placed on the local machine in the **/home/<user>/iron_results** directory.

The details of running an experiment with the automated experiment execution scripts and viewing experiment results are described in the following sections.

13.9.2 Running an Experiment

Once your environment is set up, the `run_exp.sh` script, located in the `<IRON install dir>/IRON/experiments/scripts` directory, is used to setup and run experiments. A full description of the script and all of its options and arguments is obtained by running the following command:

`run_exp.sh -h`

A more detailed description of the most commonly included `run_exp.sh` options and arguments is provided below.

- The make option, `-m`, builds the GNAT executables. This is a local operation and does not interact with any testbed nodes.
- The stage option, `-s`, creates a local staging area for the experiments. Additionally, it creates a tarball that is used during the installation phase. If the `remote_execution_node` is set in the Testbed Topology File, then the experiment tarball is copied to the remote node and untarred there. All remaining commands will run on this remote node.
- The configure option, `-c`, generates the experiment configuration files for a selected testbed topology. The configure command reads in the testbed topology file and the `exp.cfg` file for each experiment and expands the generic node names in the experiment configuration to fully qualified node names. It also interacts with the testbed nodes to dynamically determine the interface names to use during the experiment. The result of this command is a modified version of the `exp.cfg` file that is placed in the appropriate experiment directory in the staging area on the local machine, `${HOME}/iron_exp_staging/`. Additionally, this step of the process specializes any parameterized configuration files in the experiment cfgs directories and creates the appropriate number of experiment run directories for each experiment.
- The install option, `-i`, installs the GNAT executables, test execution scripts, and the experiment-specific configuration files on the testbed nodes for all of the experiments to be run.
- The run option, `-r`, starts the experiments. Note that this results in possibly multiple runs for an experiment when the `params.txt` file is used. Additionally, multiple experiments may be provided to the script. The requirement for running multiple experiments is that each experiment that is to be run has to operate on the single testbed topology configuration. When each experiment terminates, the test components are stopped and the experiment results are collected and placed in experiment run specific directories.
- The process option, `-p`, generates plots of throughput, instantaneous queues, and instantaneous send rate in the appropriate `experiment/runX/nodeY/results/` directory for each GNAT and application node.
- The node reservation option, `-l` or `-L`. The `-l` option is a colon separated list of the enclaves that are to be reserved, e.g., `1:2:3` indicates that the experimenter wishes to use En-

claves 1, 2, and 3. If any of the enclaves in the list are currently locked, the script terminates with an error code. The -L option indicates the number of enclaves that are required for the experiment. If the number of requested enclaves are not available for reservation, the script terminates with an error code. When the requested list or number of enclaves have been reserved, the file *enclaves.cfg* is created and placed in the ***\${HOME}/iron_exp_staging/*** directory. This file indicates the real enclaves that are to be used for the experiment.

- The reservation host option, -o, identifies the testbed reservation system host. If the experiment testbed does not have a reservation server configured, the experimenter can specify 'none' as the reservation host when the -o option is provided on the command-line, indicating that reservation of testbed nodes will not be attempted. Additionally, if the experiment is running on the ISI DeterLab testbed, reservation of testbed nodes will also not be attempted. Note that if there is no reservation server, care must be taken to ensure that only a single experiment utilize any given Enclave in the testbed, as the experiment execution scripts can no longer enforce this.
- The user option , -u <user_name>, identifies the name of the user that the experiment is to be run as. Typically, this is set to *iron* for experiments run on the local testbeds and it is set to the experimenter's user name for ISI DeterLab experiments. The user must have sudo privileges on the experiment nodes, preferably without prompting for passwords.
- The testbed configuration file option, -t <testbed_topo_file>, identifies the name of the testbed topology file, which contains the mapping of generic node names to real hostnames and associates IP Addresses with generic link names. These files are located in the ***IRON/experiments/testbeds/*** directory. Note that this topology does not need to match the topology of the experiment created for DeterLab, but the nodes/links in the DeterLab experiment must be a subset of the nodes/links in this topology file.
- The arguments to the script are the names of the experiments to run. As described earlier, more than one experiment may be run provided that all can be run on the same testbed topology.

The following command is used to build the executables, stage, configure, install, run, and process our example experiment where the experimenter desires that real physical Enclaves 1, 2, and 3 be used for the experiment:

`run_exp.sh -mscirl -l 1:2:3 -u iron -t bbn_testbed.cfg 3enclave_example_exp`

If the experimenter does not care which physical enclaves are used, the following command can be used:

`run_exp.sh -mscirl -L 3 -u iron -t bbn_testbed.cfg 3enclave_example_exp`

If there is no reservation server configured in the testbed, our experiment can be run with the following command:

`run_exp.sh -mscirl -u iron -t bbn_testbed.cfg -o none 3enclave_example_exp`

Recall that care must be taken to deconflict testbed assets for the above command as the deconfliction can no longer be enforced by the experiment execution scripts.

13.9.2.1 Generated Configuration Files

A number of experiment run directories may be created when an experiment is configured using the *run_exp.sh* script described above. The run1 directory is always created, but up to N run directories may be created. The number of directories is determined by the number of values that are specified for the parameterized substitution tags in the *params.txt* input file. Recall that our example experiment has 3 values for the **NUM_HI_PRIO_TCP_FLOWS** substitution tag. This results in the following directory structure being created for our example experiment:

Directory Structure Following Experiment Configuration

```
/home/<user>/iron_exp_staging/
└── 3enclave_example_exp
    ├── cfgs
    ├── logs
    ├── pcaps
    └── run1
        ├── cfgs
        ├── logs
        └── pcaps
    └── run2
        ├── cfgs
        ├── logs
        └── pcaps
    └── run3
        ├── cfgs
        ├── logs
        └── pcaps
└── bin
└── scripts
└── testbeds
```

The information in the **3enclave_example_exp/cfgs/** directory is only used to configure the experiment. When the example experiment is configured, a script is run to perform the parameter substitutions in the configuration files. The information in the **3enclave_example_exp/cfgs/** directory creates the **3enclave_example_exp/run1/**, **3enclave_example_exp/run2/**, and **3enclave_example_exp/run3/** directories. The configuration files in each of these directories have been tailored per the entries in the *params.txt* file. Each experiment run will use the information contained in the corresponding runX directory. Also note that the executables, the scripts, and the testbed file for the experiment are also staged. This provides a record of the experiment that is run.

13.9.2.2 Experiment Results

Following the execution of an experiment, the experiment artifacts will, by default, be collected and copied to the **results_location** as specified in the experiment's testbed topology file. The following shows the directory structure for run1 of our example experiment. There will be similar directories for run2 and run3 of our example experiment.

Experiment Result Directory Structure

```
/home/<user>/iron_results/<date_time>/
  └── 3enclave_example_exp
      └── run1
          ├── enclave1
          │   ├── app1
          │   │   ├── cfgs
          │   │   ├── logs
          │   │   ├── pcaps
          │   │   └── results
          │   ├── iron
          │   │   ├── cfgs
          │   │   ├── logs
          │   │   ├── pcaps
          │   │   └── results
          │   └── lel
          │       ├── cfgs
          │       ├── logs
          │       └── pcaps
          └── enclave2
              ├── app1
              │   ├── cfgs
              │   ├── logs
              │   ├── pcaps
              │   └── results
              ├── iron
              │   ├── cfgs
              │   ├── logs
              │   ├── pcaps
              │   └── results
              └── lel
                  ├── cfgs
                  ├── logs
                  └── pcaps
      └── enclave3
          ├── app1
          │   ├── cfgs
          │   ├── logs
          │   ├── pcaps
          │   └── results
          ├── iron
          │   ├── cfgs
          │   ├── logs
          │   ├── pcaps
          │   └── results
          └── lel
              ├── cfgs
              ├── logs
              └── pcaps
  └── bin
```

We see that at the top-level of the experiment's results directory is a **bin/** directory that contains a copy of all of the binaries that were used for the experiment as well as status information from the GIT repository from which the experiment was run, if available. The GIT information contains the latest commit information and includes any local changes in the experimenter's GIT sandbox included in the experiment. This information is sufficient to recreate the experiment at a later date, if desired.

The above directory structure is organized by "notional enclaves" for each of the runs. We see **enclave1/**, **enclave2/**, and **enclave3/** directories which contain the configuration information, log files, packet captures, and results for the application (**app1/**), IRON (**iron/**), and link emulation (**le1/**) nodes for the respective enclaves. For simplicity, only the **run1/** directory is shown above. There would be similar directories for **run2/** and **run3/** for our example experiment. If the experiment was dual homed, there would also be a **le2/** directory for each enclave directory. Similarly, if there were more than one application node per enclave, there would be additional **appX/** directories.

If, for any reason, the experimenter needs to visit the physical node for any of the components that ran during the experiment, the *node_to_enclave_map.txt* file, located in the staging directory, can be viewed. This file is used by the automated experiment execution scripts and contains information of the following form:

generic_node_name hostname results_directory_name

The following depicts the entries for our experiment when run on real physical Enclaves 1, 2, and 3:

```
node1 gnat-app1 enclave1/app1
node2 gnat1 enclave1/iron
node3 gnat-le1 enclave1/le1
node4 gnat-le1 enclave1/le2
node5 gnat-app2 enclave2/app1
node6 gnat2 enclave2/iron
node7 gnat-le2 enclave2/le1
node8 gnat-le2 enclave2/le2
node9 gnat-app3 enclave3/app1
node10 gnat3 enclave3/iron
node11 gnat-le3 enclave3/le1
node12 gnat-le3 enclave3/le2
```

So, if we need to track down an issue with the source of our flows, we know we should visit **gnat-app1**.

13.9.2.3 Viewing Results

If the **-p** flag is provided to the *run_exp.sh* script, a set of post-processing steps, outlined in the *process.cfg* configuration file, are executed. In our example experiment, the post-processing steps include the generation of goodput, latency, and loss plots from the log files collected on the destination node. Recall that the flow definitions for our example experiment are from source *\$enclave1_app1_node\$* to *\$enclave2_app1_node\$*. To view the goodput plot at the destination (as a PNG image file), simply do the following:

```
cd /home/<user>/iron_results/<date_time>/3enclave_example_exp/run1/en-
```

```
clave1/app1/results
```

```
eog mgen_goodput.png
```

13.10 Common Debugging Techniques

If the generated experiment results do not look as expected, it may be necessary to re-run the experiment after reconfiguring it to generate additional debugging information to aid in figuring out what is going on. Two common modifications include changing the log levels to generate additional diagnostic output from the GNAT components and enabling packet captures in various places in the GNAT network.

13.10.1 Experiment Logging

The log level of the core GNAT components, BPF (BackPressure Forwarder), TCP Proxy, and UDP Proxy, is controlled by the configuration item *Log.DefaultLevel* found in the *bpf_common.cfg*, *tcp_proxy_common.cfg*, and *udp_proxy_common.cfg*, respectively. The format of the value for the configuration item is any combination of the letters F (Fatal), E (Error), W (Warning), I (Information), A (Analysis), and D (Debug). By default, the log level is set to "FEW". Note that increasing the log levels can lead to very large log files as some of the diagnostic output is per packet.

If turning up the log levels generates too much logging output, each of the components can be configured to support increased logging on a finer-grained scale, by enabling and configuring class level logging. The configuration item *Log.ClassLevels* in the *bpf_common.cfg*, *tcp_proxy_common.cfg*, and *udp_proxy_common.cfg* files (found in the experiment configuration directory) accomplishes this. The format for specifying this configurable item is as follows:

Log.ClassLevels ClassName1=LogLevel1;ClassName2=LogLevel2;...;ClassNameN=LogLevelN

The following entry in the *tcp_proxy_common.cfg* configuration file configures the Socket class to log at levels FEWI and the SendBuffer class to log at levels FEWIAD:

Log.ClassLevels Socket=FEWI;SendBuffer=FEWIAD

The class level logging is generally useful during the development stage when turning up the log levels component-wide generates so much logging output that it is difficult to understand.

13.10.2 Experiment Packet Captures

Another useful debugging technique is looking at packet captures. The *exp.tpl* file that was generated when we created our experiment is where we identify where, in the GNAT network, we want the packet captures to occur. The process of creatig this file is fully described in the *Creating A New Experiment* section. The commented-out *PCAPS* and *DECAP* lines is where we specify this information. To avoid possible performance implications, we typically do not do packet captures on the GNAT nodes. This leaves the application and link emulation nodes in our example experiment as available places to capture packets. The application node has a WAN-facing interface and the link emulation node has a LAN-facing (toward the application) and a WAN-facing (toward a remote enclave) interface. To identify where packets are capture, we

need to identify the node and link in "configuration template replacement" string notation. This leads to the following possibilities for our packet captures, entered in the *PCAPS* entry in the *exp.tpl* configuration template:

- *\$enclaveX_app1_node\$:\$enclaveX_app1_wan_link\$*: The WAN-facing link on the application node in enclave X
- *\$enclaveX_le1_node\$:\$enclaveX_le1_lan_link\$*: The LAN-facing link on the link emulation node in enclave X.
- *\$enclaveX_le1_node\$:\$enclaveX_le1_wan_link\$*: The WAN-facing link on the link emulation node in enclave X.

The following entry instructs the automated experiment execution scripts to capture packets on the LAN-facing interface of the link emulation node in enclave 1:

PCAPS=(\$enclave1_le1_node\$:\$enclave1_le1_lan_link\$)

Note that the packets captured on the link emulation nodes are GNAT CAT (Capacity Adaptive Tunnel) encapsulated packets. The *sliqdecap* utility can be used to decapsulate the packet capture after the experiment completes and the artifacts are collected and placed in the results directory previously described. The automated experiment execution scripts can also automatically decapsulate the packet captures on the link emulation nodes prior to the data collection phase. This is controlled by configuring the *DECAP* entry in the *exp.tpl* configuration template. To identify which packet captures are to be decapsulated, we need to identify the node, link, and type of decapsulation in configuration template replacement string notation. Assuming that we capture packets on the LAN-facing interface of the link emulation node, the following line will instruct the automated experiment execution scripts to decapsulate it for us:

DECAP=(\$enclave1_le1_node\$:\$enclave1_le1_lan_link\$:sliq)

Note that the *:sliq* portion of the above example instructs the automated experiment execution scripts to use the *sliqdecap* utility to decapsulate the packet capture. The end result of doing this is a packet capture in which the GNAT CAT headers have been removed, leaving the original application flows in the resulting packet capture file.

14 Installing and Configuring PIM-SM on a GNAT Testbed Host

Within our GNAT testbed, we use PIM-SM (Protocol Independent Multicast - Sparse Mode) as a multicast routing protocol. It primarily provides the mechanism for getting multicast packets from application hosts into GNAT nodes, and multicast packets from GNAT nodes back to application hosts (i.e., the GNAT nodes must accept and respond to IGMP messages from application hosts in order for multicast packets to be forwarded into and out of GNAT nodes). An important secondary reason is adding the ability to compare GNAT performance against a baseline multicast-enabled network.

Within our testbed, PIM-SM daemon is configured to run over gre tunnels, mimicking what would be needed if multicast is not enabled on each and every router between GNAT nodes in a larger network.

14.1 Step-by-step guide

To add the host "gnatX" as a new PIM router within our testbed with gnat0 as a PIM-SM rendezvous point:

1. On gnatX, add a gre tunnel called *grermp0* from gnatX to gnat0 by adding the following to */etc/network/interfaces* (making appropriate substitutions for "X"):
 2. auto grermp0
 3. iface grermp0 inet static
 4. address 10.100.X.2/30
 5. pre-up ip tunnel add grermp0 mode gre local 10.X.1.2 remote 172.24.0.254 ttl 255
 6. up ifconfig grermp0 multicast
 7. up route add -net 10.100.0.0/16 gw 10.100.X.1 dev grermp0
 8. up echo 0 > /proc/sys/net/ipv4/conf/grermp0/rp_filter post-down iptunnel del grermp0
9. On gnatX, turn off reverse path filtering by adding the following to */etc/sysctl.d/99-sysctl.conf*:
 10. net.ipv4.conf.all.rp_filter = 0
 11. net.ipv4.conf.default.rp_filter = 0
11. Load the new configuration on gnatX by executing the following command:

sudo sysctl -p

12. On gnat0, add a gre tunnel called *grermpX* from gnat0 back to gnatX by adding the following to */etc/network/interfaces* (making appropriate substitutions for "X"):
 13. auto grermpX
 14. iface grermpX inet static
 15. address 10.100.X.1/30
 16. pre-up ip tunnel add grermp6 mode gre local 172.24.0.254 remote 10.X.1.2 ttl 255
 17. up ifconfig grermpX multicast
 18. up echo 0 > /proc/sys/net/ipv4/conf/grermpX/rp_filter post-down iptunnel del grermpX

19. On gnat0, edit the configuration file `/etc/pimd.conf` to allow pimd to include grermpX as one of its allowed interfaces. This should immediately follow the existing list of interfaces already enabled in this file.

```
phyint grermpX enable
```

20. On gnatX, tell the running pimd daemon to reload its configuration information (the -l below is a lower case "L")

```
sudo pimd -l
```

21. On gnatX, install pimd via

```
sudo apt-get install pimd
```

22. On gnatX, edit the startup file `/etc/systemd/system/multi-user.target.wants/pimd.service` to disable pimd on all interfaces by default. The edit will change the line from this

```
ExecStart=/usr/sbin/pimd -f -c $CONFIG_FILE $DAEMON_ARGS
```

to this

```
ExecStart=/usr/sbin/pimd -f -N -c $CONFIG_FILE $DAEMON_ARGS
```

23. On gnatX, edit the configuration file `/etc/pimd.conf` to allow pimd to include both the interface to gnat-appX, and the newly created gre tunnel grermp0:

24. # By default, all non-loopback multicast capable interfaces are enabled.

25. # If you want to use loopback, set the interface multicast flag on it.

26. phyint eno2 enable
phyint grermp0 enable

27. On gnatX, tell the system to reload its service definitions

```
sudo systemctl daemon-reload
```

28. On gnatX, restart the pimd service

```
sudo service pimd restart
```

14.2 Configuration Note

Originally, we configured the application subnet in the GNAT testbed enclave to be a proper subnet of the associated GNAT node. As an example, the application-facing interface on gnat1 was configured as

10.1.1.201/26

and the cloud-facing interface on gnat1 was configured as

10.1.1.1/24

This configuration confused pimd (pimd did not recognize these as 2 interfaces that multicast packets must be routed between) and caused pimd to ignore the application-facing interface. This resulted in ignoring the IGMP group join packets generated by the application node in the enclave. Once reconfigured in the following fashion

10.1.3.1/24 (application-facing interface address)

10.1.1.1/24 (cloud-facing interface address)

IGMP group join packets were received and multicast packets were successfully received.

14.3 Testing Installation

A test multicast application (sender and receiver) has been installed on the the application hosts in the GNAT testbed, gnat-appX (where X is 1-12 and 19-24), in the */home/iron/multicast* directory. This application can be use to test out our pimd installation. One application node acts as the source of the multicast packet and any number of application nodes act as the destination of the multicast packet. For each destination node do the following:

ssh gnat-appX (as the iron user)

cd multicast

./mcast_recv 10.X.3.2 (Note: the provided address is the address of the interface on which multicast packets are to be received)

For the source node do the following:

ssh gnat-appX (as the iron user)

cd multicast

./mcast_send 10.X.3.2 (Note: the provided address is the address of the interface on which the multicast packet is to be sent)

The following is observed at each of the destination application nodes when the **mcast_send** process is executed on gnat-app6:

```
Received: 62 6f 62 0
From: 10.6.3.2
```

The **mcast_send** application sends the string "bob" to the multicast group 227.9.18.27. Each of the **mcast_recv** applications subscribe to the 227.9.18.27 multicast group. On receiving the multicast packet the bytes are printed out in hex (line 1 above) and the source address of the multicast packet is the WAN-facing address of gnat-app6 (line 2 above).

15 Setting Up and Running USC's Jupiter

This HOWTO describes the steps taken to get the USC Jupiter Orchestrator running on a set of BBN rack mounted Linux servers. The information captures the installation, configuration, and execution instructions for the required components and provides a convenient "cookbook recipe" that can be used to get the Jupiter Orchestrator running on a set of new machines. Throughout the discussion, there are links provided to pages that contain additional details and configuration options for the various components.

15.1 Execution Environment

The following list provides some of the relevant details about the environment in which the Jupiter Orchestrator is run. This is for illustrative purposes only and the instructions can be easily modified to include additional (or fewer) nodes, as desired.

- 6 Rack Mounted Linux servers for kubernetes cluster (1 master node, 5 worker nodes)
 - Operating System: Ubuntu 16.04 LTS
 - Memory: 4GB RAM
- Kubernetes: v1.13.0
- Docker: v 17.03.2-ce

For the purposes of this HOWTO, the kubernetes master node is gnat-app1.bbn.com and the kubernetes worker nodes are gnat-app2.bbn.com, gnat-app3.bbn.com, gnat-app4.bbn.com, gnat-app5.bbn.com, and gnat-app6.bbn.com.

15.2 Clone Jupiter Repository

The following commands can be used to clone the Jupiter Repository:

```
git clone --recurse-submodules https://github.com/ANRGUSC/Jupiter.git
cd Jupiter/
git submodule update -remote
```

15.3 Installing Kubernetes and Required Components

The Jupiter documentation identifies a set of [requirements](#) for running the Jupiter system. The following set of instructions elaborates on the information provided in the Jupiter documentation. The scripts in the following sections must be run as root or with sudo privileges.

15.3.1 Install Kubernetes

The following script installs the kubernetes components:

```
#!/bin/bash

apt-get update && apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
```

```
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF

apt-get update
apt-get install -y kubectl
```

15.3.2 Install pip3

The following script was used to install pip3:

```
#!/bin/bash

apt-get install python3-pip
pip3 install --upgrade pip
```

15.3.3 Install Required Python Packages

The following script installs the Python packages that Jupiter requires:

```
#!/bin/bash

pip3 install -r Jupiter/k8_requirements.txt
```

The Jupiter/k8_requirements.txt file is provided as part of the Jupiter repo. The script, as written, requires that the Jupiter repo be located in the Jupiter directory in the script directory's location.

15.3.4 Install Kubernetes Tools

The following script installs the required kubernetes tools, namely kubelet and kubeadm:

```
#!/bin/bash

apt-get update && apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF

apt-get update
apt-get install -y kubelet kubeadm
```

15.3.5 Install Minikube

The following script install minikube:

```
#!/bin/bash

curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.27.0/minikube-
linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

15.3.6 Docker

15.3.6.1 Install Docker

The following script installs Docker:

```
#!/bin/bash

apt-get update
apt-get install -y docker.io
```

15.3.6.2 Configure Docker to Execute Commands Without sudo

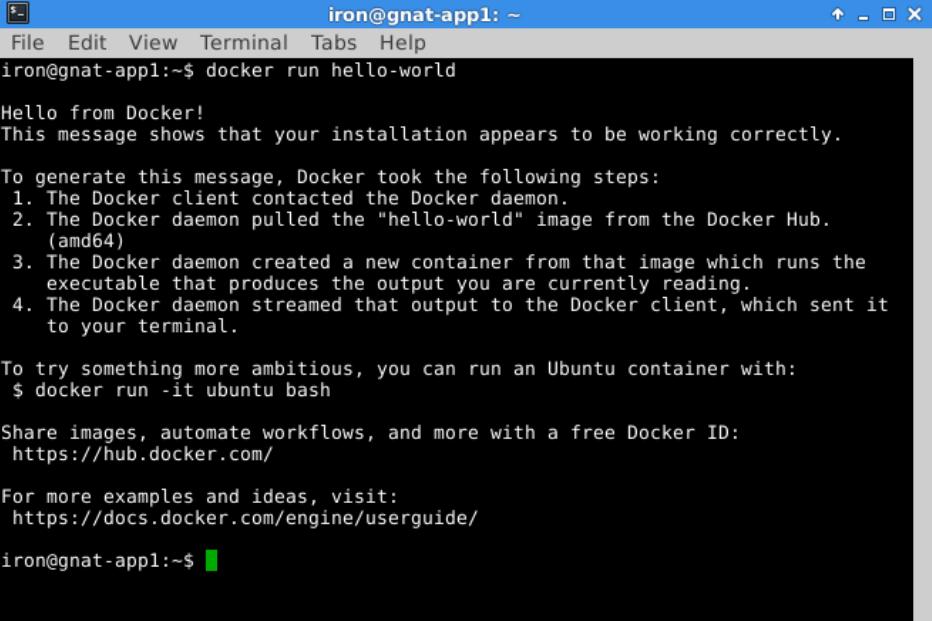
To run the Jupiter Orchestrator, it is necessary to run docker commands without sudo privileges. This can be accomplished by executing the following command:

```
sudo usermod -aG docker <username>
```

for each <username> to be added to the docker group on the testbed nodes. After adding the **iron** user to the docker group, the following command is executed on gnat-app1.bbn.com to test out our docker installation:

```
docker run hello-world
```

The result of executing the above command on gnat-app1.bbn.com is shown below:



A screenshot of a terminal window titled "iron@gnat-app1: ~". The window shows the command "docker run hello-world" being run, followed by the output of the "Hello from Docker!" message. The message includes instructions on how Docker runs the command, how to run an Ubuntu container, and links to the Docker Hub and documentation. The terminal window has a standard Linux-style interface with a menu bar and a scroll bar.

```
iron@gnat-app1:~$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

15.4 Setting Up Kubernetes Cluster

The kubernetes cluster was created using **kubeadm**, as described [here](#). The following sections go into a little more detail and try to capture the information that we accumulated by visiting many different web pages.

Now that all of the required components have been installed and configured, we can set up our kubernetes cluster, composed of a single master node and a set of worker nodes. For illustrative purposes, the master node will be gnat-app1.bbn.com and the worker nodes will be gnat-app2.bbn.com, gnat-app3.bbn.com, gnat-app4.bbn.com, gnat-app5.bbn.com, and gnat-app6.bbn.com. The cluster nodes are set up on bare metal, no virtual machines are used. All commands in this section should be run as root.

To run without error, kubernetes requires that swap be disabled. This is accomplished by executing the following command on all testbed nodes:

```
swapoff -a
```

The nodes that will be part of the kubernetes cluster have 2 physical interfaces, a public network control plane interface and a private network data plane interface. We will configure the cluster so that all cluster and pod communications occur over the private network interface. It just happens that this is not the default route interface and will require some additional command-line options when starting the cluster components, which is described below.

15.4.1 Kubernetes Master Node

15.4.1.1 Initializing Master Node

The first step to create the kubernetes cluster is to initialize the master node. The **kubeadm** executable is utilized to accomplish this. Two command-line options, *--pod-network-cidr* and *--apiserver-advertise-address*, will be provided to **kubeadm** during the initialization process. The *-pod-network-cidr* option is required because [flannel](#) will be used as the pod network add-on, chosen because it is the pod network the Jupiter team uses. The *--apiserver-advertise-address* option is provided to ensure that the cluster communications occur over the private network interface and not the default public network interface.

See [here](#) for a more complete description of the **kubeadm init** command.

The following command is used to initialize the kubernetes master node:

```
kubeadm --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=10.1.3.2 init
```

The 10.244.0.0/16 address is the address that **must** be provided when flannel is used as the pod network (see [flannel](#) documentation) and the 10.1.3.2 address is the interface address of the private network on the master node.

The results of running the above command to initialize the kubernetes master node is depicted below:



A screenshot of a terminal window titled "root@gnat-appl: /root". The window displays the output of the "kubeadm init" command. The output shows the initialization process, including pulling images, generating certificates, writing configuration files, and creating static Pod manifests for various components like kube-apiserver, kube-controller-manager, and kube-scheduler. It also marks the node as a control-plane and applies essential addons like CoreDNS and kube-proxy. The terminal ends with instructions to deploy a pod network and join other nodes.

```
root@gnat-appl:/root# kubeadm --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=10.1.3.2 init
[init] Using Kubernetes version: v1.13.1
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Activating the kubelet service
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [gnat-appl.bbn.com kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 10.1.3.2]
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [gnat-appl.bbn.com localhost] and IPs [10.1.3.2 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [gnat-appl.bbn.com localhost] and IPs [10.1.3.2 127.0.0.1 ::1]
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from directory "/etc/kubernetes/manifests". This can take up to 4m0s
[apiclient] All control plane components are healthy after 30.503061 seconds
[uploadconfig] storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config-1.13" in namespace kube-system with the configuration for the kubelets in the cluster
[patchnode] Uploading the CRI Socket information "/var/run/dockershim.sock" to the Node API object "gnat-appl.bbn.com" as an annotation
[mark-control-plane] Marking the node gnat-appl.bbn.com as control-plane by adding the label "node-role.kubernetes.io/master=''"
[mark-control-plane] Marking the node gnat-appl.bbn.com as control-plane by adding the taints [node-role.kubernetes.io/master:NoSchedule]
[bootstrap-token] Using token: ayr8qz.sfu2sgcbjtzf6wxw
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstrap-token] configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for nodes to get long term certificate credentials
[bootstrap-token] configured RBAC rules to allow the csrapprover controller automatically approve CSRs from a Node Bootstrap Token
[bootstrap-token] configured RBAC rules to allow certificate rotation for all node client certificates in the cluster
[bootstrap-token] creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

    kubeadm join 10.1.3.2:6443 --token ayr8qz.sfu2sgcbjtzf6wxw --discovery-token-ca-cert-hash sha256:e73f6600257aa7bec593808a94897de93fc27ce0c31ca4fala7bc810a2a9b7db

root@gnat-appl:/root#
```

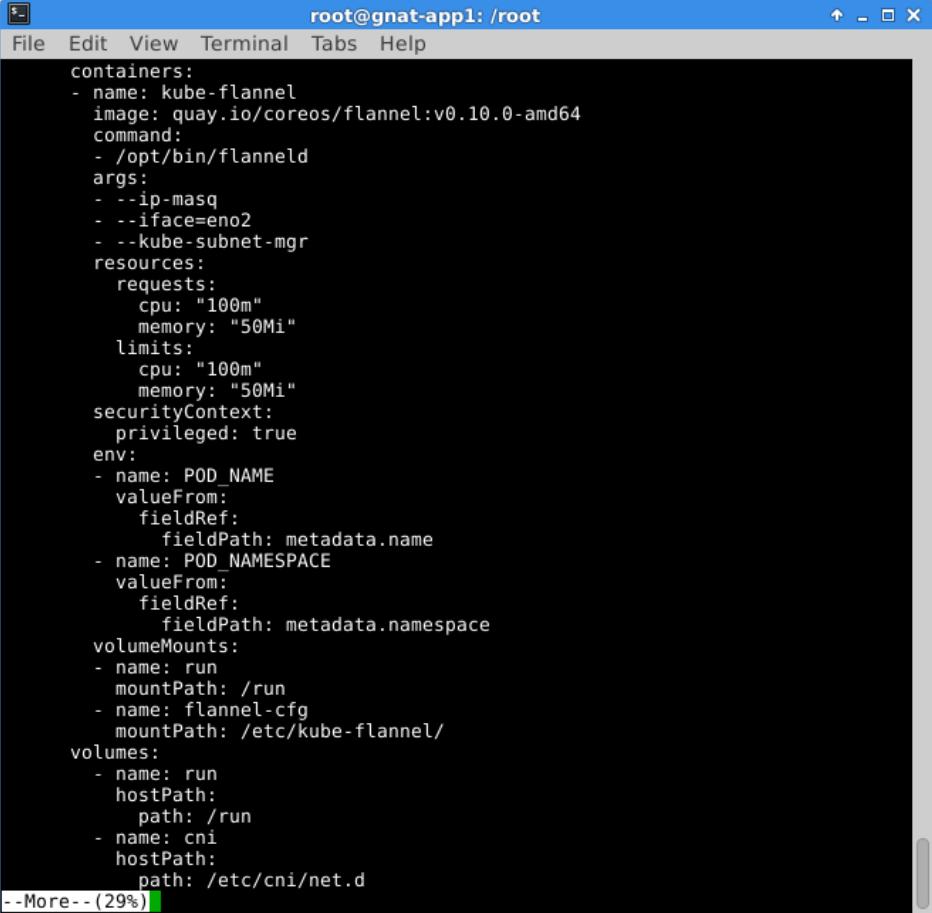
The last line of output from the *kubeadm init* command will be used when the worker nodes join the cluster.

15.4.1.2 Pod Network

As mentioned earlier, a pod network add-on must be installed so the pods can communicate. The default behavior for flannel is to use the default route interface for pod-to-pod communications. Since we want this to occur over the private network interface, we must modify the flannel configuration. First, get the flannel configuration file using the following command:

```
 wget raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

Next, add the required "--iface=XXX" argument to the "args" section of the "kube-flannel" container of the kube-flannel.yml configuration file to indicate that the pods communications are to occur over the private network interface. For our testbed nodes, the private network interface is the **eno2** interface. The modified flannel configuration is illustrated below:

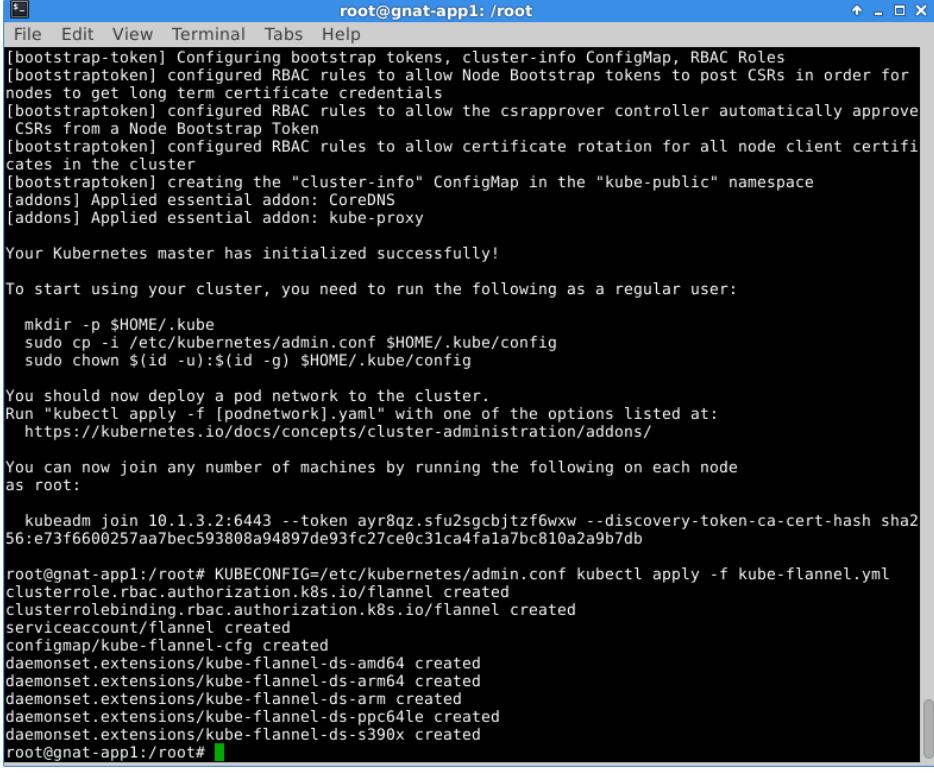


```
root@gnat-app1: /root
File Edit View Terminal Tabs Help
containers:
- name: kube-flannel
  image: quay.io/coreos/flannel:v0.10.0-amd64
  command:
  - /opt/bin/flanneld
  args:
  - --ip-masq
  - --iface=eno2
  - --kube-subnet-mgr
  resources:
    requests:
      cpu: "100m"
      memory: "50Mi"
    limits:
      cpu: "100m"
      memory: "50Mi"
  securityContext:
    privileged: true
  env:
  - name: POD_NAME
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
  volumeMounts:
  - name: run
    mountPath: /run
  - name: flannel-cfg
    mountPath: /etc/kube-flannel/
  volumes:
  - name: run
    hostPath:
      path: /run
  - name: cni
    hostPath:
      path: /etc/cni/net.d
--More-- (29%)
```

We can now install the pod network with the following command:

```
KUBECONFIG=/etc/kubernetes/admin.conf kubectl apply -f kube-flannel.yml
```

The result from running the above command is shown below:



A screenshot of a terminal window titled "root@gnat-appl: /root". The window displays the output of the kubeadm init command. It shows the configuration of bootstrap tokens, RBAC rules for Node Bootstrap tokens, and the creation of ConfigMaps for cluster-info and kube-public namespaces. It also lists applied addons like CoreDNS and kube-proxy. A message indicates that the Kubernetes master has initialized successfully. It provides instructions for starting the cluster as a regular user, including commands to copy the admin.conf file to \$HOME/.kube/config and change ownership. It also suggests deploying a pod network using "kubectl apply -f [podnetwork].yaml" and joining machines to the cluster as root using "kubeadm join". The terminal ends with a root prompt at the bottom.

```
[bootstraptoken] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstraptoken] configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for
nodes to get long term certificate credentials
[bootstraptoken] configured RBAC rules to allow the csrapprover controller automatically approve
CSRs from a Node Bootstrap Token
[bootstraptoken] configured RBAC rules to allow certificate rotation for all node client certifi
cates in the cluster
[bootstraptoken] creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

    kubeadm join 10.1.3.2:6443 --token ayr8qz.sfu2sgcbjtzf6wxw --discovery-token-ca-cert-hash sha2
56:e73f6600257aa7bec593808a94897de93fc27ce0c31ca4fa1a7bc810a2a9b7db

root@gnat-appl:/root# KUBECONFIG=/etc/kubernetes/admin.conf kubectl apply -f kube-flannel.yml
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.extensions/kube-flannel-ds-amd64 created
daemonset.extensions/kube-flannel-ds-arm64 created
daemonset.extensions/kube-flannel-ds-arm created
daemonset.extensions/kube-flannel-ds-ppc64le created
daemonset.extensions/kube-flannel-ds-s390x created
root@gnat-appl:/root#
```

15.4.1.3 Controlling Kubernetes Cluster as Non-Root User

Once the master node has been initialized and the pod network has been loaded, set up the local user's environment so the cluster can be controlled without the *sudo* command. The following commands are used to establish this:

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
sudo cp -i /etc/kubernetes/admin.conf $HOME
sudo chown $(id -u):$(id -g) $HOME/admin.conf
```

Note: lines 3 and 4 are required for the Jupiter deployment scripts to execute properly.

15.4.1.4 Start Kubernetes Proxy

The Jupiter Orchestrator requires a working kubernetes cluster with proxy capability, so we need to start the proxy next. This is typically done in a separate terminal as the local user and is accomplished with the following **kubectl** command:

```
kubectl proxy -p 8080
```

15.4.1.5 Master Isolation

By default, Kubernetes will not schedule pods for the master node. The Jupiter Orchestrator, however, schedules pods on the master node. This is enabled by executing the following command on the master node:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

15.4.1.6 Verifying Master Initialization

Now that we have a master node initialized and a pod network installed, we can view the current state of our cluster using the **kubectl get** command. The following command can be used to continuously monitor, using the Linux **watch** command, the nodes and pods in our cluster:

```
watch "kubectl get nodes && echo \"\" && echo \"\" && kubectl get pods --all-namespaces"
```

The result of executing the above command is shown below:

```
iron@gnat-app1: ~
File Edit View Terminal Tabs Help
Every 2.0s: kubectl get nodes && echo "" && echo "" && kubectl get pods...  Thu Dec 13 09:28:25 2018
NAME          STATUS    ROLES      AGE       VERSION
gnat-app1.bbn.com   Ready     master    9m17s    v1.13.0

NAMESPACE     NAME           READY   STATUS    RESTARTS   AGE
kube-system   coredns-86c58d9df4-59d28   1/1     Running   0          9m3s
kube-system   coredns-86c58d9df4-tl2vd   1/1     Running   0          9m3s
kube-system   etcd-gnat-app1.bbn.com    1/1     Running   0          8m37s
kube-system   kube-apiserver-gnat-app1.bbn.com  1/1     Running   0          8m31s
kube-system   kube-controller-manager-gnat-app1.bbn.com  1/1     Running   0          8m13s
kube-system   kube-flannel-ds-amd64-4n4p2   1/1     Running   0          5m19s
kube-system   kube-proxy-8t4db        1/1     Running   0          9m3s
kube-system   kube-scheduler-gnat-app1.bbn.com  1/1     Running   0          8m19s
```

We can see that we currently have 1 node in our cluster, the master node, gnat-app1.bbn.com and that there are a number of pods running in the **kube-system** namespace. Now that we have a working master node, we can add the worker nodes to our cluster.

15.4.2 Kubernetes Worker Nodes

15.4.2.1 Join Kubernetes Worker Nodes

To join the worker nodes to the kubernetes cluster, execute the following command, as **root**, in a terminal on each of the worker nodes (namely, gnat-app2, gnat-app3, gnat-app4, gnat-app5, and gnat-app6):

```
kubeadm join 10.1.3.2:6443 --token ayr8qz.sfu2sgcbjtzf6wxw --discovery-token-ca-cert-hash sha256:e73f6600257aa7bec593808a94897de93fc27ce0c31ca4fa1a7bc810a2a9b7db
```

Note that the above command is provided at the end of the master node **kube init** command that was executed to initialize the master node. The result of executing the above command is shown below for node gnat-app2.bbn.com:

```
root@gnat-app2:~# kubeadm join 10.1.3.2:6443 --token ayr8qz.sfu2sgcbjtzf6wxw --discovery-token-ca-cert-hash sha256:e73f6600257aa7bec593808a94897de93fc27ce0c31ca4fa1a7bc810a2a9b7db
[preflight] Running pre-flight checks
[discovery] Trying to connect to API Server "10.1.3.2:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://10.1.3.2:6443"
[discovery] Requesting info from "https://10.1.3.2:6443" again to validate TLS against the pinned public key
[discovery] Cluster info signature and contents are valid and TLS certificate validates against pinned roots, will use API Server "10.1.3.2:6443"
[discovery] Successfully established connection with API Server "10.1.3.2:6443"
[join] Reading configuration from the cluster...
[join] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[kubelet] Downloading configuration for the kubelet from the "kubelet-config-1.13" ConfigMap in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Activating the kubelet service
[tlsbootstrap] Waiting for the kubelet to perform the TLS Bootstrap...
[patchnode] Uploading the CRI Socket information "/var/run/dockershim.sock" to the Node API object "gnat-app2.bbn.com" as an annotation

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

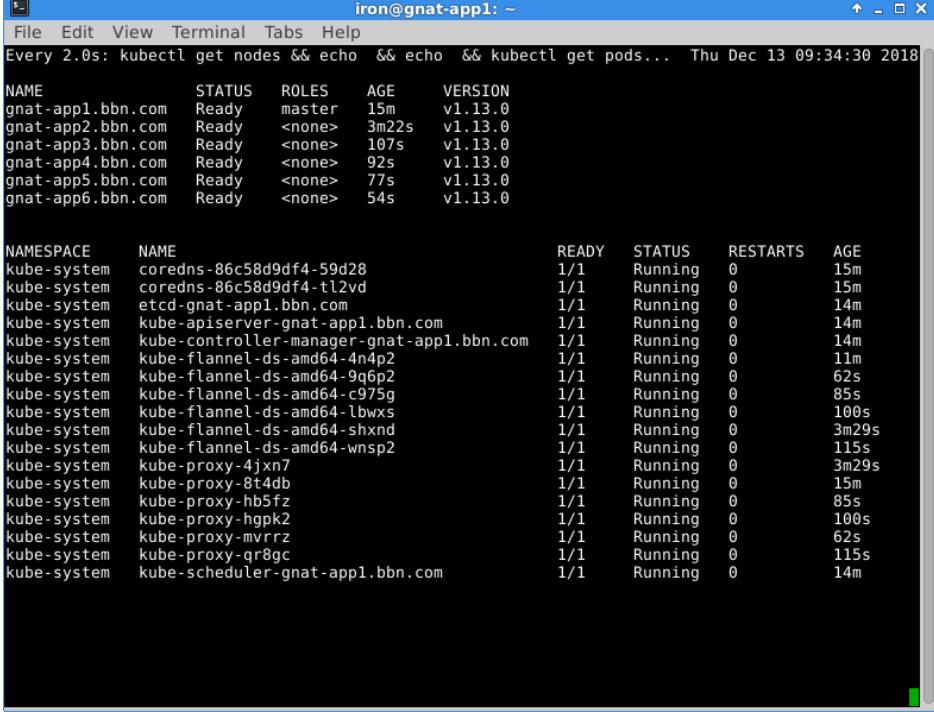
Run 'kubectl get nodes' on the master to see this node join the cluster.

root@gnat-app2:~#
```

Similar output can be observed on each of the other worker nodes.

15.4.2.2 Verifying Worker Nodes Cluster Joins

Now that we have joined all of the worker nodes, we can once again look at the output from the continuous monitoring of the cluster state, first described in [verifying master initialization](#). The output from this monitoring is shown below:



```
iron@gnat-app1: ~
File Edit View Terminal Tabs Help
Every 2.0s: kubectl get nodes && echo && echo && kubectl get pods... Thu Dec 13 09:34:30 2018

NAME           STATUS  ROLES   AGE    VERSION
gnat-app1.bbn.com  Ready  master  15m   v1.13.0
gnat-app2.bbn.com  Ready  <none>  3m22s  v1.13.0
gnat-app3.bbn.com  Ready  <none>  107s  v1.13.0
gnat-app4.bbn.com  Ready  <none>  92s   v1.13.0
gnat-app5.bbn.com  Ready  <none>  77s   v1.13.0
gnat-app6.bbn.com  Ready  <none>  54s   v1.13.0

NAMESPACE      NAME          READY  STATUS    RESTARTS  AGE
kube-system    coredns-86c58d9df4-59d28  1/1   Running  0          15m
kube-system    coredns-86c58d9df4-tl2vd  1/1   Running  0          15m
kube-system    etcd-gnat-app1.bbn.com  1/1   Running  0          14m
kube-system    kube-apiserver-gnat-app1.bbn.com  1/1   Running  0          14m
kube-system    kube-controller-manager-gnat-app1.bbn.com  1/1   Running  0          14m
kube-system    kube-flannel-ds-amd64-4n4p2  1/1   Running  0          11m
kube-system    kube-flannel-ds-amd64-9q6p2  1/1   Running  0          62s
kube-system    kube-flannel-ds-amd64-c975g  1/1   Running  0          85s
kube-system    kube-flannel-ds-amd64-lbwxs  1/1   Running  0          100s
kube-system    kube-flannel-ds-amd64-shxnd  1/1   Running  0          3m29s
kube-system    kube-flannel-ds-amd64-wnsp2  1/1   Running  0          115s
kube-system    kube-proxy-4jxn7   1/1   Running  0          3m29s
kube-system    kube-proxy-8t4db   1/1   Running  0          15m
kube-system    kube-proxy-hb5fz   1/1   Running  0          85s
kube-system    kube-proxy-hgpk2   1/1   Running  0          100s
kube-system    kube-proxy-mvrrz   1/1   Running  0          62s
kube-system    kube-proxy-qr8gc   1/1   Running  0          115s
kube-system    kube-scheduler-gnat-app1.bbn.com  1/1   Running  0          14m
```

We can now see that we have 1 master node, gnat-app1.bbn.com, and 5 worker nodes, gnat-app2.bbn.com, gnat-app3.bbn.com, gnat-app4.bbn.com, gnat-app5.bbn.com, and gnat-app6.bbn.com, as desired. We also have additional pods running, which are associated with the added worker nodes. Now that we have a working kubernetes cluster, we can configure and deploy the Jupiter Orchestrator.

15.5 Setup Local Private Docker Registry

15.5.1 Registry Configuration and Execution

We set up a local private Docker registry for the Jupiter pods so that we didn't have to push them to the publicly available [Docker Hub](#). The instructions for deploying a Docker registry server can be found [here](#). To run our experiments, we set up an insecure registry. Note that Docker must be installed on the machine that will host the registry. A summary of the steps taken are described below.

1. Configure the registry to be insecure, as described in detail [here](#). Add the following information to the **/etc/docker/daemon.json** file on all nodes that will host the Jupiter pods:

```
2. {
3.   "insecure-registries" : ["gnat0.bbn.com:5000"]
4. }
```

If there is no **/etc/docker/daemon.json** file create it. If the file exists and has entries in it, terminate the last line of the existing configuration with a ',' (comma) before adding the "insecure-registries" line from above. Once the **daemon.json** file has been updated, restart docker, as **root**, with the following commands:

```
systemctl daemon-reload  
systemctl restart docker
```

5. Start the Docker registry, named **gnat_registry**, with the following command, as **root**, on the machine to host the registry:

```
docker run -d -p 5000:5000 --name gnat_registry registry:2
```

We are now able to push and pull docker images from our insecure local private docker registry.

15.5.2 Examining Registry Contents

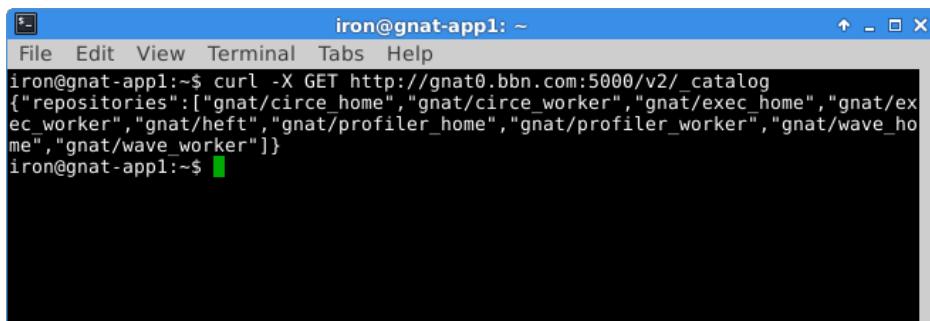
Once we have pushed some images to our newly created registry, we can examine the contents of it.

15.5.2.1 Listing Repositories

The following command is used to list all repositories in our Docker registry:

```
curl -X GET http://gnat0.bbn.com:5000/v2/_catalog
```

Example output from executing this command is shown below.



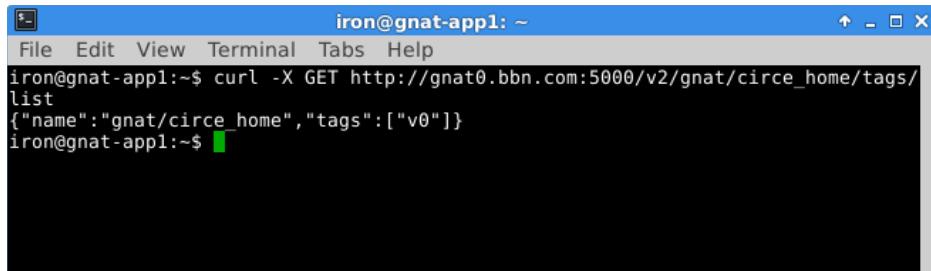
A screenshot of a terminal window titled "iron@gnat-app1: ~". The window shows the command "curl -X GET http://gnat0.bbn.com:5000/v2/_catalog" being run, followed by its JSON output. The output lists several repository names under the key "repositories": ["gnat/circe_home", "gnat/circe_worker", "gnat/exec_worker", "gnat/heft", "gnat/profiler_home", "gnat/profiler_worker", "gnat/wave_home", "gnat/wave_worker"].

15.5.2.2 Listing Repository Tags

The following command, executed on the node hosting our docker registry, is used to list the tags associated with a docker repository by repository name:

```
curl -X GET http://gnat0.bbn.com:5000/v2/<repository_name>/tags/list
```

Example output from executing this command for the **gnat/circe_home** repository is shown below.



A screenshot of a terminal window titled "iron@gnat-app1: ~". The window shows the command "curl -X GET http://gnat0.bbn.com:5000/v2/gnat/circe_home/tags/list" being run, and the response {"name": "gnat/circe_home", "tags": ["v0"]}.

15.5.3 Registry Control

15.5.3.1 Stopping Registry

The registry can be stopped with the following command:

```
docker container stop registry
```

15.5.3.2 Starting Registry

The registry can be started with the following command:

```
docker container start registry
```

15.5.3.3 Removing Registry Data

Data can be removed from the registry (once stopped) with the following command:

```
docker container rm -v registry
```

15.6 Configuring and Deploying Jupiter

The Jupiter documentation provides a comprehensive set of steps to [deploy Jupiter](#). The following set of changes were made to the Jupiter configuration files per the instructions:

Step 2: Update Node List – The *nodes.txt* file was modified as instructed. The contents of the modified file are as follows:

- home gnat-app1.bbn.com root PASSWORD
- node2 gnat-app2.bbn.com root PASSWORD
- node3 gnat-app3.bbn.com root PASSWORD
- node4 gnat-app4.bbn.com root PASSWORD
- node5 gnat-app5.bbn.com root PASSWORD
- node6 gnat-app6.bbn.com root PASSWORD

Step 3: Setup Home Node – The following line was changed in the *jupiter_config.py* input file:

```
HOME_CHILD = 'localpro'
```

Step 4: Setup APP Folder – We did not do this step. Instead, we simply used the sample application that is distributed with the Jupiter Orchestrator.

Step 8: Push the Dockers – Since we set up a [local private Docker registry](#), modify the *jupiter_config.py* file to point to our registry. Change all instances of **docker.io** with **gnat0.bbn.com:5000**.

Note: Before executing the script to push the docker images, make sure the Python path is set up correctly by executing the following command in the local terminal:

```
export PYTHONPATH=$PYTHONPATH:/home/iron/Jupiter/Jupiter/scripts/
```

Then execute the following command to push the docker images:

```
python3 scripts/build_push_jupiter.py
```

Step 8: Setup the Proxy – We did not do this here. This was done when we [started a kubernetes proxy](#).

Step 9: Create the Namespaces – The following script was used to create the namespaces:

- `#!/bin/sh`
-
- `kubectl create namespace gnat-profiler`
- `kubectl create namespace gnat-exec`
- `kubectl create namespace gnat-mapper`
- `kubectl create namespace gnat-circe`

Also, replace all instances of 'johndoe' with 'gnat' in *jupiter_config.py*. These names must match the names of the namespaces created by the above script. Here one would substitute their login username in place of 'gnat'.

Step 10: Run the Jupiter Orchestrator – Before running the documented commands, we had to set the following environment variables in the terminal:

- `export KUBECONFIG=$HOME/admin.conf`
- `export PATH=$PATH:/usr/local/go/bin`
- `export PYTHONPATH=/home/iron/Jupiter/Jupiter`

15.6.1 Verifying Jupiter Deployment

Following the execution of the command to execute the Jupiter Orchestrator, we can one more time take a look at the continuous cluster monitoring that we started in [verifying master initialization](#).

```
iron@gnat-appl: ~
File Edit View Terminal Tabs Help
Every 2.0s: kubectl get nodes && echo && echo && kubectl get pods... Thu Dec 13 10:05:21 2018

NAME           STATUS  ROLES   AGE    VERSION
gnat-app1.bbn.com  Ready   master  46m    v1.13.0
gnat-app2.bbn.com  Ready   <none>  34m    v1.13.0
gnat-app3.bbn.com  Ready   <none>  32m    v1.13.0
gnat-app4.bbn.com  Ready   <none>  32m    v1.13.0
gnat-app5.bbn.com  Ready   <none>  32m    v1.13.0
gnat-app6.bbn.com  Ready   <none>  31m    v1.13.0

NAMESPACE      NAME                READY  STATUS    RESTARTS  AGE
gnat-circe     aggregate0-89bdd7b76-f9fn6  1/1   Running  0          4m49s
gnat-circe     aggregate1-84c6d67ff8-fvqkd  1/1   Running  0          4m51s
gnat-circe     aggregate2-84f4dcbb8-t8cxj  1/1   Running  0          4m54s
gnat-circe     astutedetector0-574485b8b-7426g 1/1   Running  0          4m49s
gnat-circe     astutedetector1-75d974d868-dh882 1/1   Running  0          4m49s
gnat-circe     astutedetector2-5df685b4db-fstgv 1/1   Running  0          4m52s
gnat-circe     dftdetector0-5d59f56cb7-zw7km  1/1   Running  0          4m54s
gnat-circe     dftdetector1-769596fd64-jdd7j  1/1   Running  0          4m52s
gnat-circe     dftdetector2-6458bdf9-78642  1/1   Running  0          4m47s
gnat-circe     dftsLave00-79898c9bd8-w2xqj  0/1   Completed 1          4m54s
gnat-circe     dftsLave01-bcc6d7bf6-dlfp6  0/1   Completed 1          4m54s
gnat-circe     dftsLave02-68c6d5b5c4-wtmkm  0/1   Completed 1          4m53s
gnat-circe     dftsLave10-75f9598c48-mqbbt  0/1   Completed 1          4m47s
gnat-circe     dftsLave11-6b4c94c74-k7l44  0/1   Completed 1          4m51s
gnat-circe     dftsLave12-f6b764648-blnt4  0/1   Completed 1          4m54s
gnat-circe     dftsLave20-5fd68c9fb6-tlhvf  0/1   Completed 1          4m49s
gnat-circe     dftsLave21-5dc66db676-v8sf5  0/1   Completed 1          4m49s
gnat-circe     dftsLave22-6cd754c8b-b7mpn  0/1   Completed 1          4m53s
gnat-circe     fusioncenter0-5777fbddfc-66q7t 1/1   Running  0          4m47s
gnat-circe     fusioncenter1-568f4b5d6b-2n96z 1/1   Running  0          4m48s
gnat-circe     fusioncenter2-795dbbdxfc-fzrdr 1/1   Running  0          4m54s
gnat-circe     globalfusion-f8886ccfc-bxmww  1/1   Running  0          4m51s
gnat-circe     home-6959b988d8-l6gpq  1/1   Running  0          4m14s
gnat-circe     localpro-56d7d95947-z7kh6  1/1   Running  0          4m54s
gnat-circe     simpledetector0-577867866c-mjr82 1/1   Running  0          4m48s
gnat-circe     simpledetector1-77bb949976-lszj9 1/1   Running  0          4m53s
gnat-circe     simpledetector2-76fb98c97f-j54s5 1/1   Running  0          4m48s
gnat-circe     teradetector0-549854f959-29bxz 1/1   Running  0          4m48s
gnat-circe     teradetector1-846866445c-59c29 1/1   Running  0          4m51s
gnat-circe     teradetector2-7b6d584bd-cxnlc  1/1   Running  0          4m52s
gnat-circe     teramaster0-554c4d94d7-8nzk9  1/1   Running  0          4m51s
gnat-circe     teramaster1-65888dbb58-zg2h4  1/1   Running  0          4m49s
gnat-circe     teramaster2-6f4f78fbff-5pkhz  1/1   Running  0          4m51s
gnat-circe     teraworker00-674f77d86b-w5ctk  1/1   Running  0          4m53s
gnat-circe     teraworker01-7df9879494-tfcrm  1/1   Running  0          4m54s
gnat-circe     teraworker02-58f5f74798-rq97v  1/1   Running  0          4m47s
gnat-circe     teraworker03-b56556f8-pwvtb  1/1   Running  0          4m48s
gnat-circe     teraworker11-56697bb978-nlgsn  1/1   Running  0          4m52s
gnat-circe     teraworker12-57c48bc5d4-nwwxb 1/1   Running  0          4m54s
gnat-circe     teraworker20-67d86c5fb4-vpkkn 1/1   Running  0          4m54s
gnat-circe     teraworker21-6975cbf74-h2lnt  1/1   Running  0          4m50s
gnat-circe     teraworker22-6979947788-65rl2 1/1   Running  0          4m54s
gnat-exec     dftsLave00-5b554877-krv5  1/1   Running  1          23m
gnat-exec     dftsLave01-6844d7bd44-r75fm  1/1   Running  1          23m
gnat-exec     dftsLave02-658d95c4c8-vsld8  1/1   Running  1          23m
gnat-exec     dftsLave10-574447ccb6-fx4qb  1/1   Running  1          23m
gnat-exec     dftsLave11-579bb4b6d-bq8bl  1/1   Running  1          23m
gnat-exec     dftsLave12-5b745797bb-lrcjq  1/1   Running  1          23m
gnat-exec     dftsLave20-55748c5c46-dnskk  1/1   Running  1          23m
gnat-exec     dftsLave21-fc7697cc8-qgslw  1/1   Running  1          23m
gnat-exec     dftsLave22-6645cf4796-5n6dp  1/1   Running  1          23m
gnat-exec     home-7fb874fc9b-2jxjh  1/1   Running  0          23m
gnat-exec     node2-cfdbdfc8c-9slw4  1/1   Running  0          23m
gnat-exec     node3-cc6d6d47d-2l1vk  1/1   Running  0          23m
gnat-exec     node4-56955c6485-sqfvm  1/1   Running  0          23m
gnat-exec     node5-5cfb9cfc864-mfq7f  1/1   Running  0          23m
gnat-exec     node6-766d766f95-t2vq6  1/1   Running  0          23m
gnat-exec     teramaster0-9dbc6b58-rrh4m  1/1   Running  0          23m
gnat-exec     teramaster1-c64949579-4p7zd  1/1   Running  0          23m
gnat-exec     teramaster2-55d9bfc98-cjrzr  1/1   Running  0          23m
gnat-exec     teraworker00-6c6547786b-bwnbq 1/1   Running  0          23m
gnat-exec     teraworker01-5b5d55bdcf-fvzdp 1/1   Running  0          23m
gnat-exec     teraworker02-77b64b65db-ltc5p 1/1   Running  0          23m
gnat-exec     teraworker10-55fd87d66b-96bfl 1/1   Running  0          23m
gnat-exec     teraworker11-57f9fdbf46-mfbqn 1/1   Running  0          23m
gnat-exec     teraworker12-66d7b4b4f8-pmmtf 1/1   Running  0          23m
gnat-exec     teraworker20-66fc55874b-vgxt  1/1   Running  0          23m
gnat-exec     teraworker21-74d65c665d-h2hfw 1/1   Running  0          23m
gnat-exec     teraworker22-7cdd6ddf66-g54mm 1/1   Running  0          23m
gnat-mapper   home-6799447975-dw7km  1/1   Running  0          23m
gnat-profiler home-5456bb75b-7n4c9  1/1   Running  0          23m
gnat-profiler node2-74fc756f65-dbn5t  1/1   Running  0          24m
gnat-profiler node3-654b5b76cc-8bfpg  1/1   Running  0          24m
```

We see from the above output that the Jupiter pods, in the gnat-circe, gnat-mapper, and gnat-profiler namespaces, are in the Running state. This means that the Jupiter Orchestrator has been successfully deployed on our new kubernetes cluster.

15.7 Some Useful Debugging Techniques

15.7.1 kubelet logs

The state of the Kubernetes kubelets can be viewed by executing the following command on the Kubernetes master node:

```
journalctl -xeu kubelet
```

During the creation of our cluster, this command identified some configuration errors that we were able to address. The result of executing the command is shown below:

15.7.2 Kubernetes Dashboard

[Dashboard](#) is a web-based kubernetes user interface.

15.7.2.1 Dashboard Deployment

The following command was used to deploy the Dashboard:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml
```

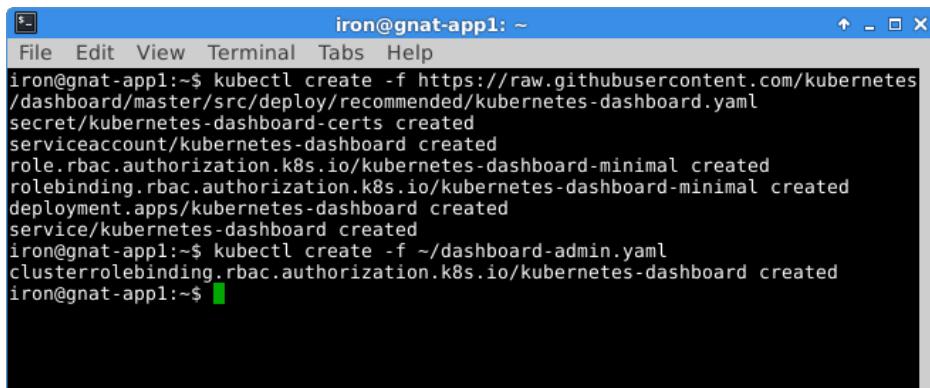
After deploying the Dashboard, configure it so the login step can be skipped (as described in full detail [here](#)). We found this to be necessary due to some recent security additions to kubernetes. In summary, this is accomplished by creating a file of the name, **dashboard-admin.yaml**, with the following contents:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: kubernetes-dashboard
  labels:
    k8s-app: kubernetes-dashboard
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: kubernetes-dashboard
  namespace: kube-system
```

Then, this was loaded with the following command:

```
kubectl create -f ~/dashboard-admin.yaml
```

The commands and their resulting outputs are shown below:



A terminal window titled "iron@gnat-app1: ~" showing the execution of several kubectl commands. The commands include creating a secret, service account, role, role binding, deployment, service, and a cluster role binding. The output shows each resource being created successfully.

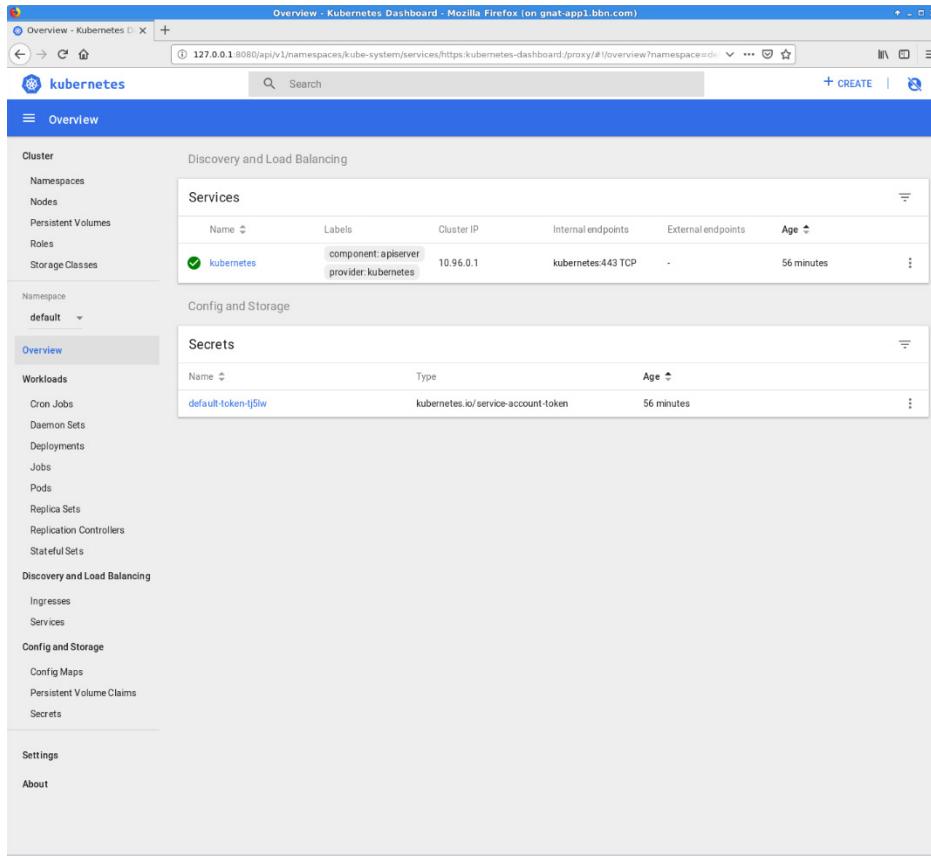
```
iron@gnat-app1:~$ kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml
secret/kubernetes-dashboard-certs created
serviceaccount/kubernetes-dashboard created
role.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
deployment.apps/kubernetes-dashboard created
service/kubernetes-dashboard created
iron@gnat-app1:~$ kubectl create -f ~/dashboard-admin.yaml
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
iron@gnat-app1:~$
```

15.7.2.2 Interacting with the Cluster via the Dashboard

Now that the Dashboard has been deployed, we can interact with our cluster. Launch a browser on the kubernetes master node and visit the following URL:

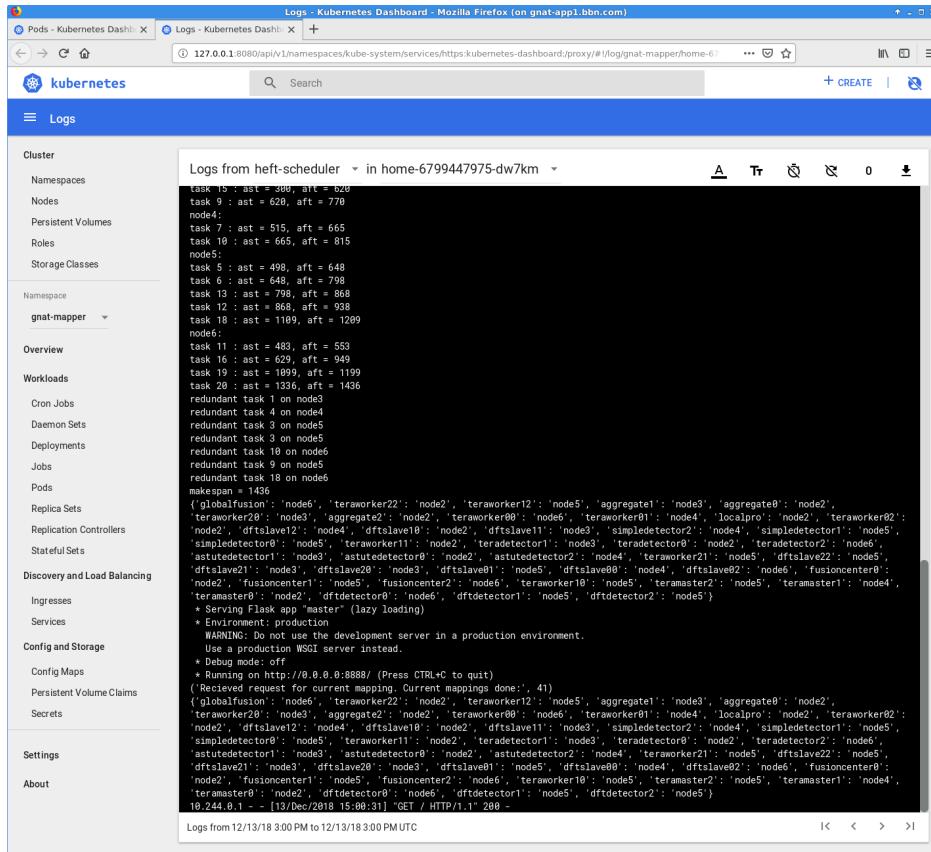
```
http://127.0.0.1:8080/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/
```

In the pop-up dialog, select 'Skip' to get to the main Dashboard page, illustrated below:



We can now view the Pods logs on the home node by performing the following:

1. Choose the desired Namespace in the Namespace drop down box on the left hand side of the Dashboard. To look at the mapper logs, shown below, the sgriffin-mapper item was chosen.
2. Choose the Pods button in the Workloads section
3. Choose the logs icon (denoted by 3 1/2 vertical lines) on the right hand side of the Pod whose name starts with home-...
4. View the contents of the home mapper Pod log file, shown below.



15.7.3 Viewing Pod Logs

In addition to the Dashboard, described above, the kubernetes **kubectl** command can be used to view the Pod log files. To do this, view all pods as described in [verifying cluster operation](#) and find the name of the Pod of interest and its corresponding Namespace. The following command was used to view the contents of one of the flannel Pods in our cluster:

```
kubectl logs -n kube-system kube-flannel-ds-amd64-4n4p2
```

The result of executing this command is shown below:

```
File Edit View Terminal Tabs Help
iron@gnat-appl:~$ kubectl logs -n kube-system kube-flannel-ds-amd64-4n4p2
I1213 14:23:13.222009 1 main.go:488] Using interface with name eno2 and address 10.1.3.2
I1213 14:23:13.234870 1 main.go:505] Defaulting external network to interface address (10.1.3.2)
I1213 14:23:13.345449 1 kube.go:131] Waiting 10m0s for node controller to sync
I1213 14:23:13.345532 1 kube.go:294] Starting kube subnet manager
I1213 14:23:14.345694 1 kube.go:138] Node controller sync successful
I1213 14:23:14.345771 1 main.go:235] Created subnet manager: Kubernetes Subnet Manager - gnat-appl.bbn.com
I1213 14:23:14.345811 1 main.go:238] Installing signal handlers
I1213 14:23:14.345997 1 main.go:353] Found network config - Backend type: vxlan
I1213 14:23:14.346085 1 vxlan.go:120] VXLAN config: VNI=1 Port=0 GBP=false DirectRouting=false
I1213 14:23:14.573040 1 main.go:300] Wrote subnet file to /run/flannel/subnet.env
I1213 14:23:14.573130 1 main.go:304] Running backend.
I1213 14:23:14.573165 1 main.go:322] Waiting for all goroutines to exit
I1213 14:23:14.573233 1 vxlan/network.go:60] watching for new subnet leases
iron@gnat-appl:~$
```

15.8 Teardown

15.8.1 Jupiter Teardown

Follow the Jupiter instructions for stopping the Jupiter Orchestrator.

15.8.2 Kubernetes Cluster Teardown

15.8.2.1 Worker Nodes Cleanup

The first step in tearing down the cluster is to drain and delete the worker nodes. This is accomplished by executing the following commands on the master node:

```
#!/bin/sh

kubectl drain gnat-app2.bbn.com --delete-local-data --force --ignore-daemonsets
kubectl delete node gnat-app2.bbn.com
kubectl drain gnat-app3.bbn.com --delete-local-data --force --ignore-daemonsets
kubectl delete node gnat-app3.bbn.com
kubectl drain gnat-app4.bbn.com --delete-local-data --force --ignore-daemonsets
kubectl delete node gnat-app4.bbn.com
kubectl drain gnat-app5.bbn.com --delete-local-data --force --ignore-daemonsets
kubectl delete node gnat-app5.bbn.com
kubectl drain gnat-app6.bbn.com --delete-local-data --force --ignore-daemonsets
kubectl delete node gnat-app6.bbn.com
```

Once the above commands have completed, run the following commands on each of the worker nodes, as **root**, to terminate any kubernetes processes:

```
#!/bin/sh

kubeadm reset
systemctl stop kubelet
systemctl stop docker
rm -rf /var/lib/cni/
rm -rf /var/lib/kubelet/*
rm -rf /etc/cni/
ifconfig docker0 down
ifconfig flannel.1 down
ip link del flannel.1
ifconfig cni0 down
ip link del cni0
systemctl start kubelet
systemctl start docker
```

15.8.2.2 Master Node Cleanup

Once the kubernetes components have been stopped on the worker nodes, run the following commands, as **root**, on the master node to terminate any kubernetes processes:

```
#!/bin/sh

kubeadm reset
systemctl stop kubelet
systemctl stop docker
```

```
rm -rf /var/lib/cni/
rm -rf /var/lib/kubelet/*
rm -rf /etc/cni/
ifconfig docker0 down
ifconfig flannel.1 down
ip link del flannel.1
ifconfig cni0 down
ip link del cni0
systemctl start kubelet
systemctl start docker
```

Once this step is complete, our continuous cluster monitoring terminal displays errors connecting to the cluster, as expected. We have now successfully destroyed our kubernetes cluster.