学　位　論　文

Design and Implementation of the SORA Multipath Virtual Network Layer

（SORA マルチパス仮想ネットワーク層の設計と実装）

平成２２年１２月１０日
指導教員：　森川博之・中尾彰宏

東京大学大学院新領域創成科学研究科
基盤情報学専攻

レーン・ジョン・ラッセル

John Russell Lane

ABSTRACT

DESIGN AND IMPLEMENTATION OF THE SORA MULTIPATH VIRTUAL NETWORK LAYER

DECEMBER, 2010

JOHN RUSSELL LANE

B.Sc., MICHIGAN STATE UNIVERSITY

M.Sc., MICHIGAN STATE UNIVERSITY

Directed by: Professor Akihiro Nakao and Professor Hiroyuki Morikawa

Application-directed routing architectures—those allowing applications some manner of choice regarding the network paths their data packets are to traverse—have been proposed as an approach to addressing many of the current Internet's most difficult problems, including policy, performance (QoS) and economics. Application-directed routing architectures provide solutions to these problems by giving applications a choice in route selection; applications can then select paths to meet given policy and performance goals; moreover, schemes have recently been proposed to foster Internet competition by using paths as a basis for the sale of Internet access.

However, many open problems are known to exist before such routing architectures can be deployed, and while testbed environments now exist which could provide a platform for realistic experimentation to address them, there exists no framework to implement application-directed routing—that is, there exists no *data plane* to forward packets based on application specifications; no *control plane* to establish network paths to be selected by applications; and no *application programming interface* by which applications and novel transport layers might be developed. This makes it difficult to perform application-directed experiments, which, in turn, hinders research and development of these open problems.

The research described herein provides a framework for application-directed routing, with the goal of enabling research and development on existing testbed networks. Specifically, it provides: an application-directed data plane usable on existing networks, a control plane for use with such data plane instances, and an application interface allowing ease of development and experimentation. Furthermore, a usage case of this framework is described wherein a means for enabling high performance with unmodified TCP/IP on multipath networks even under heavy packet reordering is developed and tested.

ACKNOWLEDGMENTS

This dissertation and the years of research and development that went into its creation would not have been possible without the long aid, encouragement, patience and kind counsel of advisors, mentors, friends and family.

First, I would like to thank Dr. Akihiro Nakao for patience and understanding through the long meetings, and for the discussion and advice he has provided over the years. Whether with few students (early on) or many (nowadays), whether in the early morning or late night, whether at the lab or while we were on the road, he provided the many comments and suggestions that helped bring this work to maturity and it would not have been possible without his efforts and support.

I would also like to deeply thank Dr. Hiroyuki Morikawa for sponsoring me long ago in 2001 and for providing continuing counsel, advice and encouragement over the years. Without his kind input and unwavering support, none of this would be possible.

Special thanks are due to the Japanese Ministry of Education, Culture, Sports, Science and Technology whose support made it possible for this research to be completed.

I would like to thank Jay Kusler, from whom I probably learned more about life than anyone, Dr. Bill Punch for his advice and encouragement, Dr. John Forsyth and Dr. Richard Reid for their patience and too many of the rest of the faculty and staff of the Michigan State University Department of Computer Science to name; particularly, the rogue secretaries. I would also like to thank my previous employers IGES, Turbolinux and the Ohtsuka International Patent Office for their understanding and support over the years.

Finally, undying gratitude is due to my mother, whose selflessness is a constant source of admiration for me, to my extended family for their long support and to the God who placed me here beyond all odds. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

Term	Definition	Page
address mapping	a relation established between a physical interface address of an entity and an entity identifier; used for mapping endhosts interfaces to virtual network and entity identifiers during ingress	85
application interface	the means by which applications interface with network data and control planes	52
application-directed routing	method for packet routing, whereby applications have access to an arbitrary number of paths and are made fully responsible for enacting a routing policy and making performance decisions via monitoring of path performance metrics	54
circuit switching	a network paradigm whereby the network concerns itself with the allocation and management of circuits between communicating endpoints	3
conduit	see *path conduit*	78
conduit endpoint	one of the two ends of a conduit	78
conduit endpoint identifier	*a unique numeric identifier identifying a single conduit endpoint instance on a given machine*	78
conduit endpoint interface	a data construct for performing communications between two virtual network entities; it manages the functions of path scheduling, path monitoring and monitored data feedback	78
conduit interface	see *conduit endpoint interface*	78
connection-oriented	a network paradigm whereby communication between endpoints is facilitated by first establishing a circuit or connection between them and then using connection	3
control plane	the set of data structures and functional components concerned with *determining how data should flow through a network*	24
convergence	a property of any distributed path computation algorithm implemented on a given set of routers and an algorithm is said to converge on a set of routes in a given system if, in the absence of further input (e.g., path faults), a single set of routes is eventually decided upon by all participating routers	15

Term	**Definition**	**Page**
data plane	the set of data structures and functional components concerned with *executing determinations made by the control plane*	24
default gateway	an computer on a local network which will perform forwarding on behalf of other local network machines	86
discovery	the means by which virtual network objects are queried for and thus obtained for use by virtual network entities	69
egress	preparation of ingressed packets for exit from a virtual network and transmission via an existing network	53
end-host path selection	method of packet switching, whereby a path is selected and specified to the network by the sending endhost	4
endpoint	an entity that is a source and/or destination of packets	58
entity	any virtual network participant which sends or receives packets via a SORA virtual network instance	58
entity identifier	a numeric identifier uniquely identifying an entity within a given virtual network	58
hop-by-hop routing	packet routing whereby each router independently selects a path from itself to the destination for a received packet	4
ingress	preparation of existing protocol (e.g., TCP/IP or UDP/IP) packets for input onto and transmission via a virtual network	53
ingressing router	a virtual network entity which acts as a default gateway on a local network while ingressing select outbound traffic and egressing any received virtual network inbound traffic	86
link abstraction	a data structure and associated methods which provide a unified interface to underlying tunnels for easing development of upper layers by hiding tunnel usage details	59
link data	data representing a link abstraction instance	69
link identifier	a numeric identifier uniquely identifying a link outbound from a given entity on a given network	59

Term	Definition	Page
link instantiation	the establishment of one or more links on a given entity using link data	72
link metric	describes one or more aspects of the link that may be criteria for use in performing best path set computation	69
link query	control plane service which allows entities to obtain virtual network link information	69
metric query	control plane service which allows obtainment of live virtual network link metric information	69
network identifier	a numeric identifier uniquely identifying a virtual network	58
network toolkit	software packages which replace and/or extend the functionality of the data and control planes and application interfaces provided by current networks	50
network tunnel	use of an existing network protocol for use in the transport of data of another protocol; examples include IPv6 in IPv4 for enabling IPv6, IP-in-IP for enabling virtual private networks, and the like	50
overlay network	see virtual network	50
packet abstraction	the data structure of the data plane by which data payload and control data is read from and written to a link instance	61
packet header	network control data included when sending a packet on a virtual network link	63
packet option	comprises packet option type, option data and option handlers	62
packet option data	comprises the actual data to be transmitted in the option	62
packet option handlers	functions which perform requisite option handling operations	62
packet option set	stores a fixed number of options within a packet instance	62
packet option type	uniquely identifies the option within a given packet option set	62
packet processing module	relatively small methods which, input a packet, perform a given task on it and return either the packet or an error	79
packet switching	a network paradigm whereby the network concerns itself with the transmission of packets from one host to another	4

Term	**Definition**	**Page**
path	a series of virtual network links connecting two endpoints; typically it is specified using the corresponding list of link identifiers	59
path conduit	an idealized construct for managing all paths between two communicating endpoints	78
path performance feedback	sending monitored information back to the originating conduit endpoint such that it can be used in path scheduling	78
path performance monitoring	either passive or active monitoring of paths being used	78
path query	control plane service which allows obtainment of path information	69
path scheduling	selection of the next path to be used for a given packet	78
research network testbed	a collection of computing and network hardware resources designed for experimentation	50
router	an entity dedicated to forwarding packets	58
routing	selection of a path for packet transmission	4
routing economic model	an instance of a means for paying for data transport	18
routing economics	means by which payment is made for data transport	18
routing policy	a set of rules which governs selection of routes or paths	12
signed link	link data accompanied by a cryptographic signature; the cryptographic signature proves to any router it is presented to that the link data has been validated by the link request service	75
source routing	method of packet switching, whereby a path is specified by a sender (source)	4
switching	selection of a path for packet transmission	4
virtual network	a logical network comprised of network tunnels between existing network computers	50
wrapper library	a dynamically loaded shared object file which overrides (*wraps*) existing shared library methods and modifies their behavior	85

CHAPTER 1
INTRODUCTION

In under two decades, the Internet has grown from a research network to a medium for a significant percentage of global communications as well as commerce. It has displaced long prevailing telephone and broadcast-video-based telecommunications models and our collective reliance on it for communications of all forms would seem irreversible. Moreover, its fundamental architecture has proven sufficiently flexible, extensible and scalable for implementation of today's wide variety of applications, sufficiently reliable and robust for non-critical applications, and has been able to scale its performance to meet the demands for ever richer content.

However, its centrality in world telecommunications and commerce has brought increasing scrutiny to shortcomings in its ability to meet future flexibility, reliability and performance demands and general agreement that the current architecture is insufficient to the challenge and in need of change. This consensus has, in turn, made improvement or even outright replacement of the current Internet structure an area of intense research, with national-level next-generation Internet research initiatives currently ongoing in Europe [57, 51], the United States [58] and Japan [3]. This consensus also provides the network research community a rare opportunity to make more than the merely incremental changes to the Internet's structure generally allowed with a production network. Therefore, it has become a priority to maximize the effect of such modifications and provide an architecture allowing its future reliability, performance and extensibility requirements to be more flexibly engineered, as future applications demand [58].

The next two sections provide historical details on the development of Internet interdomain routing and Section 1.3 discusses issues with its structure and their sources. Section 1.4 identifies past and present attempts at addressing these issues and Section 1.5

summarizes the current state of Internet interdomain routing with respect to these attempts. Section 1.6, formally defines the problem addressed herein; Section 1.7 defines how the present work addresses this problem. Finally, Section 1.8, provides a layout of the rest of the dissertation.

1.1 Historical Overview of Networking

This section and the next provide a historical overview of data networks sufficient to describe the issues that this work addresses, why they exist and why they are significant. Specifically, this section describes the development of communications from analog to digital technologies and from circuit to packet-switched paradigms. The next section discusses the current Internet regime, its guiding design philosophy, and its development with respect to packet routing.

1.1.1 Analog Circuit Switching to Digital Circuit Switching

Early communications systems, beginning in the mid-1800's with the telegraph and continuing through the 1940's with the telephone system, allocated a physical, end-to-end *circuit* to each communicating terminal or endpoint. Early telephone systems were manually switched via plugs, switchboards and human switchboard operators that intercepted calls and directed them to the destination requested by the initiator by extending a physical circuit successively closer to it. In the 1950's and 1960's, a fully automatic circuit allocation system began to fully replace the manually switched telephone network. The result was the public switched telephone network (PSTN)—a highly engineered, electro-mechanical switching network dedicated to one specific purpose: automatic allocation and deallocation of a circuit—a portion of the physical network—to a given phone call.

By the early 1980's, the PSTN had become global in scope, automatically providing for end-to-end circuit allocation, setup and teardown from the US to Europe to Asia. However, until the 1980's, in keeping with its original requirements, the PSTN was still largely an analog network—generally only for voice. Each circuit possessed a minimum bandwidth of eight kHz; the few existing digital devices (e.g., fax machines and modems) sent data over the network by modulating their bitstream onto the eight kHz signal telephone line, transmitting it, and demodulating the received signal back into a bitstream at the receiver. However, because of its increased bandwidth, its increased medium utilization, and its ability to be carried over a wider variety of physical medium (e.g., copper, microwave and fiber optic cable), the 1980's and 1990's saw PSTN carrier networks begin to quickly transition from analog data transmission to digital. Transition to digital transport also allowed them to offer a wide variety of data transport services with the same (or better) end-to-end quality of service. Examples of early circuit switching systems include Frame Relay [144] and X.25 [16].

Such systems were primarily concerned with a "packetized" version of the traditional method of communication: circuit switching. That is to say, they still established a circuit between communicating nodes and used the established circuit to carry data, but with two important distinctions from previous circuit-based systems like the PSTN. First, they were generally all digital and the data sent over the circuit was packetized. Second, while the PSTN reserved a physical circuit for communication, these networks did not necessarily do so, instead allocating network resources such as time slots sufficient to satisfy the application (e.g., voice) requirements. Either way, however—whether a physical circuit was allocated or the circuit was "virtual"—*a resource allocation was generally made for each connection and state information about that connection was stored in each router on the path*. Such circuit-based services soon become known as "connection-oriented" or "circuit-switched" because the network did not concern itself with the packets per se, as these were handled by the circuits, but the network was concerned with the management of the circuits which carried the packets.

So, while data was packetized and transmitted digitally, nevertheless, the fundamental purpose of a digital, circuit-switched network was generally the same as that of the previous analog/electro-mechanical network: the allocation and deallocation of an end-to-end communication circuit—a network resource to facilitate the reliable transmission of data (generally voice).

1.1.2 Digital Circuit Switching and Digital Packet Switching

At about the same time digital circuit switching was being developed, research into a different type of digital network, which had begun in the late 1960's was coming to fruition. This model was different in that *it did not divide network resources up and allocate them to data, it divided the data up into chunks* or "*packets*" (or "*datagrams*") and sent it all over the same, shared network. Each packet was typically individually addressed and independently routed[1], where *routing* or *switching* of a packet refers to the selection of a path for its transmission. Many examples of early *packet switching* architectures exist; notable among these include systems such as DECNET (DNA) [160], CYCLADES [126], TYMNET [157], TELENET [132], DATAPAC [30], SNA [60, 70], ARAPNET [38, 28], IP [125] and more recently, OSI [72].

Although they both use packetized data, these packet switching systems were distinct from the circuit switching systems in that they did not require any specific resource allocation. In fact, as will be discussed below with respect to IP, most of their designs purposely provided absolutely no means for resource allocation to be performed at all. Instead, nodes sent packets over the network by either specifying a given destination or an explicit route within the packet itself. When specifying an explicit route (e.g., via "Explicit Routes" in SNA), the specified route takes the form of a list of hop addresses or identifiers for each successive router on the specified path. Nevertheless, even when an explicit route was provided with the packet, no extra state information needed to be stored on the routers themselves. When a route was *not* specified, as was the common case in all of these systems, a destination address was provided with the packet and *forwarding performed at each router*. That is, each router along the path generally used the destination address to determine a path and thus a next hop for the packet.

Thus, schemes whereby a path is specified by an endhost or an application running thereupon are generally referred to as *source routing*, *endhost path selection* and, more re-

[1]Note that herein the terms switching and routing will be used interchangeably.

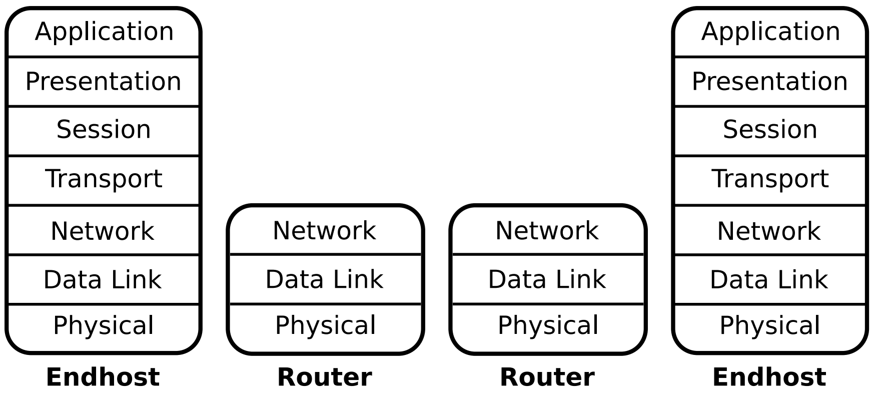


Figure 1.1: OSI Reference Model Network Stack

cently, *application-directed routing*. Schemes whereby each router independently chooses the path for each packet are known as *hop-by-hop routing*.

Of the early protocols mentioned above, only two survive to the present day: IP and OSI. Of these two, only IP is still in common use; OSI is still discussed today in large part because it codified a well-accepted, pedagogical model for packet-oriented networks known as the OSI network stack. Published in 1985, the ISO Open Systems Interconnect (OSI) Basic Reference is an ISO standard reference for implementing packet-oriented telecommunications systems [72]. As depicted in Figure 1.1, it divides the tasks of transmitting data to and from on network endhosts into seven layers: (1) physical, (2) data link, (3) network, (4) transport, (5) session, (6) presentation and (7) application.

Each of the seven OSI layers represents a distinct set of data handling responsibilities and the upper layers use the features of the lower. The bottom three layers: physical, data link and network are concerned with transporting the packet from one end of the network to the other and thus all network hosts must implement their functionality. Their roles are relatively well defined: the physical layer treats how to move bits through a given medium from one host interface to another; the data link layer treats how a given physical layer should be accessed (e.g., if multiple hosts are accessing it simultaneously) as well as how data running through a given physical layer should be framed, addressed and transmitted (e.g., issues such as flow control and error detection); the network layer treats how to move packets from gateway to gateway; from a network source to a network destination.

The top four layers generally function on communicating endhosts. Briefly, the transport layer is concerned with transparent transport of data from an application running on one endhost to an application running on another. The transport layer may include a variety of features; examples include instances that provide only a simple interface for unreliable transmission of datagrams and instances that provide error-free, in-order stream-based transmission, reordering, error detection and retransmission. The session layer is generally concerned with communication sessions between applications; this allows, for example, a single application session to span multiple transport layer instances. The presentation layer provides data conversion (or translation) services so that application layer instances can, for example, use different data representations and still communicate seamlessly. For instance, this would include translation between ASCII and EBCDIC character sets or between one XML format and another; the presentation layer allows applications to be more encoding-independent. Finally, the application layer is that closest to the user and provides the application-level software interface to the network.

While OSI was proposed as an international communications standard, it was never adopted for a variety of reasons, not the least of which was the high complexity of its upper layers [23], particularly in light of the more pragmatic implementation of IP, which only included the lower four layers in its design and left the rest to be implemented by individual applications.

In summary, packet switching offers a number of advantages over circuit switching. First, digitization allows data to be carried over a wider variety of medium, and packet switching allows data to be carried over the same wide variety of medium without allocation of a circuit. The ability to switch data without the need to allocate resources to it is of little importance when the data to be sent requires the same constant bit rate (e.g., voice), but greatly increases utilization when bitrates vary as they may when a variety of data is sent over the same network. Thus, packetization allows increased utilization when networks carry a greater variety of data. Second, because packet switches (i.e., routers) need not maintain state information regarding circuits, their construction is simpler than circuit switching equipment. Taken together, when transporting data with a variety of requirements, packet switching simultaneously increases utilization and decreases complexity; this, in turn, reduces network construction and operating cost.

1.2 A Brief History of Interdomain IP Routing

This section discusses the guiding design philosophy of the Internet Protocol, how it routes packets between domains, as well as how payment for packet service flows between domains. The Internet Protocol (IP) was designed for use in interconnected systems of packet-switched computer communication networks [125]. RFC 791, the IP specification, further states that the "protocol is called on by host-to-host protocols in an internet environment," and "calls on local network protocols to carry the internet datagram to the next gateway or destination host." IP itself contains no explicit mechanisms to implement end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols. Thus, IP is analogous to layer three of the OSI layer stack; it provides a method for packet exchange between hosts via cooperating networks, which may or may not be using IP internally.

1.2.1 Design Philosophy

Two distinctive elements characterize the design of IP routing: the *end-to-end argument* and its fundamental construction as a *network of networks*.

1.2.1.1 The End-to-End Argument

IP's design philosophy follows the so-called end-to-end argument in system design which states that, when drawing boundaries between functions in a communications system, when a given function "... can completely and correctly be implemented only with the knowledge and help of [an] application standing at the endpoints of the communication system ... providing that questioned function as a feature of the communication system itself is not possible." [138] In other words, communication functions which require knowledge from an application in order to function properly should be implemented at the endpoints, not in the network.

As was the case with the PSTN, communications networks constructed up until this point were highly engineered to precisely meet the needs of a specific application (voice). In this case, system parameters such as required throughput, bitrate, and latency as well as average duration of communication and number of terminal nodes were well known. However, the end-to-end argument described above is directed at *a more generic network environment where such application parameters cannot be known a priori*. The effect of the end-to-end argument was to place the complexity of handling application-specific uses in the same place where the application ran: the endpoints. For example, since the choice of what type of transport (e.g., reliable or unreliable) is required is application-dependent in IP, transport layer instances such as TCP [25], UDP [124] and, much later, SCTP [149] implement their end-to-end data reliability, flow control, sequencing and like features on the endhosts.

1.2.1.2 Routing Architecture: A Network-of-Networks

As its name suggests, the Internet Protocol was designed from the start as a protocol to provide packet routing between existing networks. Based on prior experience with ARPANET, IP was designed to be capable of interoperating with the many existing network protocols described above. That is, IP was designed to be a packet-switched *network which connected inter-connected packet-switched networks* (something historically known as a "catenet" [125]).

1.2.1.3 Interior and Exterior Gateway Protocols

Thus, from its inception, IP was agnostic about how packets were routed on local networks—any protocol could be used provided it was capable of delivering local network endhost packets to and from a gateway router connected to the Internet. Such local networks came to be called "interior" networks and, in contrast, the Internet was referred to as an "exterior" network. For sharing local routing information on such interior or intradomain networks, many protocols exist such as RIP [104] and OSPF [112, 32] and IS-IS [119]. However, the focus of this dissertation is interdomain routing.

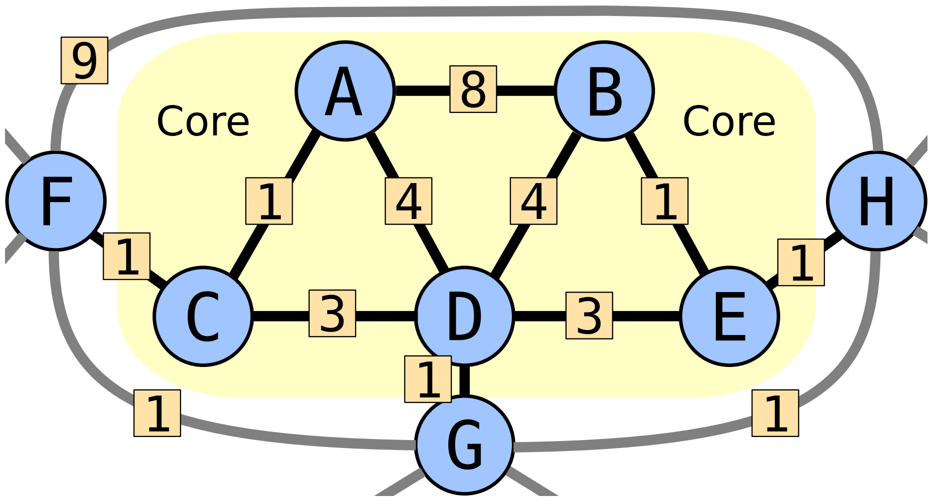
1.2.2 EGP

The original IP specification, released in 1981, specified that the Gateway-to-Gateway Protocol (GGP) be used be used to coordinate routing and other Internet control information [125]. Around 1985, GGP began to be completely replaced by EGP [134, 139, 109], the reasons for this being its overhead, the rigidity and inflexibility in the routing algorithm used with GGP and difficulty experienced in implementing fault isolation between gateways [134].

Like, GGP, EGP's purpose was to allow exchange of net-reachability information between Internet gateways. However, EGP introduced the concept of "autonomous systems"—sets of routers under a single technical administration, using an interior gateway protocol and common metrics to route packets internally [139]—and set forth a more formal framework for the implementation of network interconnection by defining the Internet's purpose as routing between these autonomous systems (AS's). The basic idea was to assign each AS an AS number (ASN) and to perform routing between the ASN's while effectively maintaining a mapping between IP address blocks (address classes) and AS numbers which served them. The AS abstraction scalably modeled the underlying split between exterior and interior networks and EGP used a distance vector-based protocol to share metrics and compute paths, which, at least in theory, provided the capability of using alternate paths in the event of a fault.

Distance vector protocols work by having neighboring systems exchange vectors of their current distance to all other systems. These protocols have three basic phases: (1) routers probe their neighbors and update their distance vector with the probe results if there is a change, (2) routers send and receive vector updates with neighbors, (3) routers recompute shortest paths to all systems (e.g., using the Bellman-Ford shortest path algorithm [13, 55]) based on the updated vectors and update their distance vector. This process is repeated until there is no change in step (1) and no updates are received via step (2);

the goal, at this point, is that all hosts should have the same view of network distances, all their routing tables should be consistent and there should be no routing loops; the process of arriving at this state known as "convergence," and a network system having reached this state is said to have "converged.".



(a) Example AS Network

To	**B**	**C**	**D**
A	8	1	4
B	0	5	4
C	5	0	3
D	4	3	0
E	1	4	3
F	4	1	2
G	3	2	1
H	2	3	2

(b) Update Vectors From B/C/D to A

A	**Next**
0	–
8	B
1	C
4	D
7	D
2	C
∞	–
8	**D**

(c) Pre-Update

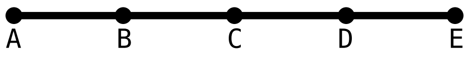
A	**Next**
0	–
6	**C**
1	C
4	D
5	**C**
2	C
3	**C**
4	**C**

(d) Post-Update

Figure 1.2: EGP Distance Vector Algorithm

For example, consider Figure 1.2 and the network in Figure 1.2a. Here, let us assume that at some point in the past, system G had failed and was no longer reachable, but has since come back online. The distance vector and next-hop routing table of system A is depicted in Figure 1.2c and the distance vector updates indicating G's reachability depicted in Figure 1.2b have now reached system A. After receiving these updates, A now calculates a new distance vector and routing table as depicted in Figure 1.2d. It then shares this with its neighbors and the process continues until the system converges.

Distance vector routing is known to propagate news of good routes quickly, but has a major flaw known as the count-to-infinity problem, which can lead to slow or non-convergence. The count-to-infinity problem arises because the algorithm includes no information regarding the actual path which a neighbor is using to reach a given destination. Consider the example of Figure 1.3. Here, when A fails, B should mark A as unreachable (a distance equal to infinity), but instead it sees that C has a route to A. Unfortunately, B has no way of knowing this route is through itself and the process slowly continues counting progressively higher distances to A as exchanges continue. While workarounds such as the split-horizon approach exist for coping with this flaw, there are graphs on which even these will fail [56, 33].



```
         1    2    3    4   < Initial Distances to A
         3    2    3    4   < After A Link Fails and B Uses C's Route
         3    4    3    4   < After First Vector Exchange
         5    4    5    4   < After Second Vector Exchange
         5    6    5    6   < After Third Vector Exchange
         .    .    .    .
         .    .    .    .
         .    .    .    .
         ∞    ∞    ∞    ∞
```

Figure 1.3: Distance Vector Algorithm Count-to-Infinity Problem

As the nascent Internet continued to add new AS's, the shortcomings in EGP's use of a simple distance vector-based protocol became obvious. For example, the EGP model limited the early Internet to spanning tree topologies with a designated "core" system at the center and "regional" systems at the leaves. Three main reasons existed for this restriction: (1) there was no generally accepted interpretation of the routing metrics used in the various networks; (2) due to the count-to-infinity problem and there being no agreement on the interpretation of metrics, there was no way to detect routing loops when they existed; and (3) even if routing loops could be ignored, when two or more paths existed to the same destination, the distance vector algorithm left no way to meaningfully compare them or to determine the AS's which they traversed (e.g., to select one which traversed the core as opposed to one or more regional networks), so it was impossible to declare one as "primary" and another as "secondary" [109, 129].

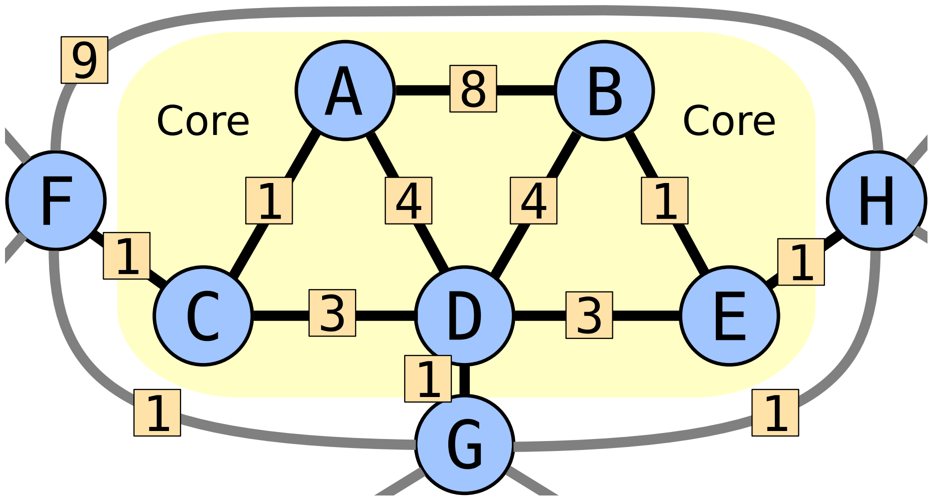
The limitation to spanning tree topologies served to protect EGP from routing loops and insure stability, but it meant that only the "core" system was allowed to indicate reachability via multiple networks; regional networks were generally not allowed to announce the reachability of another AS via a non-core path [129]. This made it difficult or impossible to implement many commonly desired routing policies. For instance, EGP did not allow regional (non-"core") AS's to establish bilateral agreements for packet exchange between themselves [129]. These problems were the main impetus for the design of the Border Gateway Protocol (BGP) [129, 113].

1.2.3 BGP

The Border Gateway Protocol (BGP) was designed to address the convergence issues of simple distance vector routing and allow implementation of a subset of "policy routing"—selection of routes based on a given set of rules. The primary technical means by which it addresses these issues is the adoption of what has now been termed "path vector" routing.

Path vector routing is a form of distance vector routing; however, it is distinct from standard distance vector routing in that shortest path (next hop) computation is not based solely on distance or cost metric information but on the *actual network paths which neighbors are using to deliver packets*; that is, based on a "path vector." With knowledge of the actual paths in use, it is possible to implement basic routing policies as well as to address the count-to-infinity problem.

A simple example of the type of limitation described above with respect to EGP, whereby EGP provides no means to meaningfully compare two or more existing paths, is depicted in Figure 1.4. Here, for instance, the administrators of an AS A would like to follow a policy of avoiding the use of paths which traverse links both of whose AS's are outside of the set of AS's in the area marked "core" (e.g., C-D and C-F are both acceptable, but F-G is not).



(a) Example AS Network

Next	Cost	Path Vector
B	9	B-E
C	5	C-F-G-H-E
D	7	D-G-H-E

(b) Distance/Path Vector Information From AS A to E

Figure 1.4: Exterior Gateway Protocol Algorithms: Distance vs. Path Vector

It is not difficult to see from this example that if AS A receives only distance information from its neighbors, it cannot know that the only path acceptable under its policy is via B. However, because it includes path vector information in its reachability information, any policy allowed by selecting from among the routes a given AS's neighbors are using (i.e., any policy implementable using hop-by-hop routing) is implementable using BGP [130, 113].

1.2.4 IP Routing Economics

While the merits of a data transport network model may be evaluated with respect to such factors as reliability, performance, extensibility or scalability, economic feasibility is often equally important in determining success or failure: if there is no way to make a system profitable, there is likely no means to pay for the infrastructure to support it.

By the early 1990's, commercial traffic was far outgrowing research traffic on NSFNET, which was largely a cooperative endeavor between autonomous systems and which was originally intended to be an exclusively research-only network. In 1992, the U.S. Congress even provided an exception allowing NSFNET to carry commercial traffic. However, a transition was already underway toward an Internet backbone formed by commercial telecommunications carriers, instead of NSFNET. Nevertheless, even after the transition was complete, the early NSFNET model described above, whereby a core set of network providers provided transit service to a larger set of regional providers was largely continued.

After the transition (and even to the present day), billing was generally implemented between AS's using BGP to implement routing policy. Here, with respect to payment for a given packet, two basic relationship types may be defined between AS's: *transit* and *peering*. Transit relationships can be further classified as *customer-provider*, whereby the former pays the latter, or *provider-customer*, whereby the latter pays the former; *peering* typically involves no fee exchange, and is entered into under the (presumed) benefit of both parties. Payment for network service was either *flat rate*, whereby a user or provider pays a single fee (typically per month) for the right to send or receive (transit) data, or *volume-based*, whereby costs increase by some function of total transit data volume; recently, however, nearly all inter-AS relationships are volume-based.

Under this scheme the most local AS's, often called "tier 3," make payment to higher-level regional AS's for transit services; these higher-level regional AS's, often called "tier 2," in turn receive transit services from and make payment to top-tier or "tier 1" AS's, which have adopted their "core" transit role directly from the old NSFNET structure. *Thus, it is worth noting that this hierarchy is largely an artifact of the original NSFNET structure.*

1.3 Current Issues in Interdomain IP Routing

The interdomain routing structure described above has provided scalability in terms of the number of hosts, routers and autonomous systems it supports, as well as the amount of traffic it routes and the speed at which it does so. However, the more responsibility is placed on the Internet for providing access to an ever-widening set of communications requirements, the more critical its shortcomings become. Such critical shortcomings in BGP have been pointed out in the literature with respect to three areas: (1) reliability and service quality, (2) policy routing and traffic engineering and (3) routing economics.

1.3.1 Reliability and Service Quality

As discussed in Section 1.2.2, *convergence* is a property of any distributed path computation algorithm implemented on a given set of routers and an algorithm is said to converge on a set of routes in a given system if, in the absence of further input (e.g., path faults), a single set of routes is eventually decided upon by all participating routers. This process is invoked each time new path information is provided to BGP routers—in the form of a new route (e.g., a restoration of a failed link), revocation of an existing route (e.g., failure of an existing link) or modification of an existing route (e.g., a change in AS preference). Thus, it is the effective means by which BGP (1) provides connectivity via failover to alternate routes, (2) maintains high performance via selection of better routes and (3) implements the afore-described policies.

Despite the fact that its path vector mechanism allows it improved convergence properties with respect to simple distance vector algorithms (e.g., avoidance of the count-to-infinity problem), the path vector algorithm has been shown to exponentially exacerbate the number of possible routing table fluctuations [90]. It has been shown that the theoretical upper bound on complexity for BGP convergence is $O(N!)$ where N is the number of AS's [89]; while the authors note this is only a theoretical upper bound, not likely to be encountered in practice, it nevertheless, indicates the level of complexity inherent in the distributed computation [89]. Because latency and packet loss have been shown to increase greatly during failover [89], the complexity of convergence has an major effect on Internet reliability and service quality; long convergence times in the face of failover, for example, mean slow response times to faults, poor reliability and poor service quality.

Unfortunately, even conservative estimates by ISP's put convergence time in the presence of a failover at around half of a minute [79] and a theoretical absolute minimum convergence time on a complete graph has also been estimated to be about 30 seconds [89]. Moreover, actual measured delays in Internet interdomain path failovers have been measured to average three minutes to to take as long fifteen minutes [90, 47] to recover completely from.

Further, a number of possible sources of routing instability have been identified, beyond simple hardware failure (e.g., cable cuts). For example, these including malicious or inadvertent misconfiguration and unforeseen interactions between exterior and interior gateway protocols as well as protocol timer interactions.

Coarse-grained policy controls are well-known to complicate BGP configuration [145]. It has been estimated that between 0.2% and 1% of BGP prefixes (address ranges mapped to AS's) experience a misconfiguration on a given day [103]. This issue has severe consequences because it has been demonstrated that it is possible for administrators to specify configurations which guarantee non-convergence [63]. Moreover, it is possible for such misconfigurations (e.g., so-called "black holes") to cause an AS in an entirely different region to experience service instability and even complete long-term service disruption; such events have actually occurred [8] and may have been the result of malicious activity [159].

Since misconfigurations have such potential for damage, it would seem that static, offline verification of BGP network configurations would be a reasonable solution. However, for competitive and security reasons, network operators are generally not inclined to release routing information. Moreover, even if all network operators were to make such routing information available, verification of global convergence conditions or even AS reachability has been shown to be either NP-hard or NP-complete, making dynamic solutions for identifying pathological configurations and coping with them the only feasible approach [63].

Moreover, even "normal" protocol interactions between BGP and various interior gateway protocols (e.g., OSPF) have been shown to be a source of delayed convergence. For example the use of time-varying metrics (from an IGP within an ISP) have been shown to produce "hot potato routing" or route flap—conditions whereby, in the presence of multiple routes, the route selected changes relatively quickly over time [153]. Such changes can produce rapid swings in end-to-end latency which are well-known to have a detrimental effect on TCP performance, for example.

BGP's path vector-based protocol addresses the shortcomings of EGP and allows for the subset of policy routing capabilities implementable using hop-by-hop routing. However, this has introduced greater complexity in terms of the number of routing table fluctuations possible as well as the computation complexity of routing table convergence. These issues with convergence have wide-spread effects, including failover times which are unacceptably high for mission-critical and live communications; moreover, research suggests it is not likely possible to improve BGP convergence times to support such applications. Moreover, it has also been shown that BGP is prone to configuration errors, that such errors are pervasive, that they can be critical to Internet performance and that static configuration validation is not generally viable. Finally, it has also been shown that even "normal" protocol interactions, where a best default route is selected, can cause routing performance issues for protocols sensitive to swings in latency such as TCP.

All of these issues are linked to BGP's dependency on convergence; thus, no matter how much BGP is improved, because its fundamental design requires convergence, it is unreasonable to expect restoration delays to be maskable, or hidden from users; moreover, it is even more unlikely that BGP or any hop-by-hop routing system can match the reliability guarantees provided by circuit-switched networks where, for example, resources can be provisioned for use in the event of a fault and failover carried out within milliseconds.

1.3.2 Policy Routing and Traffic Engineering Issues

As described in Section 1.2.3, policy routing refers to selection of routes based on a given set of rules. Rules, here, typically refer to the administrative desires of network operators. For example, a large network operator may provide fee-based data services (e.g., upstream Internet access) to regional AS's (e.g., ISP's). Such rules allow a network operator to provide those ISP's *transit service*—service connecting other AS's to them—

but to deny transit services to other AS's. However, BGP-based rules may sometimes also used to implement more political policies regarding networks or even territories that data my traverse. For example, "traffic originating in and destined for Canada should not traverse the United States," or "traffic should only transit Iceland if there is no other alternative." Such policies are configured on each BGP router.

Nevertheless, there are policies which BGP cannot enforce. For example, it is not possible with BGP to select a route to a given destination that a neighboring AS does not use for its own traffic; that is, BGP does not enable one AS to send traffic to a neighboring AS intending that the traffic take a different route from that taken by traffic originating in the neighboring AS [130]. This shortcoming arises because all traffic from one AS to another is expected to follow the same route—BGP is inherently restricted by hop-by-hop routing and can generally provide no means for paths to be selected on some other basis than destination AS.

Moreover, BGP uses selective path propagation as a means of policy enforcement—that is, an AS only announces an AS path to a neighbor if it wishes that neighbor to use the path [145]. At least two common problems arise from this practice: (1) BGP generally relies upon the lack of general knowledge of a route to prevent its use, however it has been shown that this method is not completely effective [61]; and (2) this selective path propagation complicates fine-grained route access control by excessively limiting propagation of route information [61]. A BGP-based solution to the first issue exists but involves the use of error-prone dynamic filters; a solution to the second requires fine-grained policy controls not currently provided by BGP [61].

While the case for finer granularity interdomain traffic engineering controls and thus finer granularity policy routing mechanisms has been made numerous times [122, 2, 46, 166, 36], the Internet's hop-by-hop routing paradigm whereby each AS independently selects the route taken by each packet, is at odds with such controls and makes them impossible to implement without a fundamental change to this paradigm.

1.3.3 Routing Economics

Routing economics refers to the means by which payment is made for data transport. An routing economic model for routing refers to an instance of a means for paying for data transport. There are also issues with present Internet routing economic model. The main problem is that (largely as an artifact of the old NSFNET structure, as mentioned in Section 1.2.4), there is generally no way for compensation to pass from a given user to all operators on the path their data flows through. This raises a number of issues. For example, *while a network user may enjoy the benefit of the resources of all operators along their data path, use of the "downward" portion is effectively paid for by the destination*, resulting in difficulties for providers. Moreover, top-tier providers effectively become a cash sink whereby compensation flows inbound to them from mid-tier providers but not outbound to mid-tier providers.

1.4 Past and Present Approaches to Interdomain IP Routing Issues

A number of approaches have been proposed to address the shortcomings of interdomain IP routing described above with respect to (1) reliability and service quality, (2) traffic engineering and policy routing and (3) routing economics. The number of extant proposals is high and varied, but this section shows that many of the proposals to address these shortcomings adopt multipath routing approaches. This section summarizes the most salient of past proposals, dividing them into five categories: those which (1) change nothing, (2) replace IP entirely, (3) extend IP from below the network layer, (4) extend IP from within the network layer, and (5) extend IP functionality from above the network layer.

1.4.1 Change Nothing: Heavily Overprovision

As discussed above, the general lack of any resource allocation primitive means that the Internet treats all packets equally, which makes it difficult for to match the performance guarantees of circuit switched networks. Indeed, it has been stated that "the Holy Grail of computer networking is to design a network that has the flexibility and low cost of the [best-effort] Internet, yet offers the end-to-end quality-of-service guarantees of the telephone network." [82]

As discussed in the next section, the search for this ideal has been a major focus of network research and development for the past three decades and the absolute necessity of resource allocation for certain services—that is, the need to treat packets carrying data of different services differently or *unequally*—in providing services on a future Internet is often an unquestioned assumption.

The issue of unequal handling of packets has recently become a topic of debate in society at large: ISP's have asserted their right to differentiate between services, ostensibly in order to improve their quality; critics have raised objections over ISPs assessing different rates to different user "classes". Such critics have pointed out that the current Internet follows a strategy of meeting service requirements by heavy *overprovisioning*—that is, by providing far more bandwidth than is required for any given service [117]. They point out that, such overprovisioning is capable of providing statistical guarantees for services without the large overhead and setup delay of a circuit-switching control plane and without the network state it introduces.

Moreover, serious proposals have been made to use the IP current network as a basis for a new type of packet routing which does not perform packet routing from one host machine to another but from a host or application to the required data [75]. Since then, a number of such *data-* or *content-centric* networking proposals [86, 74] have drawn research attention away from replacement of the current IP regime and toward more intelligent data location.

However, since they do not address the issue of the actual network paths selected for data transfer and only focus on obtaining data from any source, such schemes may actually provide performance worse than the current Internet. Moreover, while they may be well-suited for services such as bulk file download, approaches such as those based on the statistical guarantees based on overprovisioning as well as content-centric network are not sufficient for critical services such 911 (emergency services) which must operate even during times when network congestion is likely. Thus, it is difficult to imagine the construction of such services without better use of the multiple paths the current Internet is known to possess.

1.4.2 Interdomain IP Replacements

The architecturally cleanest approach to addressing the issues described above is to completely redesign the current routing infrastructure from the ground up and reimplement it with new software and/or hardware to provide better use of multiple paths. Unfortunately, this approach also incurs the highest up-front cost due to the need to update software and possibly hardware on all routers, switches and endhosts. Nevertheless, serious and technically sound proposals to replace IP have been made and include ATM, IPv6, Nimrod, 4D and RCP, as discussed below.

1.4.2.1 ATM

The central features of Asynchronous Transfer Mode (ATM) are its small frame (cell) size, making it well suited for voice communications, and its virtual circuit switching paradigm, which implemented the resource allocation primitive discussed above. However, instead of allocating a physical circuit of fixed bandwidth, ATM allowed allocation of a virtual circuit, with bandwidth and latency requirements granted and guaranteed by the network. ATM was originally designed to carry data over voice-centric networks [143] and provides for generalized QoS guarantees—applications may request a virtual circuit with a variety of characteristics (e.g., bandwidth and latency). Accordingly, ATM's routing and signaling protocols are relatively advanced and intricate.

Furthermore, ATM assumes both control and data plane functions; as such it is intimately concerned with both how data is carried over its virtual circuits and the actual forwarding to implement those circuits. It uses a short (53 byte, 48 byte payload, 5 byte header), fixed cell (packet) format amenable to fast hardware implementation and high network utilization. Partly to allow for this short size, ATM cells do not contain a destination address—only virtual path and circuit identifiers, which are used to select an output port at every switch along the path of a virtual circuit. Because ATM is packet-switched, using these a virtual path and circuit identifiers, it is said that ATM is *packet-switched* and *circuit-oriented*.

This packet switching scheme implies the need for a protocol not only to calculate routes through the network, but also to assign virtual path and circuit identifiers (to ports) at each switch along a given route. A protocol known as PNNI (physical network-to-network interface) provides for the establishment, maintenance and teardown of paths and circuits. That is, PNNI provides (1) for the selection of a path meeting the specified requirements, and (2) for the assignment of virtual path and circuit identifiers from input to output ports at each switch along the path. However, given the global scope of ATM, a flat network of such identifiers would simply not scale. Therefore, one of the main features of PNNI is hierarchical routing structure, allowing ATM subnetworks to abstract internal routing details from higher layers—providing for scalability.

ATM saw wide deployment in the 1990's and is still in use today. ATM remains unique in that it was intended as a complete end-to-end data routing and transport solution: its architecture allowed interdomain resource allocation and thus for bandwidth and latency guarantees to be made across network operator boundaries. This is important because unless resource allocation and reservation is made on a complete, end-to-end basis, performance guarantees are not generally possible (i.e., on the non-reserved links) [34].

While ATM specifications were even proposed for portable devices and wireless environments [6], for a variety of reasons, not the least of which was its high interface price and negligible performance benefit relative to Ethernet at the time, ATM never replaced IP and generally only saw widespread adoption within data centers for use in traffic engineering. [80]

1.4.2.2 IPv6

Long proposed as the successor to IP version four described above, IP version six (IPv6) [37] has been the subject of much criticism and debate [161], and even now that IPv6 implementations are complete in most modern operating systems and routers, the difficulty in completing its implementation has been great, due in no small part to the fact that there has been insufficient economic and technical incentive to fully switch over to IPv6 [87]. IPv6 represents no great architectural change to IPv4: it proposes no new resource allocation primitives [37] for reliable connection-oriented operation; it would likely continue to use BGPv4 via proposed extensions [105]; and its most effective provisions are a 128-bit address space to replace IPv4's 32-bit addresses and an extensible packet format allowing new endhost-based options to be implemented.

IPv6 was developed largely to address a perceived imminent need for a larger address as well as make routing more flexible; however, the use of network address/port translation (NAPT) in home routers and ISP's has, in large part, ameliorated this problem. While NAPT complicated the hosting of inbound services such as web servers on machines behind NAPT-based routers, peer-to-peer services such as Skype [142] and BitTorrent [31] have found suitable workarounds for implementing services on them. Accordingly, the perceived imminent need for IPv6 has never materialized. Architecturally, it represents an incremental step and does not provide new solutions to the routing challenges presented above. Moreover, because of its large address space, IPv6 is also likely to increase the number of prefixes which must be mapped to AS numbers for Internet routing. Since these mappings must be stored in the routing tables of every router, recent studies suggest that this may ultimately limit IPv6's scalability [120].

Accordingly, while IPv6 provides useful features such as stateless automatic host configuration [116] and packet header extensions, it does not address the routing issues discussed above and a solution to them must be sought even after it has been implemented.

1.4.2.3 Nimrod

Nimrod [24] is a hierarchical, map-based routing architecture implemented as a set of protocols and distributed databases. Given a traffic stream's description and requirements, Nimrod provides network devices with information sufficient to allow them to select their own network paths. The architecture addresses scalability by allowing arbitrary levels of clustering and abstraction of internetworks, by restricting routing information distribution, by caching and by limiting forwarding information. Users of Nimrod control establishment of paths while service providers control the maintenance of paths. Nimrod accomplishes path selection through the propagation to end-hosts of clustered, abstracted, recursively resolvable internetwork maps. It establishes several modes of transport including those similar to IP source routing, another similar to native IP hop-by-hop routing and finally a flow-state based protocol. Nimrod addresses current IP reliability and performance issues by allowing endhosts to select paths to meet their own application requirements, without requiring connection-switching and per-connection state maintenance on routers. While an interesting proposal, a consensus was never established on the sufficiency of this method at addressing current IP issues and providing for future network requirements, and thus it was never implemented.

1.4.2.4 4D and Tesseract

The 4D architecture [61] as well as its implementation Tesseract [167], is based on the supposition that the Internet is fragile and difficult to manage because of the complexity of its control plane (used to implement distributed routing algorithms across network elements) and management plane (used to monitor the network and configure data plane mechanisms and control plane protocols). Accordingly, they advocate the elevation of three key principles: *network-level objectives*, *network-wide views* and *direct control* via the use of four architectural planes: *decision*, *dissemination*, *discovery* and *data*. This design, they argue, allows for a separation of decision logic from protocol operation—something not possible with today's Internet architecture.

Network-level objectives call for each network to be configured via a high-level specification of performance requirements and expectations. *Network-wide views* allow a "coherent" snapshot of the state of each network component, providing network administrators an accurate, up-to-date view of the data plane. *Direct control* refers to the principle that decision logic should not be the responsibility of routers and switches which manipulate the data, but should be the purview and responsibility of control and management systems which communicate this information to them.

The *decision plane* is where all decisions driving network control are made including reachability, load balancing, access control, security and interface configuration. The *dissemination plane* exists to provide the system a robust and efficient communication substrate connecting routers/switches with decision elements. It attempts to address the problem of how to bootstrap a network configuration with a minimum of static configuration. The *discovery plane* is responsible for discovering physical components in the network and creating identifiers to represent them. It also provides information to the decision plane for construction of network-wide views. The *data plane* simply handles individual packets based on the state that is output by the decision plane. It may also have fine-grained support for collecting measurements on behalf of the discovery plane.

As noted in its ACM public review, 4D is not without critics. Many point to marked similarities between 4D's more salient features (such as centralized management) and those of past research and production architectures (e.g., IBM's SNA [60]) which, for a number of reasons, have ultimately failed to bring about the type of change 4D seems to promise. Many do agree, however, that 4D's problem summation is accurate and succinct; particular with regard to IP's conflation of control and data plane [61]. Finally, while it may provide a more flexible means for manipulating router forwarding tables, it does not directly address the use of multiple paths in meeting Internet performance and flexibility goals.

1.4.2.5 RCP

In addition to Tesseract, the 4D system's ideals are, in part, reflected in a more concrete proposal known by its authors as the Routing Control Platform (RCP) [50, 145].

Like 4D, RCP emphasizes architectural separation of routing policy from protocol. It accomplishes this by removing the task of route selection, propagation and policy enforcement from individual routers of a domain and placing it into a logically-centralized system termed the "Routing Control Platform". Feamster, et al. claim that RCP could be implemented incrementally: first AS-internally (as an intra-AS replacement for iBGP and internal BGP route reflectors), then externally (by handling communication with neighboring AS's and performing AS-wide routing decisions), and the final step–full intra-AS RCP.

The authors claim the accruing benefits from incremental implementation as follows. Intra-AS RCP: constraint checking, no undesirable "hot-potato" routing and more flexible traffic engineering. AS-wide RCP: simpler router configuration, network-wide traffic engineering, intelligent flap damping and improved prefix aggregation. Inter-AS RCP: new routing protocols and improved diagnostics and trouble-shooting.

RCP is predicated upon three fundamental principles: a *consistent view* of routing state, control of *routing protocol interaction* and support for *flexible, expressive policies*. A *consistent view* of the network is implemented via an unspecified, distributed method of centralization ("logical centralization"). Providing a consistent view allows easier policy expression, prevention of undesirable inter-router dependencies and better routing by eliminating inter-router state inconsistencies.

Better control of *routing protocol interaction* is another byproduct of centralization. Its advantages are said to include: no hard-wired protocol interactions (e.g., eBGP-IGP) to constrain operator control, fewer protocol inconsistencies (e.g., iBGP-IGP) to cause forwarding loops and/or route oscillation and possible enhanced overlay-Internet interactions by exposing more network information to overlay networks providing far greater efficiency than current polling-based methods allow.

Finally, centralization also affords more flexible, expressive policy support. This in turn is said to allow: a *direct* method for policy enactment (as opposed to BGP's current *indirect* means); improved policy enactment (over what BGP currently allows); and, greater inter-AS cooperation through configuration simplification.

RCP is a promising proposal, along the same lines as 4D for improving the flexibility of managing wide-area Internet routing. Challenges to implementation center around robustness ("logical centralization" requires careful planning to avoid a single point of failure), scalability (any centralization must be planned to avoid bottleneck creation) and convergence speed (to ensure that routing table updates to routers are timely and symmetric).

1.4.2.6 Architecture for User-Selectable Routes

Yang proposed [170] the development of an infrastructure based on the current Internet which could be incrementally adopted with the goal of allowing applications on endhosts to select the first few hops from a source toward the network core and last few hops toward a destination. The data plane described herein is different in that it is directed at experimental use and provides full path selection.

1.4.2.7 PoMo

PoMo (Post-modern Internetwork Architecture) [15] proposes a "post-modern" internetwork architecture which attempts to address some of the major problems of the current Internet routing architecture, mainly by: separating forwarding from routing, making addressing independent from routing, improving peering relationships, allowing greater information flow to endhost applications and creating a mechanism for secure path sales.

PoMo's data plane architecture is similar to that described herein in that it is based on a unidirectional "link" abstraction between two forwarding elements and paths are essentially formed via lists of link identifiers. The data plane described herein differs from that of PoMo in that it is targeted at flexible research and development use and provides no means for secure path sales; similarly, the control plane described herein also differs from PoMo's in that it is not intended for full-scale Internet use but is RPC query-based and meant for ease of experimentation.

1.4.2.8 Pathlet Routing

Pathlet routing proposes a new routing protocol based on the distribution of interdomain path segments which sources concatenate in order to form end-to-end paths [59]. It uses source routing based on forwarding identifiers placed in the packet header which identify a given pathlet. Routers learn of pathlets via a control plane which distributes them. It is argued that, because pathlet segments only depend upon the initial and final routers of the path segment, pathlet routing can provide for local transit policies not possible with BGP, and that, while this makes it impossible for AS's to implement policy based on a packet's ultimate destination and slightly increases state, that the benefits of enhanced traffic engineering and routing freedom outweigh these shortcomings [59]. Moreover, pathlet routing is also sufficiently flexible to allow emulation of other routing protocols such as BGP and NIRA. Pathlet routing's implementation is said to increase state only slightly over BGP and thus be implementable using routing technologies already available. The main challenge to path routing is that it requires significant modifications to existing Internet protocols such as BGP in order, for example, to distribute pathlet identifiers.

1.4.2.9 Current Next Generation Initiatives

Finally, for the past few years large-scale initiatives to construct and test next-generation Internet architecture alternatives have been ongoing in a number of countries; for example, the GENI project in the US [58], FIRE in the EU [51] and AKARI in Japan [3]. Thus the dominant thought on networks seems to be that the current Internet architecture is not extendable and a new architecture is required to meet future needs. Nevertheless, it is instructive to see what problems past initiatives to extend the Internet sought to address.

1.4.3 Extending IP From Below

As discussed above, computer network designers long sought a network architecture with the flexibility and low cost of the Internet and the end-to-end quality-of-service guarantees of the telephone network. After it became clear that an all-in-one (physical layer to application layer) solution via ATM was not feasible and IP would dominate, network designers sought a middle ground that was agnostic to physical and data link layers and would operate nicely with IP to provide the best of both worlds: a resource allocation primitive without poor utilization. The result was MPLS and, later, GMPLS.

1.4.3.1 MPLS

Multi-Protocol Label Switching (MPLS) [133] evolved in the early-to-mid 1990's from several technologies, but mainly from ATM and was eventually brought forth as a standard by the IETF (MPLS Working Group) in January 2001. MPLS is often described as a "layer 2.5" (L2.5) protocol because it generally performs network switching functions by examining a small header wedged between layer 3 (e.g., IP) and layer 2. The protocol was conceived during the early-to-mid 1990's when it was supposed that conventional IP routers would be unable to scale to meet the increasing throughput needs of future networks (gigabit and beyond). However, the advent of wire-speed IP routers and routing switches proved this incorrect and many supposed MPLS might disappear. Nevertheless, one of MPLS's design goals was to allow the hardware switching advantages of ATM to be realized on non-ATM layer 2 hardware. Accordingly, MPLS has been adopted as a traffic engineering tool for network operators to use internally. It is currently widely deployed, for example, in the core routers of ISP's to handle internal routing of transit traffic.

A basic MPLS nomenclature is as follows. MPLS *domain*: a contiguous set of MPLS-capable nodes; *edge nodes*: border nodes which connect to non-MPLS systems or other domains; *forwarding equivalence classes* (FEC's): classes of packets all of which are forwarded in the same manner (same path, same priority, etc.); *labels*: short, fixed-length identifiers (generally of local significance) placed in a frame to identify a particular FEC; *NHLFE*: the Next Hop Label Forwarding Entry maps labels to actions; *label stack*: an ordered set of labels (in a frame).

MPLS switching is conceptually simple: upon receiving a frame, a node examines the label atop the label stack, looks this label up in its NHLFE and takes action accordingly: usually popping the label from atop the stack and forwarding the packet out on the interface indicated. However, a given NHLFE can also trigger other actions. A few examples are: pushing several more labels on top of the stack—perhaps tunneling the frame (e.g., for a VPN); rewriting the stack labels entirely (e.g., for traffic engineering in a multipath setting); and giving the packet priority in queueing (e.g., in a QoS setting).

The stack-based approach is simple, yet powerful, but major complications come in the form of NHLFE table propagation. In the most common case of IP-based MPLS networks, basic FEC's are formed directly from wide-area (i.e., EGP) routing and/or local area (i.e., IGP) routing tables. Though the mechanism of FEC label propagation is left to MPLS, the logic of those labels is determined directly by routes propagated by the given higher-level (IP) protocols.

To some degree, this still allows network operators to perform traffic engineering and add QoS functionality (e.g., by using static FEC's for traffic engineering and a QoS signaling protocol such as RSVP [68] for QoS). However, pairing MPLS and current IP with its limitations, which, as will be discussed later, include slow convergence, lack of multipath routing options, and the like, serves to diminish MPLS's capabilities.

To sum, MPLS brings the benefits of an end-to-end switched underlay network and has proven invaluable to network operators in traffic engineering. As is discussed later with respect to RSVP, it has also been used as a tool for implementing resource allocation with resources being accessed via assigned labels [7, 85]. Moreover, unlike ATM, MPLS is data link layer-agnostic, capable of being used with any link layer technology, and some in the research community have seen promise in its use with edge networks and even wireless edge nodes [164, 140, 95, 101], raising the possibility that it could be an end-to-end multiprotocol switching solution. However, with regard to end-to-end use in securing network resources, it is not generally possible to use MPLS labels between network operators and therefore end-to-end resource allocation is still not feasible with MPLS even in view of protocols like RSVP.

1.4.3.2 GMPLS

Generalized Multi-Protocol Label Switching (GMPLS), as its name suggests, generalizes multiprotocol label switching for use with all-optical networks and lambda-switching or dense wavelength division multiplexing (DWDM) in service provider networks. Until recently, networks were constructed from a large variety of underlying physical and data link layer technologies.

For instance, DWDM technologies have been introduced, whereby multiple carrier signals in a number of frequency bands are multiplexed onto the same physical fiber, each signal providing high speed transport for network data; atop that, time division multiplexing (TDM) technologies from the voice arena such as SONET/SDH provide a method for multiplexing disparate data streams onto a single channel to increase utilization; above that, ATM or MPLS is often used to provide traffic engineering (TE) and quality of service (QoS) guarantees for the IP traffic transported atop these technologies. [9]

Lack of a coherent design and all-encompassing control plane to unify administration of equipment providing multiplexing of data on so many levels (i.e., DWDM and WDM, TDM, fiber cable and even IP layer) made network management time-consuming and error-prone. It was seen as critical by the networking industry to address this issue while still supporting newer networking technologies and GMPLS was the response. [9]

First, the sheer number of virtual communication channels which DWDM multiplexes onto a single fiber (soon to number in the thousands or more) [9] make managing DWDM networks using today's largely manual and error-prone methods unacceptable. Not only would it become even more difficult to identify what data was traversing which endpoint, link or lambda, but assigning IP addresses to all of these is not feasible. To address these issues, GMPLS establishes an automatic address configuration mechanism which assigns "unnumbered addresses" to routers and has routers assign numbers to each of their link endpoints. Links, then, are ultimately identified by the tuple [router ID, link number]. These tuples are then used by routing protocols to identify routes on the network. Duplicates are unlikely and are handled by a separate process. [9]

Second, the variety of physical and data link layer technologies mentioned earlier (e.g., WDM, SONET, MPLS for IP) had an unfortunate consequence: each had a distinct and individually managed control plane. In other words, each came with its own setup and management interface – greatly complicating configuration, setup and administration and making it very difficult to automate configuration in direct support of traffic and network demands. [10]

In response, GMPLS presents a unified switching hierarchy reflecting the nature of DWDM. Higher hierarchy levels represent greater data stream specificity and are nested (or transported) within lower layers. The lowest level in this hierarchy is the coarsest; at this level, switching refers to the transfer of data signal from one switch port or fiber to another. This is generally known as fiber or port switching and equipment capable of this type of switching is known as Fiber Switching Capable (FSC).

In the next level up, switching refers to the mapping of a lambda on an input port to a (possibly different) lambda on an output port. This is generally known as lambda or wavelength switching and equipment capable of this type of switching is known as Lambda Switching Capable (LSC).

Switching one level further up in the hierarchy refers to the mapping of data input to time slot (i.e., to time division multiplexing). This is generally known as TDM-switching and equipment capable of this type of switching is known as Time Switching Capable (TSC).

The highest level in the hierarchy refers to switching based on the content of a packet header such as an IP or possibly an MPLS label. This is generally known as packet switching and this type of equipment is generally known as a packet switch or a router. [9]

Switching in GMPLS is performed based on a label stack like that of MPLS where the stack has a type identifying to which part of the hierarchy it is referring. Thus, GMPLS generalizes MPLS to the management of a hierarchy of optical networking technologies and represents a control plane therefor.

However, GMPLS has a few drawbacks. First, because it is primarily aimed at control of optical technologies, it is not a "universal" network control plane. For example, although work has been done to partially alleviate this issue [97], GMPLS is still not generally usable for controlling Ethernet devices, which have a strong installed base. Second, and most importantly, GMPLS, like MPLS, still does not operate between operator boundaries. While work continues on the latter issue [168, 36], these two drawbacks currently make it impossible to use GMPLS as an end-to-end switching solution, for example, to allow better exploitation of existing paths.

1.4.4 Extending IP From Within

Simultaneous with the development and eventual rollout of ATM in the mid-to-late 1990's, extensive research and development was being performed with the goal of implementing QoS on IP networks *without* the aid of an underlying QoS-supporting layer 2 network like ATM. Two models emerged from the IP research community: *integrated services* and *differentiated services*.

Integrated services or IntServ allows the network to identify individual packet flows and respond to (and provision for) their needs; the term refers to a "an Internet service model that includes best-effort service, real-time service, and controlled link sharing." [21]. In other words, IntServ is a QoS model which allows the network to respond to *each* application requirement. In an integrated services network, applications may make direct requests of the network for precisely the type of network resources they require and the network will respond—granting the request if possible, denying it otherwise.

IntServ requires the network to maintain state for flows which request special resources (beyond best effort delivery). Although it does present an attractive platform upon which to build applications, state maintenance makes IntServ difficult to implement on otherwise stateless networks like the Internet.

Differentiated services [17, 64] or DiffServ defines a fixed number of behavior classes. All packets which traverse a network are assigned to a specific traffic class. Each class is associated with a particular type of handling required of any given packet in the class. These handling types, known as a per-hop-behaviors (PHB's), are then used by routers in determining which packets are dropped when congestion occurs or possibly even which packets should traverse faster routes, or those more in keeping with the goals of their traffic class.

DiffServ is relatively easy to implement on a stateless network such as the Internet and, as such, represents a pragmatic and realistic approach to providing QoS guarantees. However, the lack of flow differentiation ultimately makes it nearly impossible for service providers to provision and *guarantee* resources for a given flow. Furthermore, the fixed number of traffic classes represents a limitation to the development of new applications.

In a theoretical sense, IntServ and DiffServ lie at opposite ends of the same spectrum; for example, an IntServ network can be conceived of as a DiffServ network which assigns a per-hop-behavior class *for each flow*. Not surprisingly, efforts at implementing the DiffServ and IntServ models on the Internet have, in part, resulted in two very different protocol suites: RSVP and MPLS, respectively (which are discussed next in Section 1.4.4.1 and briefly in Section 1.4.3.1). It is not without irony, then, that RSVP and MPLS were harmoniously merged into GMPLS, the subject of Section 1.4.3.2.

1.4.4.1 IntServ and RSVP

At the heart of the proposed IntServ model [21] is RSVP [173]. RSVP is a network resource reservation protocol with the following stated goals: (1) a modular design to accommodate heterogeneous underlying technologies; (2) accommodation of heterogeneous receivers; (3) an ability to exploit the resource needs of different applications in order to use network resources efficiently; (4) allow receivers to switch channels; (5) an ability to adapt to changing multicast group membership; (6) control protocol overhead so that it does not grow linearly (or worse) with the number of participants; and, (7) adapt to changes in the underlying unicast and multicast routes.

In accomplishing these, it assumes the following design principles: receiver-initiated reservation, separating reservation from packet filtering, providing different reservation styles, maintaining "soft state" in the network, and protocol overhead control and modularity. RSVP is said to be "... primarily a vehicle used by applications to communicate their requirements to the network in a robust and efficient way, independent of the specific requirements." [173]

RSVP-capable routers make use of an independent admission control mechanism to reserve resources; this mechanism is provided a flow requirements specification by the receiver in order to determine whether a given request can be accepted. Thus, RSVP is a method for message resource reservation request passing and uses whatever underlying routing protocol exists for message delivery. Separating it from the routing mechanism, QoS implementation and admission control mechanism allows it to accomplish its first goal of modularity.

The two most central RSVP messages are: *Path* messages which specify the properties of an existing data source flow, and *Receive* messages which embody a resource request made on the part of a receiver. Both of these message types contain flow specification requirements or "flowspecs," describing the specific requirements of a data flow (in the case of a *Path* message) or the specific requirements of a host (in the case of a *Receive* message), along with a few other details. By allowing both sources and receivers to provide flow specifications, RSVP allows individual hosts to request *only* the resources they actually require—not necessarily the resources specified by a source's flow specification. Here, RSVP meets its second and third goals.

Furthermore, the protocol allows receivers to provide a "filter" (e.g., a list of IP addresses to receive from) allowing them great flexibility in specifying precisely the senders they wish to "spend" their allocated resources on; this filter can also, in some circumstances, be dynamically modified to allow receivers to "change channels" (change sources) without making additional resource reservations. In this way, RSVP meets its fourth requirement.

The specific method by which RSVP forwards messages is straightforward. *Path* messages are simply sent along the path from source to receiver determined by the underlying routing algorithm (i.e., they are sent "down" a multicast tree). These messages indicate the resource requirements of the given source (but do not imply that a reservation is made).

Receive messages indicate the desire of a receiver to reserve the resources specified in the flowspec of the given message. These messages are sent on precisely the reverse path of the *Path* messages sent by the source. RSVP implements this "reverse forwarding" by storing the address of the last hop upstream (toward a source) when it receives *Path* messages. It is thus able to "reverse" the route of the underlying network and allow upstream message passing along the same downstream path from the source. This is important because while RSVP cannot expect the underlying routing algorithm to provide matching upstream and downstream routes, RSVP still requires a method to pass messages to neighbors on the path.

When an RSVP-capable router receives a *Receive* message, it passes the corresponding flowspec to the afore-mentioned admission control mechanism which determines if the request should be granted. If not, the request is rejected and RSVP "routes" this reject message back to the requester via the router from which it received the request (or the requester itself if they are adjacent); each router passes the rejection back downstream (eventually to the receiver, if they are not adjacent). If the request is accepted, RSVP passes the message along to the next router up the multicast tree as described above. In this way, each request is handled in a distributed fashion while requiring as few routers as necessary to process them, thereby meeting RSVP's sixth goal

Reservations are stored using "soft state": that is, source availability and path reservations are both expired after a given time interval and must be periodically re-registered by sources and receivers (using the same *Path* and *Receive* messages). Though the use of soft state requires more updates, it has the advantage of allowing RSVP to regain consistency and stability even in the face of host crashes and lost path and receive messages. Further, *Path* message updates allow RSVP to quickly adapt to changes in the underlying routing tables, meeting its seventh goal.

1.4.4.2 BGP Fixes

A number of mathematical models have been proposed to improve both BGP convergence and stability. These models first generalize BGP to some abstract and then move on to prove conditions for properties such as convergence and optimal path selection using it.

Research has been performed to mathematically generalize path-vector protocols and present an algebraic framework for manipulating them, thereby allowing them to be generalized and reasoned about more formally [146]. Therein, protocols were shown to possess one or both of the properties of isotonicity and monotonicity; and it is proven that algebras having the isotonicity property converge on optimal paths; those having monotonicity can be shown to eventually converge. This algebraic framework allows a convergence property to be proven with respect to a set of assumptions regarding the type of path computation performed on each system. One practical proposal for improving BGP uses the path vector algebra to allow construction of wide-area routing protocol schemes with basic consistency guarantees [62]. Nevertheless, it is difficult or impossible to enforce such assumptions on a system as large and varied as today's Internet.

Research has also been conducted to build on some basic observations regarding pathological BGP configurations and generalize them into five properties—each representing an important aspect of wide-area routing [48]. The five properties and a brief description of each: route *validity*: routes should reach their destination; path *visibility*: originating nodes of a given path should also know about the path; *safety*: a stable routing state exists; *determinism*: given a starting state and a set of inputs, the protocol should always reach the same end state; and *information-flow control*: routing messages should not expose more information than necessary. These wide-area logic rules have also been proposed as a basis for a tool which automatically detects static BGP routing configurations violating the given constraints [49].

1.4.4.3 **NIRA**

NIRA [169] is a system that attempts to foster economic competition between providers by allowing users to select the provider networks their packets will traverse. It accomplishes this by dividing the Internet into a hierarchy of routing domains roughly based on degree of interconnectivity and predominant business relationship (sibling, peering or provider-customer). Top-level or "core" domains are highly interconnected and tend to have only sibling or peering relationships, whereas lower-level domains tend to have less interconnection and feature peering and more often provider-customer relationships.

NIRA routers are assigned address prefixes for which they route all traffic. Recursively, in a top-down fashion (starting from the core), routers are assigned subprefixes of higher-level routers. Multiply-connected routers are accordingly assigned multiple subprefixes. In this fashion, a path from a low-level router (possibly traversing higher-level routers) and proceeding directly to the core can be completely specified simply by providing the appropriate subprefix. Finally, end-hosts are assigned intra-network IDs unique within their domain. These end-hosts may have multiple routers associated with them, each with address prefixes associated with them. A path from an end-host to the core may then be specified simply by appending the given host's network ID.

NIRA's fundamental idea is simple and elegant, however a good deal of complication is created by "non-canonical" paths which connect endhosts without reaching the core (e.g., those created when two ISP's peer). NIRA mandates that such paths generally be specified using multiple addresses. Methods for provider billing are suggested but not discussed concretely. Path discovery is provided for via a path propagation protocol which operates on a broadcast basis within domains. Inter-domain path discovery is provided for via a DNS like lookup service.

NIRA is an interesting architecture in that it addressed a complex routing problem through a clever use of addressing. However, its heavy reliance on the concept of an "Internet core", its failure to allow users to select end-to-end paths and its heavy reliance on the existing hierarchical billing structure remain open problems.

1.4.4.4 BANANAS

BANANAS [81] proposes much finer-grained control over path selection than NIRA—allowing users to select individual routers along a given path or subpath. Its authors propose to extend IP by adding a "path ID" to packets, allowing specific routes to be specified. This path ID can come in two forms—one inflexible and one somewhat more flexible.

The inflexible method uses a simple MD5+CRC (32 bit entity) of the ordered path segments which prevents routers along the path from intelligently reading and possibly manipulating it and also requires them to maintain state for each path. The other uses a bit vector where each n-bit-segment represents an encoding of an interface number of a router on the desired path. Though the first method is more robust in the face of

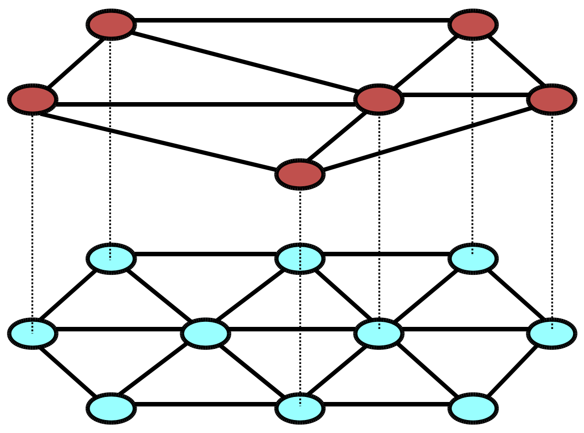


Figure 1.5: Overlay Network Depiction

rapid routing table changes (e.g., the norm with mobile ad hoc networks or in BGP), the second method is more feature-rich and more scalable.

The system is similar in spirit to MPLS (treated in Section 1.4.3.1), though it removes some flexibility (path lengths are not unlimited or “stackable”) and adds one feature: no explicit signaling is required to distribute labels. The authors propose updates to support multiple path usage in intradomain OSPF and also apply their idea to inter-domain BGP allowing some extension to basic BGP functionality. Beyond modifications of support for intra-domain routing protocols, BANANAS implements no method for multi-path route discovery—specifically no method for discovery of multiple inter-domain paths is provided.

While the system does provide the ability for end-users to specify end-to-end paths, BANANAS lacks any method for discovery of such paths and/or path components or any explicit method for their computation.

1.4.5 Extending IP From Above

Due to the difficulty and expense of modifying the Internet architecture to suit new requirements and the immediate need for a solution, many approaches to implementing the required functionality have been made at the application layer. Most of these use some manner of what is generally referred to as an *overlay network*.

Overlay networks overcome deficiencies inherent in the Internet routing structure "overlaying" logical networks atop the existing structure in a fashion like that depicted in Figure 1.5 and operate by placing packet routing functions onto endhosts and passing packets from endhost to endhost—using encapsulation between them—to effect a desired route. Often this desired route is one which is not selectable by the current Internet infrastructure, thus overlay networks can provide a method to immediately route around path faults and provide a means for selecting better-performing paths.

Overlay networks have often been employed as a solution in decentralized systems—those without any main server. Because they employ the direct cooperation of peers, they are sometimes referred to as peer-to-peer networks. Such applications range broadly and include file storage and sharing [88, 65, 29, 35, 135, 67, 31], host mobility [110], distributed databases [71] and multi/anycast [127, 128, 137, 177, 150], key/value storage and retrieval [136, 151, 176] and telephony [142].

Moreover, overlay networks have also been employed to address general routing issues. Examples here include enhanced path fault resiliency QoS [98], path (route) selection [150] and enhanced routing [4]. Below are outlined two of the more salient proposals for generic *packet routing* implemented at the application layer of the current Internet, *i*3 and RON.

1.4.5.1 Internet Indirection Infrastructure (*i*3)

The Internet Indirection Infrastructure (*i*3) [150] proposes an indirection-based routing system which uses a distributed hashing function to map an m-bit identifier (a semi-random bit-string) on to a network address space (or possibly a stack of other identifiers). The system is an overlay network, presently based on the current IP architecture.

It accomplishes its routing function using two primitives: `insert(identifier, address)` and `send(identifier, data)`. Both use the afore-mentioned distributed hashing function: `insert()` places a "trigger" in the network (i.e., it maps `identifier` to `address`) and this trigger is acted upon by the `send()` primitive (i.e., it looks up `identifier` and sends `data` to `address`, which it maps to).

In its simplest form, $i3$ places a proxy between two hosts which handles lookups on and updates of the identifier which represents the address of their communication session. It enables mobility by allowing `insert()`'s of identifiers in ongoing sessions. It enables multicast by allowing any number of network addresses to be mapped to a given identifier. It enables anycast through the use of inexact matching of identifiers: an anycast group chooses a k-bit ($k < m$) identifier for their anycast session and this is used to form the top k bits of their identifier. The bottom $m - k$ bits are application-specific. Packets are sent to the closest matching address.

A function similar to IP source routing is enabled by allowing stacks (or chains) of identifiers to be either (1) sent in the packet (sender-controlled); or (2) registered to a given identifier (receiver controlled). In both cases, identifiers are recursively resolved and sent, in stack sequence, to the hosts to which the identifiers map. This allows, for example, implementation of third-party data processing (e.g., HTML-WML translators).

Seen from another perspective, $i3$ is essentially a naming system in which identifiers are recursively mapped to services.

A number of security issues, which arise when hosts are allowed to control paths, are also addressed (albeit with marked increases in latency). Altogether, the system's main drawback may well be latency, which is not well addressed. Further, while novel, the architecture's reliance upon distributed hashing makes its reliability dependent upon the reliability of the hash itself, which is only capable of making statistical guarantees regarding availability.

1.4.5.2 Resilient Overlay Networks (RON's)

Resilient Overlay Networks (RON's) are overlay-network-based routing systems designed to address some of the deficiencies in current wide-area Internet routing [4] discussed above with respect to BGP. Specifically, its slow failover, policy routing deficiencies and the lack of ability to use more than one interdomain path simultaneously.

RON's attempt to address these issues by forming small groups of cooperating nodes (less than 50) which (both actively and passively) monitor the state of the (virtual) network links established between all of them, aggregate and exchange the collected data and update locally-stored routing tables accordingly [4]. Network metrics measured can be application-supplied and applications may choose routes to best suit their requirements.

Beyond application-specified performance and reliability improvements, cited applications include: high-performance proxy and network address translation services, multipath routing, policy routing and possibly even QoS guarantees.

The main problem with RON's is scalability: because they generally employ full mesh topologies in order to discover as many underlying routes as possible, the routing table of a RON instance grows at $O(N^2)$, where N is the number of participating RON nodes. Thus, in practice, RON's do not scale well and have been, in practice, kept at around 50 nodes. Moreover, because they actively ping each virtual link (i.e., all N^2 links), if the author-suggested model of one application per RON is applied, running many RON's simultaneously could be detrimental to the underlying network.

1.4.5.3 OCALA

OCALA [78] aims to allow users to access different overlays simultaneously, to allow hosts in different overlays to communicate with each other and to ease the burden on overlay developers by providing an API toolkit for tasks common to most overlays. For example, OCALA offers features allowing simultaneous use of services from *i*3 and RON networks. OCALA features a modular architecture divided into an *overlay convergence* layer which fits between the transport and network layers, and an *overlay-dependent* layer which contains details unique to respective overlays. It also attempts to provide users a means for easy overlay selection (in their current implementation via a faked top-level DNS suffix which).

1.4.5.4 X-Bone and GX-Bone

X-Bone [155] and GX-Bone [156] allow the configuration and management of routing overlay networks on standard Internet hosts. However, in sharing an address space with IP, in their use of existing IP routing mechanisms in packet forwarding, and in their mixing of best path computation and forwarding mechanisms, they are strongly tied to the old architecture and thus inherit its addressing and routing limitations. Furthermore, their scope precludes providing inherent means for path discovery or path selection.

1.4.6 Addressing Economic Issues

Several proposals have recently been made to address the shortcomings in the Internet billing model described in Section 1.3.3, and these systems generally address these shortcomings by allowing compensation from the enduser or originating network to follow the data along the entire path. Three salient examples exist and are described below.

1.4.6.1 Bill-Pay

Bill-Pay [42] is designed to leverage the existing scheme of bilateral contracts between operators while augmenting the network layer packet header to contain data allowing micropayments from users to flow along the path their packets traverse. When congestion occurs along the default path, micropayments are paid by traffic initiators to operators to route along a less congested path as compensation for the "good" network service provided by doing so. However, while the proposed system does provide for the better use of multiple paths, it does not provide a means for endhost or endnetwork path selection.

1.4.6.2 MINT

MINT (A Market for INternet Transit) [158] proposes the auction of path segments by network operators. Use of segments would be enabled via MPLS or similar means, and evidence is provided that modern routers have sufficient capacity to store lookup tables of sufficient size to allow such auctions at reasonable time frames. MINT would allow network operators to monetize enhanced routing services while providing both providers with enhanced traffic engineering and users with better path selection.

1.4.6.3 Path Brokering

In proposing *path brokering* [93], the present author suggests that *when the sole authority to select routes is placed in the hands of parties benefiting from their sale, a conflict of interest arises* and proposes the sale of end-to-end Internet paths via a third party *broker*. The broker is responsible for accepting compensation from consumers and passing it on to ISP's and thus acts as a natural advocate for consumers, similar to the role of travel agents in the airline inudstry.

1.5 Summary of Interdomain IP Routing Issues and Approaches

Sections 1.1 discussed the two major communications paradigms, circuit- and packet-switching, how and why the packet-switched paradigm came to dominate in the form of IP as well as the relative merits of the two paradigms; Section 1.2 provided a broad overview of networking with a specific focus on packet-switched networks, and IP's interdomain routing protocol implementation, BGP; Section 1.3 discussed shortcomings that have been raised in the literature with IP routing and BGP and Section 1.4 described some of the main approaches that the networking community has put forth to address these specific shortcomings.

Three things are of note. First, nearly all of the efforts discussed above at improving Internet policy, performance, reliability and even economic competition share a common feature—generally stated, they directly or indirectly provide some form of application-directed routing. Application-directed routing is routing wherein an application (and thus its users) are allowed choice in the paths its packets are to traverse. Examples include most of the technologies discussed above: ATM via circuit-switching and automated failover, IPv6 via source routing, NIRA via address-based paths, BANANAS and PoMo via direct path selection, Nimrod, 4D and RCP via their exposing of multiple selectable paths to applications and Pathlet Routing by allowing path stitching. Therefore, it is reasonable to assume that, whatever architecture is adopted in a future Internet, it will allow applications some manner of choice in the paths their packets are to traverse.

Second, no architecture exists to provide for *all* the requirements necessary to implement application-directed routing: a *data plane* to move packets across the network, a *control plane* to manage the data plane and an *application interface* to interact with both. As discussed above, a number of technologies exist which could be used to implement an application-directed data plane; examples include ATM, MPLS, IPv6 source routing and even overlay routing networks. However, while some control plane proposals do exist (e.g., 4D and PoMo) at present, no viable control plane exists to realize application-directed routing. Finally, no implementations or proposals exist for an application interface to application-directed networks; this is surprising since the needs of routing-aware applications (e.g., path scheduling, path performance monitoring, etc.) are more varied than that currently provided for by the current `socket`-based interface.

Third, even once data and control planes allowing some form of application-directed forwarding are available and an application interface is created for their use with applications, a number of serious problems remain unsolved. These include *path scheduling algorithms* for efficient use of multiple simultaneous paths under a variety of conditions, *path monitoring metrics and accompanying metric analysis algorithms* for ensuring paths in use are still meeting an application's requirements; multipath-aware *best path computation algorithms* to provide paths to applications; *means to prevent route oscillation* to prevent network instability while still allowing applications to select paths as freely as possible; and *means for allowing existing IPv4 and IPv6 applications to enjoy the benefits of path selection*, to name a few.

1.6 Problem Statement

Many application-directed routing deployment issues still exist before practical realization of application-directed routing can be achieved. However, it is difficult to perform the essential research required to overcome these challenges without a data plane, control plane and application interface usable on current networks to experiment with. As will be discussed in more detail in Section 2.2, *while large-scale testbed networks such as PlanetLab and Emulab exist which could aid in such research, and network toolkits exist to* *extend their data plane capabilities, either singularly or in combination, they currently do not provide an environment allowing research and development of application-directed routing.*

1.7 Thesis Statement

The *research presented herein provides an infrastructure for performing application-directed routing-related research and development on large-scale testbeds such as PlanetLab*; it includes a data plane supporting forwarding on a wide-variety of networks, a control plane for setup and management of the data plane and an application interface allowing interface with data and control planes as well as research and development of new applications and transport layers as well as experimentation with existing applications and protocols. Moreover, *a usage case is described wherein this infrastructure is used in the development and verification of a practical means for using unmodified TCP/IP on endhost-directed networks*.

1.8 Dissertation Overview

This balance of the dissertation is constructed as follows. The next chapter presents the SORA multipath virtual network layer framework, which enables application-directed routing research and development on current networks, including large-scale testbeds such as PlanetLab. Chapter 3 presents a usage case of SORA in the investigation of the performance of a scheme to allow bandwidth aggregation of unmodified TCP traffic via dispersion of packets over multiple paths, even under heavy packet reordering. Chapter 4 concludes the dissertation and the appendices provide further details regarding SORA.

CHAPTER 2

THE SORA MULTIPATH VIRTUAL NETWORK LAYER

2.1 Introduction

Application-directed routing architectures—those allowing applications some manner of choice regarding the network path their data packets are to traverse—have been proposed as an approach to addressing many of the current Internet's most difficult problems, including policy [170], performance (QoS) [172] and economics [93].

Internet users and operators currently have no real control over the specific path their outbound packets are to traverse, the routing choices being left completely to each network operator to determine. If the default selected path does not suit a user or application, they have little recourse. Moreover, there is no easy way for a user or operator to know what the default selected path actually is.

Application-directed routing architectures provide solutions to these problems by giving applications a choice in route selection; applications can then select paths to meet given policy and performance goals; moreover, schemes have recently been proposed to foster Internet competition by using paths or subpaths as a basis for selling Internet access [93, 158, 59].

Despite its advantages, however, no infrastructure currently exists to allow application-directed routing. Moreover, many open problems exist which prevent the realization and utility of such an infrastructure such that even if the Internet *were* now able to provide application path selection, its adoption would not be possible. These open problems include, for instance: *path scheduling algorithms* for efficient use of multiple simultaneous paths under a variety of conditions, *path monitoring metrics and accompanying analysis algorithms* for ensuring paths in use are still meeting an application's requirements; multipath-aware *best path computation algorithms* to provide paths to applications; *means* *to prevent route oscillation* to mitigate network instability while still allowing applications to select paths as freely as possible; and *means for allowing existing IPv4 and IPv6 applications to enjoy the benefits of path selection*, to name a few.

While large-scale testbed environments now exist which could provide a platform for realistic experimentation, there exists no framework for application-directed routing—that is, there exists no *data plane* to forward packets based on application specifications; no *control plane* to establish network paths to be selected by applications; and no *application programming interface* by which applications and novel transport layers might select their paths. This inability to perform application-directed experiments, in turn, hinders research and development of these open problems.

The design and implementation of such a framework for application-directed routing brings special requirements when applied to current networks and testbed environments: (1) an efficient, application-directed data plane usable with existing routers, network stacks, and endhost network software and applications; (2) a control plane allowing convenient management of large-scale experiment networks; and (3) an application interface allowing ease of development and experimentation.

The sections that follow present the design and implementation of a framework which provides for these requirements and has already been used in research and development addressing application-directed routing problems. They are organized as follows: the next section discusses related architectures and implementations; Section 2.3 provides a more detailed overview of the infrastructure described herein and its requirements; Sections 2.4, 2.5, and 2.6 describe the design and implementation of the data plane, control plane and application interface, respectively. Section 2.7 provides an analysis of the performance of these three components. Section 2.8 briefly discusses a few examples of how this infrastructure has been used. Section 2.9 discusses future work and Section 2.10 concludes.

2.2 Related Work

A number of architecture proposals exist which could someday provide a basis for an application-directed data plane and/or control plane (e.g., NIRA [171], more recent work by Yang et al. [170] and PoMo [15]). However, in examining related work, since the scope of this research is limited to operation on current networks, only work which is readily implementable and deployable on current networks and testbeds are considered. Below, are summarized the most relevant of these and compare and contrast the features of each with the system described herein.

2.2.1 IPv4 and IPv6 Source Routing

"Source routing" describes any routing means which allows all or a part of a route to be specified at the packet source; it has long been proposed [152] for allowing greater application routing control and is a part of both IPv4 [125] and IPv6 [37] specifications. IPv4 and IPv6 source routing allows source endhosts to specify some or all of the network interface addresses a packet should traverse.

Unfortunately, for security reasons, it is widely disabled in IPv4 and therefore currently unusable. Moreover, even if, as expected, IPv6 does eventually make source routing available, its reliance on interface addresses as a means for specifying a route makes it difficult to select from a variety of potential links between two routers; moreover, it is also not robust against IP address changes. Finally, no control plane means is currently provided for applications to discover what network topology exists or to compute paths and only a basic application interface exists allowing path specification.

2.2.2 BANANAS

BANANAS [81] proposes a modification to current Internet routers which allows applications on endhosts to specify the individual routers their packets are to traverse. BANANAS details the construction of a data plane for application-directed routing by allowing the selection of the outbound interface or tunnel for each router on the path.

However, BANANAS does not suggest a means for hiding the complexity using various interfaces and tunnel types simultaneously.

The data plane described herein is similar to that of BANANAS in that both use packet control data to select the path a packet is to traverse; it is different in that the data plane described herein generalizes the concept of network interface selection to include a variety of tunnels. Moreover, BANANAS provides no details regarding a control plane and uses the current `socket`-based interface as its application interface.

2.2.3 Testbeds: PlanetLab, VINI, CoreLab, ...

Research network testbeds such as PlanetLab [27], CoreLab [115], OneLab [118], GENI [58] and Emulab [162] provide a collection of computing and network hardware resources designed for experimentation. They are therefore a basis for the research and development of next-generation networks such as those described above. However, current testbeds provide data and control planes only for IP-based forwarding; they currently do not provide explicit support for application-directed routing.

Network toolkits—software packages which replace or extend the functionality of the data and control planes and application interfaces provided by current networks—do provide for a basis for implementation of application-directed forwarding. They do so by forming *virtual networks* atop existing networks, with testbed computing resources acting as routers and the networks between them acting as links. Thus, said *virtual networks* are logical networks comprised of network tunnels between existing network computers; such networks are also often called *overlay networks*, since they are *overlaid* atop the existing network. A *network tunnel* is the use of an existing network protocol for transport of data of another protocol; examples include IPv6 in IPv4 for enabling IPv6, IP-in-IP for enabling virtual private networks, and the like. A number of such toolkits which enable virtual networks via tunnels use have been developed, as discussed below.

2.2.4 RON

Resilient Overlay Networks (RON's) [4] are routing systems designed to address some of the deficiencies in current wide-area Internet routing, particularly with respect to poor routing performance and reliability. RON's attempt to address these issues by accessing routes not selected by the Internet. They do this by forming virtual (overlay) networks atop the Internet using endhosts as routers and tunnels between these endhosts as virtual links. They then run routing protocols (e.g., RIP [104] or OSPF [112, 32]) atop the virtual network links, monitor their state and use the best virtual network path for transport, thereby obtaining improved performance.

While the original description [4] does not suggest its use for application-directed routing, RON's could be adapted to this use by adopting an approach like that specified in BANANAS; however, until now, this has not been done. Moreover, RON's provides no details regarding a control plane and uses the current `socket`-based interface as its application interface.

2.2.5 OCALA

OCALA [78] aims to allow users to access different overlay networks simultaneously, to allow hosts in different overlays to communicate with each other and to ease the burden on overlay network developers by providing an API toolkit for tasks common to most overlays. While the original description [78] does not suggest its use for application-directed routing, OCALA could be adapted to provide data plane services similar in spirit to BANANAS; however, to date this has not been implemented. Moreover, OCALA provides no details regarding a control plane and does not provide for the path scheduling, path monitoring or path monitoring feedback needs of application-directed networks via an application interface.

2.2.6 X-Bone & GX-Bone

X-Bone [155] and GX-Bone [156] allow the configuration and management of routing overlay networks on standard Internet hosts. However, they provide no explicit means for application-directed forwarding, no control plane for link or path discovery and no application interface their use.

2.2.7 Application Interfaces: SCTP, MPTCP, Ingress & Egress

In this work, an *application interface* will be defined as the means by which applications interface with network data and control planes. For instance, `socket` and its associated methods and data structures comprise the application interface in the case of IPv4 and IPv6.

Because the Internet provides no means for applications to select the paths their packets are to traverse, focus for improving performance has typically been on *interface selection*. Like path selection, interface selection allows an application to use several network interfaces simultaneously to improve performance. However, interface selection differs from path selection in that interface selection only allows selection of the outbound and inbound network interfaces, whereas path selection provides much finer-grained selection.

2.2.7.1 SCTP

Nevertheless, the `socket` interface has been extended to include support for interface selection via the Stream Control Transport Protocol (SCTP) [148]. SCTP is the current Internet standard transport protocol allowing multiple network interfaces to be used simultaneously in order to provide enhanced performance. While its use of multiple interfaces originally targeted failover, recent efforts have been made at using SCTP for concurrent transport [73].

However, interface selection schemes like that of SCTP suffer two main disadvantages with respect to path selection. First, the Internet is known to have excellent path diversity [96], but because it effectively offers only selection of the first and last hops of a path, *interface selection alone cannot fully exploit this path diversity*. Second, interface selection cannot provide any ultimate guarantee that the paths selected by its interfaces are disjoint and therefore *cannot guarantee actual performance gains*.

2.2.7.2 MPTCP

Accordingly, research has been conducted with the aim of augmenting SCTP with a path selection means [99]. Moreover, more recent efforts have been aimed at a complete redesign of TCP [11]. However, the lack of an application-directed data plane for use with SCTP or newer transport protocols makes this difficult to implement. Moreover, SCTP still provides *no means for monitoring of metrics such as* per-path *packet loss, latency or of feedback of such information for use in path scheduling*.

2.2.7.3 Ingress & Egress

The use of existing protocol traffic with application-directed networks has also recently been studied [26] with the aim of bringing the benefits of application-directed routing to such protocols. Such schemes must have some means for sending and receiving existing traffic via multiple paths in order to study the effects of multiple path use. However, there exists *no convenient means for ingress or egress of traffic from existing multipath-unaware protocols such as TCP or UDP on to application-directed networks*. Virtual network *ingress* is defined as the preparation of existing protocol (e.g., TCP/IP or UDP/IP) packets for input onto and transmission via a virtual network. Similarly, virtual network *egress* is defined as the preparation of ingressed packets for exit from a virtual network and transmission via an existing network.

2.2.8 Summary

While tunnel-based technologies exist by which an application-directed data plane could be implemented on current networks, there exists no framework to ease research and development when interfacing with tunnel-based networks, particularly those comprised of a variety of tunnel types. Moreover, no control plane exists to allow such networks to be conveniently configured for use in large-scale experiments on current testbeds.

Further, no application interface exists to provide for per-path scheduling, monitoring and performance data feedback or to provide for application-directed forwarding research and development on current networks.

2.3 Overview

The primary objective of SORA is to provide an R&D infrastructure for application-directed routing on current networks: particularly, large-scale testbeds. The core goal is an immediately usable solution to the problem of how to conveniently develop and test both existing and future applications and transport protocols on application-directed routing networks using the current network infrastructure.

Application-directed routing is *a packet routing method, whereby applications are given access to an arbitrary number of paths and are made fully responsible for enacting a routing policy as well as making performance decisions by monitoring path performance metrics.*

The general requirements of this infrastructure are a *data plane* for executing application-directed forwarding, a *control plane* for management of said data plane, and an *application interface* to the application-directed data and control planes. Thus, SORA acts as a toolkit for extending the data plane capabilities of existing networks.

2.3.1 Data Plane Requirements

The purpose of the data plane is to provide for application-directed packet forwarding on current networks via tunneling. There are three basic requirements for the data plane.

Because many applications are not implementable without sufficient forwarding performance, the *first data plane requirement is forwarding performance so as not to be a bottleneck* on current testbed networks.

The *second data plane requirement is minimal network setup burden and router resource waste by* optionally *supporting forwarding for multiple testbed experiments simultaneously.*

The *third data plane requirement is ease of development of upper layers* which use the data plane, including the ability to conveniently specify arbitrary per-packet control data, including at least the path the packet is to traverse and feedback of monitored performance information.

2.3.2 Control Plane Requirements

The purpose of the control plane is to facilitate configuration and management the data plane outlined above: to provide for link instantiation on routers, link discovery, link metric discovery and path computation. There are three general requirements for the control plane.

The *first control plane requirement is convenient data plane link, link metric and path discovery*. Link, link metric and path discovery refer, respectively, to the means by which an entity queries for and obtains data plane links inbound to and outbound from a given entity, metric data regarding said links and paths through a given virtual network. A full Internet-scale means for performing these three tasks is beyond the scope of the current work; what is required is a convenient means for use with experiments on both large and small virtual networks.

The *second control plane requirement is convenient router link instantiation on all virtual network routers*. Before a link from one entity to another may be used, the underlying tunnel must be set up; for example, in the case of a UDP socket-based tunnel, a socket must be allocated.

The *third control plane requirement is supporting setup and management of forwarding for multiple testbed experiments simultaneously*. When multiple virtual network data planes are run simultaneously, sharing of router and link resources between data planes allows for a reduction in router resources consumed because only one router instance is required for all virtual networks; it also allows for a reduction in total router link instantiation time because existing, shared links need not be instantiated. *However, this shared testbed mode should be* optional*; the framework must generally be usable as a standalone infrastructure*.

2.3.3 API Requirements

The purpose of a software interface for application-directed networks is to allow applications to interface with the control plane (e.g., to obtain link and path data) and send and receive data via the data plane. There are two general requirements for the application interface.

The *first API requirement is ease of application-directed routing-aware application development and experimentation*. The goal of this framework is to aid in the research and development of application-directed forwarding applications and transport layers. Accordingly, for example, the application interface should allow for easy swapin and swapout of different path scheduling and monitoring configurations and include a library for performing common tasks.

The *second API requirement is usability with existing network applications and protocols*. While new application development is one goal, the framework must also be usable with existing protocols and applications in order to aid testing of their performance on application-directed networks and to allow comparison with new protocols and applications.

The data plane, control plane and application interface are the respective subjects of the three sections that follow.

2.4 Data Plane

As described in Section 2.3.1, the three data plane requirements are: (1) forwarding performance so as not to be a bottleneck on current testbed networks, (2) optional forwarding for multiple experiments simultaneously and (3) ease of upper layer development.

2.4.1 Challenges

The first challenge is providing for application-directed forwarding on existing networks *while also* providing forwarding performance that is not a bottleneck on them. However, by adopting the well-known technique of using virtual network composed of tunnels (e.g., as discussed with respect to RON's), near native forwarding performance can be obtained while using a variety of routing schemes [4]. Unlike previous networks, however, each tunnel in the virtual network is selectable, as is described below in Section 2.4.2. Moreover, use of virtual networks atop existing networks also provides the advantage of allowing forwarding for multiple virtual network instances simultaneously, meeting the second requirement.

The challenge in the construction of a router to implement the virtual network data plane lies in efficiently supporting a variety of testbed environments; specifically, in order to maximize performance, the router should ideally support operation both when no kernel-level implementation is possible and when kernel-support is possible. This challenge is addressed by an implementation of the data plane structures in a shared library `libsora` and a router implementation based on the Click modular router framework [84].

However, two challenges remain related to ease of development (third goal) on tunnel-based virtual networks.

First, while it is possible to use tunnels that form a virtual network directly for transport, simultaneous use of multiple tunnels of different types (e.g., UDP/IP encapsulation, TCP/IP encapsulation, IPv6 encapsulation, etc.) makes development more difficult; therefore, the challenge to easing development on such networks is to *provide a unified network interface to underlying tunnels*, which *hides their usage details from upper layers*, including tunnel management (setup and teardown), packet send, receive and specification of a list of tunnels in order to form a path. This challenge is addressed by the link abstraction.

Second, operation with various link types (e.g., UDP/IP and IPv6) requires control (e.g., path) and payload data to be specified in different locations with respect to each other. For instance, a UDP/IP encapsulation may place control data contiguously above payload data, however, an IPv6 encapsulation-based tunnel, may place control data in an option extension, while the payload data is positioned, non-contiguously, in the IPv6 payload.

Moreover, while it is possible for each application to manage its own control data (options) format, forcing each application to independently implement its own options processor makes development more difficult: specifically, it becomes difficult to share and reuse code between implementations when options are specified in different formats.

Thus, the second challenge is providing a structure which allows: (1) *control and payload data to be positioned independently* for use by the link abstraction; (2) *application-directed control data (e.g., path and monitoring feedback data) data to be specified compactly*; and (3) *arbitrary new application-defined types to be added*. This challenge is addressed by the packet abstraction.

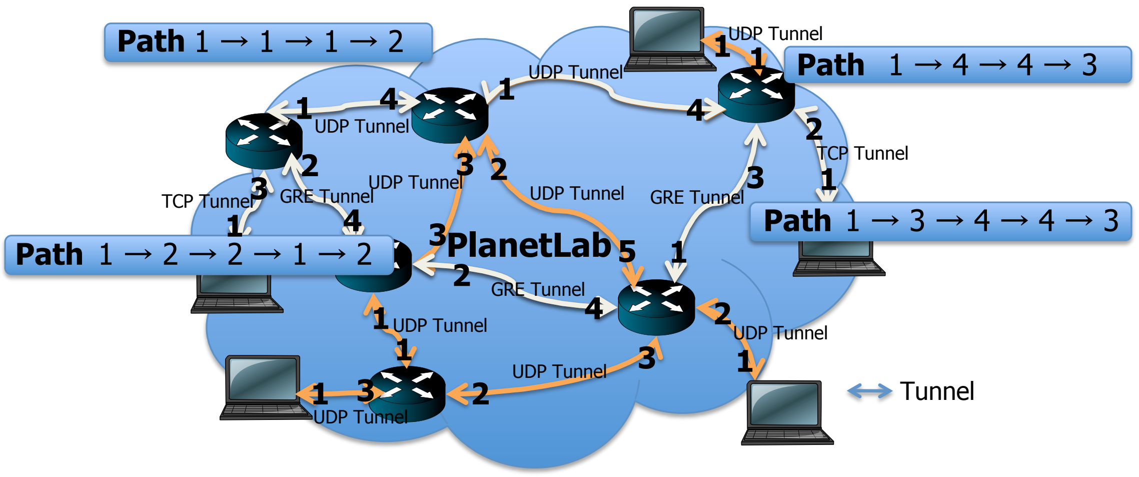


Figure 2.1: Application-Directed Testbed-Based Data Plane With Selectable Paths Via Numbered Tunnels

2.4.2 Approach

First, a general structure is defined wherein a number of virtual network data planes may simultaneously operate on a given network. Here, *virtual network identifiers* provide the means for differentiation between virtual network data planes. Virtual network identifiers are implemented as 32-bit values.

As depicted in Figure 2.1, each virtual network has a number of entities—virtual network routers or endhosts, which have one or more inbound or outbound links, as described below. *Entity identifiers* provide the means for identifying an individual entity on a given virtual network, independent of the many network interface addresses (e.g., IPv4 and IPv6) it may have. Entity identifiers are also implemented as 32-bit values. Endpoints are defined as sources and destinations of packets; routers are defined as entities dedicated to virtual network packet forwarding.

As discussed later, these identifiers are used mainly for link lookup (to uniquely identify a link) and path computation (to uniquely identify entities and links in a virtual network graph).

2.4.3 Link Abstraction

The link abstraction is defined as a data structure and associated methods which provide a unified interface to underlying tunnels for easing development of upper layers by hiding tunnel usage details. Links provide packet transport from one virtual network entity to another. All links are unidirectional—even if the underlying transport means is bidirectional, packets travel in only one direction. A *path* is defined as a series of virtual network links connecting two entities; typically it is specified using the corresponding list of link identifiers.

2.4.3.1 Design

The link abstraction provides upper layers a unified interface to setup, teardown, read packets from and write packets to a given tunnel. As depicted in Figure 2.2, a link essentially comprises virtual network metadata to identify it on a given virtual network: a virtual network ID, source and destination entity IDs and a link ID; a link is uniquely identified by the three-tuple: <virtual network ID, source entity ID and link ID>; as discussed in Section 2.5.4, it may also comprise a metric. Access to the underlying tunnel is abstracted via methods for setup, teardown and packet send and receive. Setup and teardown methods establish or remove an inbound or outbound tunnel. Send and receive methods perform input and output of instances of the packet abstraction discussed below.

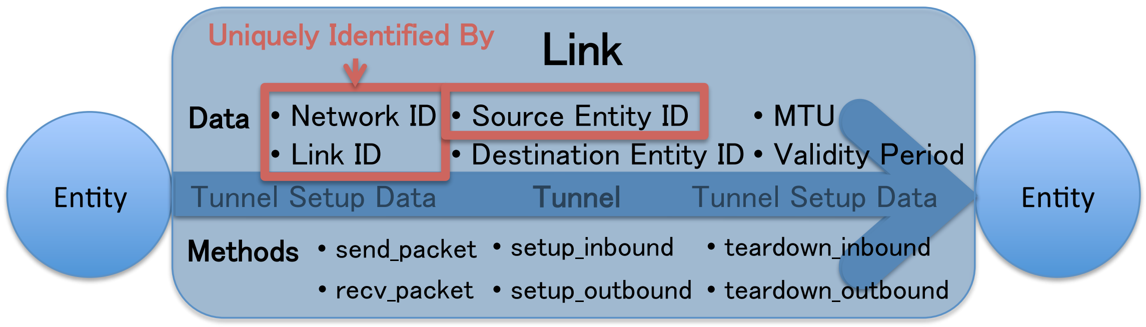


Figure 2.2: Link Abstraction

2.4.3.2 Implementation

As depicted in Figure 2.3, the link abstraction interface is defined by the C++ pure virtual class `SoraLink`. Pure virtual classes cannot be instantiated directly: each given link type implementation inherits the `SoraLink` interface and implements a basic set of methods for the tunnel; specifically: setup of inbound and outbound links (respectively, `setup_inbound` and `setup_outbound`), teardown (`teardown_inbound` and `teardown_outbound`), send (`send_packet`), receive (`recv_packet`), setting of blocking and non-blocking read and write (respectively, `set_blocking` and `set_non_blocking`), determination of the location of the packet header and payload within the data of a given transport means (respectively, `get_sora_header` and `get_sora_payload`) and retrieval of the underlying tunnel file descriptors (`get_fd`).

For instance, as is also depicted in Figure 2.3, the `SoraUDPIPLink` class inherits the `SoraLink` interface and implements methods to operate on UDP/IP sockets. Specifically, `SoraUDPIPLink` uses setup data (its attributes) comprising source and destination ports and IP addresses to open a datagram socket, the file descriptor of which it stores (in `socket_fd`). To send a packet, it obtains the header and payload from the packet, via the means described below, encapsulates the header at the top of the UDP/IP packet payload, followed by the payload. To receive a packet, it parses the packet header, using the means described below, then obtains the payload and returns a packet structure formed from these.

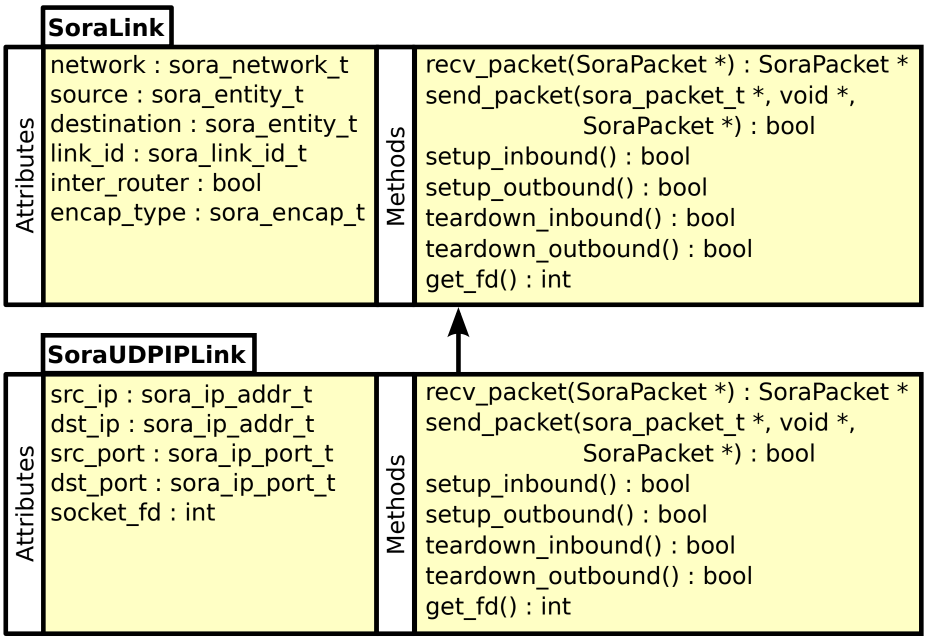

SoraLink
Attributes
network : sora_network_t
source : sora_entity_t
destination : sora_entity_t
link_id : sora_link_id_t
inter_router : bool
encap_type : sora_encap_t
Methods
recv_packet(SoraPacket *) : SoraPacket *
send_packet(sora_packet_t *, void *, SoraPacket *) : bool
setup_inbound() : bool
setup_outbound() : bool
teardown_inbound() : bool
teardown_outbound() : bool
get_fd() : int
SoraUDPIPLink
Attributes
src_ip : sora_ip_addr_t
dst_ip : sora_ip_addr_t
src_port : sora_ip_port_t
dst_port : sora_ip_port_t
socket_fd : int
Methods
recv_packet(SoraPacket *) : SoraPacket *
send_packet(sora_packet_t *, void *, SoraPacket *) : bool
setup_inbound() : bool
setup_outbound() : bool
teardown_inbound() : bool
teardown_outbound() : bool
get_fd() : int

Figure 2.3: Classes `SoraLink` and `SoraUDPIPLink`

Other links may be implemented similarly. For instance, TCP/IP sockets may also be used for encapsulation; the `SoraTCPIPLink` class has precisely the same interface as `SoraUDPIPLink`. It also uses its attributes to set up a socket and stores the file descriptor for reading or writing. To send a packet, the TCP/IP link instance obtains header and payload and sends them similarly to the UDP/IP link instance. However, to receive a packet, since the underlying transport is a stream, the read method first decodes the header using the means described below, determines the payload size, and then reads the entire payload out before returning the completed packet instance.

A path is formed solely from link data; it comprises a virtual network identifier, source and destination entity identifiers and a series of link identifiers specifying the next hop link outbound from each successive entity on the path from the source entity to the destination. It has the same format as that described with respect to the control plane in Section 2.5.3.2 and in Figure 2.8.

The methods of the link abstraction provide a unified network interface to underlying tunnels and hide transport details from upper layers, while still allowing new link types to be implemented easily.

2.4.4 Packet Abstraction

The packet abstraction is the data structure of the data plane by which data payload and control data is read from and written to a link instance. Its purpose is to means for: (1) control and payload data to be positioned independently for use by the link abstraction; (2) application-directed control data (e.g., path and monitoring feedback data) data to be specified compactly; and (3) arbitrary new application-defined types to be added.

2.4.4.1 Design

As depicted in Figure 2.4, the packet abstraction comprises a set of options and a buffer for payload data. The option set stores a fixed number of options. Each option comprises an *option type*, *option data*, and a set of *option handlers* which define read and write operations on the option data. The option type uniquely identifies the option within a given option set. The option data comprises the actual data to be transmitted in the option and may take any format desired. The option handlers are functions which perform requisite option handling operations; there are five such operations to be defined for all option data instances: *initialization* to set a starting value to the data, *length obtainment* to determine the length of the option data, *serializing* to prepare the option data for transmission (e.g., conversion to network byte order), *marshaling* to prepare the option data for reading by the host (e.g., conversion to host byte order), and *character conversion* to convert the option data to a user-readable character string for printing.

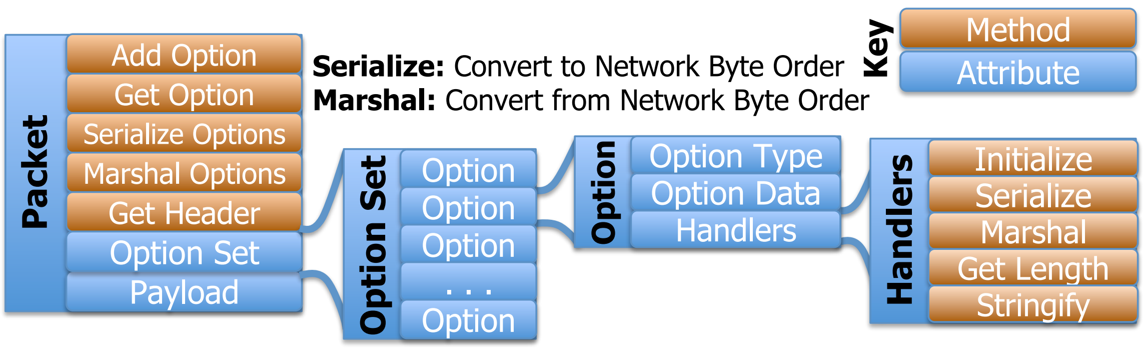


Figure 2.4: Packet Abstraction

The framework defines several basic option types and handlers, including those for storing virtual network ID, source and destination entity IDs and path data. Applications define new types by assigning an option type and defining the five handlers described above. Further details regarding the implementation specifics of this structure, including its header representation, are discussed below.

2.4.4.2 Implementation

To implement the abstraction depicted in Figure 2.4, structures were adopted which minimize both the space consumed in the packet header as well as the data copy during both read and write link operations. The packet header is network control data included when sending a packet on a virtual network link.

The challenge in minimizing header space consumed lies in reducing the amount of redundant data contained in the header: packet options typically include both option type and length information; however, inclusion of these for each option is redundant. First, *type information for all options contained in the entire packet header can be summarized in a single bit vector*, wherein each bit indicates the presence or absence of a given type. The presence or absence of each option in the packet is then encoded by a single bit in the bit vector. Second, as discussed above, *length information for any given option can be obtained via the pre-defined handler* for each option type, which are called with the given option data.

```
0       1       2       3       4       5       6       7
0123456701234567012345670123456701234567012345670123456701234567
+-------+-------+-------+-------+-------+-------+-------+------+
|            Checksum           |Payload Length | Header Length|
+-------+-------+-------+-------+-------+-------+-------+------+
|       Option Bit Vector       |       [... Options ...]      |
+-------+-------+-------+-------+-------+-------+-------+------+
```

Figure 2.5: Packet Header

As depicted in Figure 2.5, the packet header contains a 32-bit (CRC-32) checksum (for error detection over lossy links), 16-bit payload and header lengths (to allow quick determination of the both the packet and header lengths), an option bit vector and option data of variable length. The option bit vector is 32 bits long, thus supporting a maximum of 32 options (per communicating endpoint pair). Option data for each option is concatenated in order, and is parsed using the length obtainment handler for each option present.

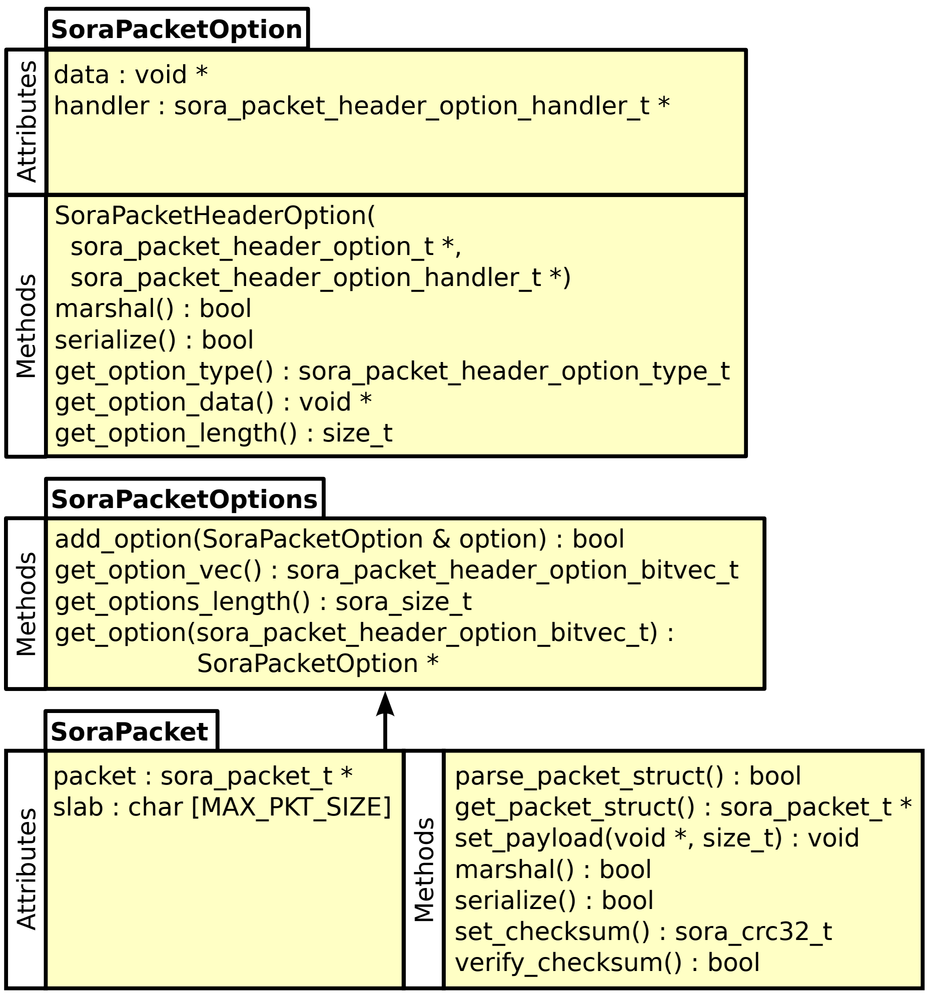

SoraPacketOption
Attributes
data : void *
handler : sora_packet_header_option_handler_t *
Methods
SoraPacketHeaderOption(
sora_packet_header_option_t *,
sora_packet_header_option_handler_t *)
marshal() : bool
serialize() : bool
get_option_type() : sora_packet_header_option_type_t
get_option_data() : void *
get_option_length() : size_t
SoraPacketOptions
Methods
add_option(SoraPacketOption & option) : bool
get_option_vec() : sora_packet_header_option_bitvec_t
get_options_length() : sora_size_t
get_option(sora_packet_header_option_bitvec_t) :
SoraPacketOption *
SoraPacket
Attributes
packet : sora_packet_t *
slab : char [MAX_PKT_SIZE]
Methods
parse_packet_struct() : bool
get_packet_struct() : sora_packet_t *
set_payload(void *, size_t) : void
marshal() : bool
serialize() : bool
set_checksum() : sora_crc32_t
verify_checksum() : bool

Figure 2.6: `SoraPacket` Class

The packet abstraction acts as an interface for packing and parsing application-specified option data to and from this packet header format. As depicted in Figure 2.6, an option set comprises an array of 32 options, an option bit vector and handlers for all possible options; the handlers allow for packed option sets to be sent and for newly-received option sets to be parsed. An option instance comprises a type, a pointer to its option data, a pointer to its handlers, as described above.

To send a packet, an application sets any payload data to be sent and adds any number of options via the add option method. The packet is then ready to be passed to the link (i.e., via the link send_packet method): the link calls the options serialization method, which calls serialize on each option to prepare it for transport; the link then calls the `get_packet_struct` method in order to obtain a packed representation of the options. The link then places the header and payload into their appropriate places for transport on a given link and sends.

To receive a packet, a link reads the packet struct (header) and payload data out from the given tunnel directly into the buffer (the `slab`) provided by the packet, identifies the location of the header and payload data and sets these in the packet structure. It then calls the options marshal method of the packet. This method first identifies the start position of the option data of each option present and stores this information in the position in the options array corresponding to each option. It then calls the marshal method of each option to prepare the data of each option for use. The packet is then ready for reading by upper layers via the options array.

The packet abstraction separates control and payload data for use by the link abstraction and provides a simple interface for new options to be specified easily. Further, use of a single buffer (slab) and pointers to options and payload data allows option and payload data to be copied just once (from the network buffer), thereby minimizing overhead.

2.4.5 Click-Based Router and Shared Library

The link and packet classes as well as all supporting methods and data types described above (and the application interface described in Section 2.6) are implemented in C++ and contained in a library called `libsora`.

The purpose of the data plane router is to support fast data plane forwarding, both when no kernel-level implementation is possible (e.g., PlanetLab) and when kernel-support is possible (e.g., Emulab). The Click modular router framework [84] allows for the creation of packet routers by linking together simple packet processing modules called element classes (elements), which are implemented as C++ classes. The task of a Click element is to accept packets on or more input ports, perform a given operation on the packet and output the packet on or more output ports. Click runs on Linux and BSD and contains libraries for processing most common transport, network and data link layer packet header formats (e.g., Ethernet, IPv4 and IPv6, UDP/IP, etc.). Click also supports both kernel-level and user-level operation with little configuration modification and has been shown to provide high performance in both modes [84].

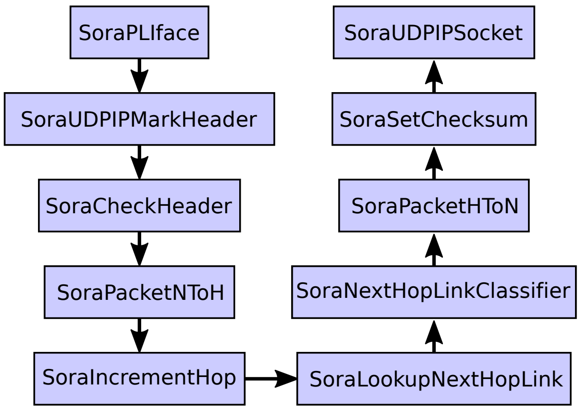


Figure 2.7: Basic Router Element Chain Configuration

2.4.5.1 Design & Implementation

The Click router implementation currently supports operation on PlanetLab and Emulab using UDP/IP-based encapsulation, but is designed to be extensible to include other types; it is implemented via a set of simple packet processing modules, each of which based on `libsora`.

These modules include the following essential processing tasks: location of a packet header within a received packet on a given link (`SoraUDPIPMarkHeader`), verification of the checksum on the header (`SoraCheckHeader`), incrementing of the hop counter (`SoraIncrementHop`), lookup of the next hop link in the next hop link table (`Sora-LookupNextHopLink`), determination of the next hop link type of the determined link (`SoraNextHopLinkClassifier`) for support of future link types, encapsulation and sending on the selected next hop link (`SoraUDPIPSocket`), and serializing and marshaling of packet options (`SoraHToN` and `SoraNToH`).

As depicted in Figure 2.7, these modules can be chained and formed into a packet router capable of receiving a packet on a given input link and outputting a packet on a given output link, as determined by the control data in the parsed packet header. It should be noted that `SoraUDPIPSocket` is capable of handling multiple link instances. Moreover, in the future, outbound link types other than UDP/IP are handled via `SoraNextHopLinkClassifier`, which selectively outputs packets to respective ports based upon their outbound link type.

The configuration of outgoing links (for lookup in `SoraLookupNextHopLink`) is performed via the control plane and is discussed in the next section. The performance of the data plane router (and thus the processing library `libsora`) in different environments is discussed in detail in Section 2.7.

2.5 Control Plane

As described in Section 2.3.2, the three requirements for the control plane are: (1) convenient link, link metric and path discovery, (2) convenient router link instantiation and (3) minimization of network setup burden and router resource waste by *optionally* supporting setup and management of forwarding for multiple testbed experiments simultaneously.

2.5.1 Challenges

Three challenges in the design and implementation of an infrastructure arise corresponding to these three goals.

First, the challenge in creating a *link, link metric and path discovery means* lies in *efficiently supporting the needs of both smaller and larger experiments*. While copying flat files containing link and path descriptions may be ideal for smaller experiments comprising only a few routers and endhosts, this task quickly becomes burdensome virtual networks comprise thousands of routers, many thousands of links and many endhosts all requiring paths through the network.

Second, the challenge in providing for *convenient router link instantiation* lies in *reducing the total amount of effort required to instantiate all the links of a virtual network on all of its routers*. Similar to the case with link, link metric and path discovery, manual link instantiation is not an option when dealing with hundreds or thousands of routers; and some means for automation is required.

The third challenge relates to operation in the shared mode described above where forwarding is being performed for multiple experiment networks simultaneously. However, the challenge is more subtle: when an experimenter installs application-defined routing software on a testbed, *they* are responsible for ensuring testbed resources are used according to the acceptable use policy of the testbed; link installation is done at their discretion and therefore it suffices for the framework to provide experimenters a simple password-based authentication which the experimenter may use to ensure only their approved links are instantiated.

However, when a shared virtual network routing service is provided, *the shared virtual network routing service itself must guarantee that testbed resources are not misused*. Nevertheless, *the service must also still support ingress of traffic onto and egress of traffic off from external networks*. Therefore, *the challenge for providing sharing of testbed resources between multiple virtual networks simultaneously lies in allowing for ingress and egress of network traffic from testbed-external links while still ensuring only authorized management and use of testbed resources*.

2.5.2 Approach

The control plane is implemented via a programming library, which includes link, path and supporting data type implementations as well as methods for their manipulation. The control plane implementation addresses the respective challenges above via: (1) *RPC-queryable link database, link metric and path computation services*; (2) an *RPC-queryable link instantiation service*; and (3) a means for providing testbed router and link sharing by *pre-instantiating all testbed links and centrally authenticating all other (testbed-*

external) links, thus allowing approved links to be instantiated by experimenters directly and avoiding the need for any central link instantiation or storage.

2.5.3 Ruby-Based Control Plane Library

The purpose of the control plane library is to facilitate creation and manipulation of all control plane-related data structures and services.

2.5.3.1 Design

To facilitate convenient virtual network construction and modification, while allowing inheritance similar to what is possible with the C++ `libsora` implementation, an object-oriented scripting language, Ruby was chosen for the library implementation. The library contains implementations of the main `Link` and `Path` structures and allows them to be read from and written to character strings for storage and retrieval.

2.5.3.2 Implementation

The fundamental classes of the library are `Link` and `Path`, as depicted in Figure 2.8; these implement essentially the same structure as in the data plane (though they lack methods for setup, teardown, send or receive).

Moreover, the library also contains a multipath path computation engine module implementation. This module allows a set of `Link` objects to be read in directly by the Ruby interpreter and for a shortest path set to be computed through the given link set. Because the module is implemented in C++, it does not suffer from the overhead of the interpreter. The module computes multiple paths by: (1) running Dijkstra's algorithm to compute the shortest path between two endpoints; (2) removing the edges of the shortest path; and (3) repeating these steps until the requested number of paths is obtained or until there exists no such path.

The library forms the basis for implementation of the discovery, link instantiation and shared resource approaches described below.

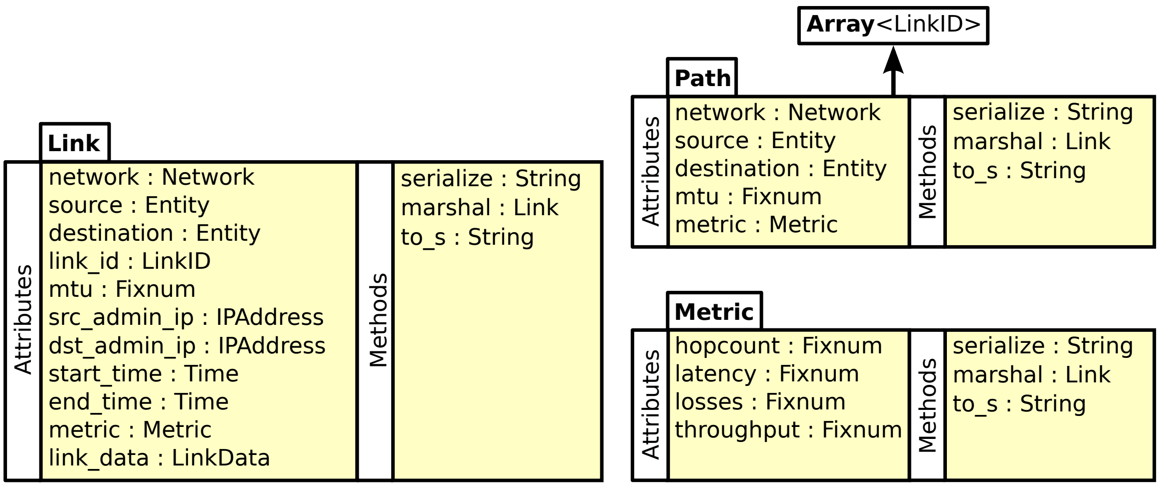


Figure 2.8: Fundamental Control Plane Library Classes

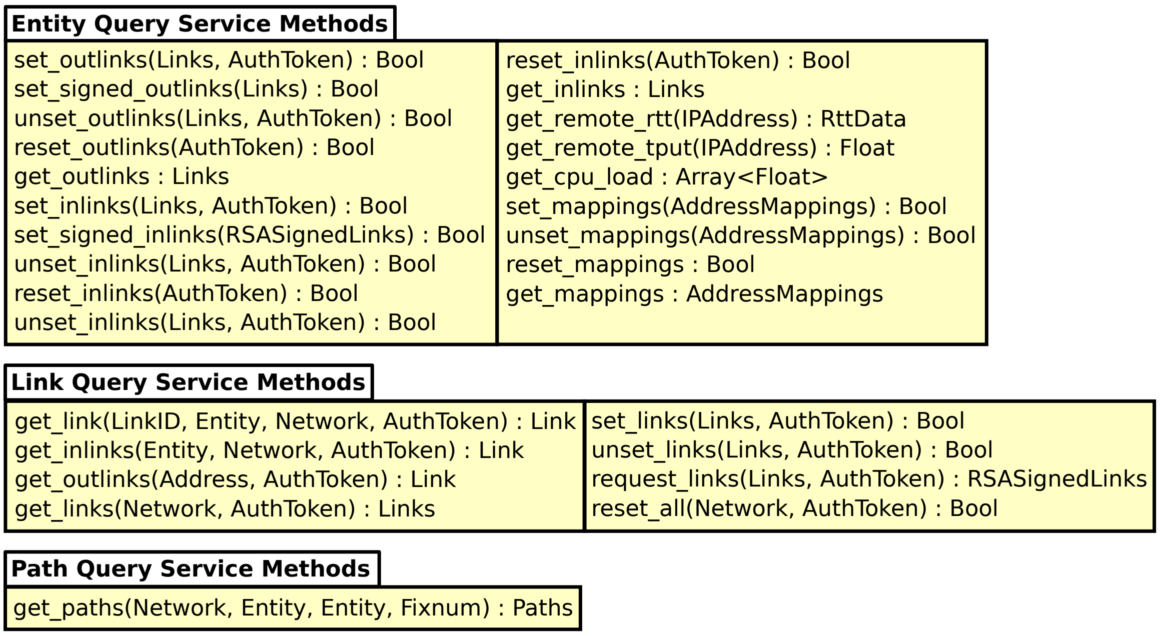
2.5.4 Link, Link Metric and Path Discovery

Link, link metric and path discovery refer to the means by which these objects are queried for and thus obtained for use by virtual network entities. The components which implement link, link metric and path query build on and extend the fundamental data structures of the control plane library. As depicted in Figure 2.9, the link metric query methods allow an entity to obtain link metric data, which is defined as data describing one or more aspects of a link instance that may be criteria for use in path set computation. It should be noted that, in application-directed routing, link metric data are *not* used by applications; applications are concerned with the monitoring and use of *paths*, not individual links.

The link query service and its methods allow an entity or setup host to obtain *link data*—data representing a link instance—and for authorized users to insert, update or remove link data. The path query service method allows an entity to obtain paths by which to send and receive packets via a virtual network.

2.5.4.1 Design

For all queries discussed above, an RPC-based means is used to request and return data. For links, a database is used to store and index link information; authentication is performed on database writes only. For link metrics, three basic metrics are supported:



Entity Query Service Methods

```
set_outlinks(Links, AuthToken) : Bool
set_signed_outlinks(Links) : Bool
unset_outlinks(Links, AuthToken) : Bool
reset_outlinks(AuthToken) : Bool
get_outlinks : Links
set_inlinks(Links, AuthToken) : Bool
set_signed_inlinks(RSASignedLinks) : Bool
unset_inlinks(Links, AuthToken) : Bool
reset_inlinks(AuthToken) : Bool
unset_inlinks(Links, AuthToken) : Bool
reset_inlinks(AuthToken) : Bool
get_inlinks : Links
get_remote_rtt(IPAddress) : RttData
get_remote_tput(IPAddress) : Float
get_cpu_load : Array<Float>
set_mappings(AddressMappings) : Bool
unset_mappings(AddressMappings) : Bool
reset_mappings : Bool
get_mappings : AddressMappings
```

Link Query Service Methods

```
get_link(LinkID, Entity, Network, AuthToken) : Link
get_inlinks(Entity, Network, AuthToken) : Link
get_outlinks(Address, AuthToken) : Link
get_links(Network, AuthToken) : Links
set_links(Links, AuthToken) : Bool
unset_links(Links, AuthToken) : Bool
request_links(Links, AuthToken) : RSASignedLinks
reset_all(Network, AuthToken) : Bool
```

Path Query Service Methods

```
get_paths(Network, Entity, Entity, Fixnum) : Paths
```

Figure 2.9: Control Plane Query Service Methods

remote RTT, remote throughput and CPU load; that is, an RPC query may be made to a remote entity to obtain the current average RTT to another entity, current throughput to another entity or the current CPU load on the remote entity.

2.5.4.2 Implementation

RPC queries are implemented via HTTPS-based XML-RPC [163, 165]. Link query service, link metric query service and path query service methods take the form depicted in Figure 2.10.

The link query service is backed by a MySQL [114] database with tables as depicted in Figure 2.11, implemented by the daemon program `sora_sqldbd`, which handles the XML-RPC interface with the database; the daemon is queryable via the API directly or via the command line utility `sora_db`. API queries which write to the database require username and password-based authentication via a token passed with the query; the token is authenticated against a password file local to the database host.

The link metric service measures and returns RTT using `ping`, throughput using `netperf` and CPU load averages using `uptime`. The link metric service is imple-

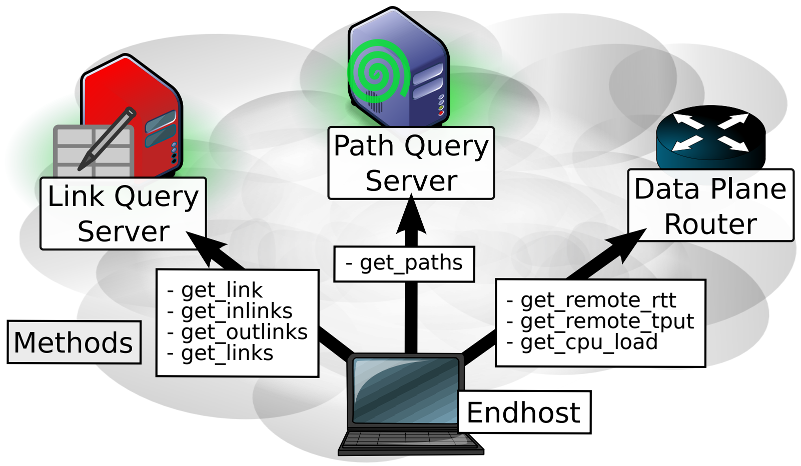
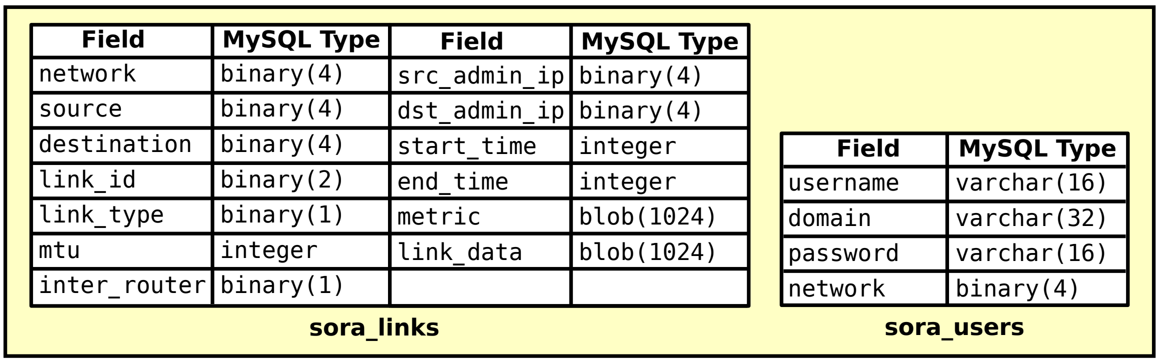


Figure 2.10: Control Plane Query Services

mented by the daemon program `sora_entityd`; it is queryable via the API directly or via the command line utility `sora_entity`.



Field	MySQL Type	Field	MySQL Type
network	binary(4)	src_admin_ip	binary(4)
source	binary(4)	dst_admin_ip	binary(4)
destination	binary(4)	start_time	integer
link_id	binary(2)	end_time	integer
link_type	binary(1)	metric	blob(1024)
mtu	integer	link_data	blob(1024)
inter_router	binary(1)		

sora_links

Field	MySQL Type
username	varchar(16)
domain	varchar(32)
password	varchar(16)
network	binary(4)

sora_users

Figure 2.11: Link Query and Link Request Service Tables

The path query service computes paths using the path computation engine module of the library discussed above. The path query service is implemented by the daemon program `sora_pcsd`; it is queryable via the API directly or via the command line utility `sora_pcs`.

By using these query services, it is possible to centrally manage link and path data such that all entities across the experiment network can query for and obtain it automatically and to automate the retrieval of link metric data from a service running on each entity. Nevertheless, when convenient (e.g., on smaller virtual networks), it is still possible to use file-based link and path storage and retrieval.

2.5.5 Router Link Instantiation

The router link instantiation component also builds on and extend the control plane library. Link instantiation is defined as the establishment of one or more links on a given entity using link data. Accordingly, as depicted in Figure 2.12, the router link instantiation service allows for *link data regarding one or more links to be sent to a given router and for the links to be established.*

2.5.5.1 Design

An RPC-based means is used to send link data and request instantiation; the router instantiates the links if and only if user authentication succeeds or signed link authentication succeeds in the case of use in shared virtual network mode, as is described in more detail in Section 2.5.6. Further, in order to support both kernel-level and user-level operation of the data plane router, the instantiation handler determines whether or not the Click kernel-level router or user-level router is in use and sends the link to the appropriate Click data plane router interface.

2.5.5.2 Implementation

RPC queries are implemented via HTTPS-based XML-RPC. The methods implementing link instantiation are summarized in the link query service methods of Figure 2.9 (`set_outlinks`, `set_signed_outlinks`, `unset_outlinks`, `reset_-outlinks`, `get_outlinks`, `set_inlinks`, `set_signed_inlinks`, `unset_-inlinks`, `reset_inlinks` and `get_inlinks`); service queries are made to respective servers in the form depicted in Figure 2.10.

When a link request is received (e.g., via `set_outlinks`), the request is authenticated against a local password file via an authentication token (username/password pair) passed with the query; in the case of signed links, the RSA signature is checked, as is described in Section 2.5.6.3. If authentication succeeds and all requested links are instantiated, the query returns true; otherwise the query returns false.

As depicted in Figure 2.12, when a link instantiation request is received and approved, it must be sent to the Click data plane router instance. This involves communicating with the router's configuration interface: if the user-level router is running, the interface is Click's `ControlSocket`, which waits on a TCP socket for configuration commands; if the kernel-level router is running, the interface is Click's proc-like filesystem, which allows configuration commands to be written to a file which passes the information to the appropriate router element running in the kernel. Either interface allows access to the Click processing elements. The query server daemon program implementing this functionality is `sora_entityd`.

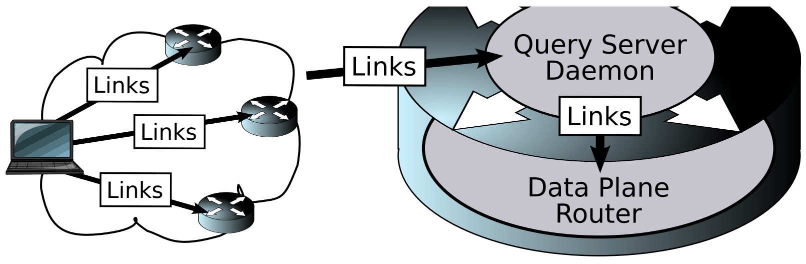


Figure 2.12: Router Link Instantiation

Currently, for PlanetLab and Emulab use, instantiation of UDP outlinks is supported. Although the query service API allows inlink requests, since UDP allows inbound connections from any host, no *inlink* instantiation is required; such requests are currently ignored.

The client instantiation request program is `sora_instantiate_links`; it optionally reduces query time by spawning multiple threads to execute queries in parallel. That is, it inputs a set of links, maps router administration IP addresses to links, and optionally spawns a given maximum number of threads simultaneously—one per router.

2.5.6 Framework for Shared Resource Usage

As described above, this framework provides a central link request service. Its purpose is to facilitate the above-described *optional* mode of operation wherein pre-instantiated links may be shared between experiments and external links added while still ensuring only authorized use of testbed resources.

2.5.6.1 Design

As depicted in Figure 2.13, the service accepts link requests along with user authentication information; it authenticates the user making the requests using the testbed's own authentication mechanism. For example, on PlanetLab it uses PLC-API, which provides access via HTTPS-based XML-RPC to the testbed user database. If authentication fails, the query fails; otherwise, the service checks each link to determine whether it is acceptable. If any check fails, the query fails. Otherwise, the server creates and returns a set of *signed links*; a *signed link* is link data accompanied by a cryptographic signature; the cryptographic signature proves to any router it is presented to that the link data has been validated by the link request service.

This design has the dual advantage of (1) allowing the same parallelizable interface for link instantiation described above to be used for all links while (2) not requiring any centralized per-virtual network link storage be performed for approved links.

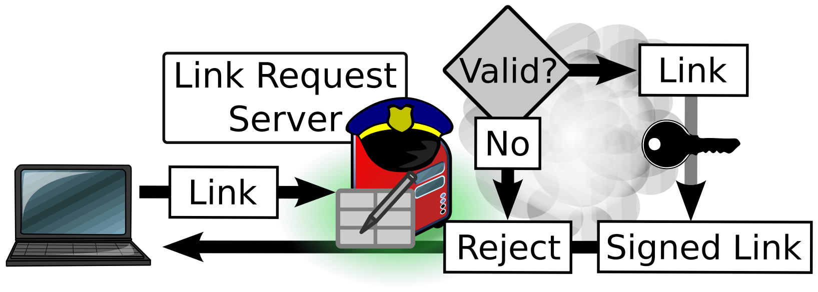


Figure 2.13: Link Request Service Overview

Pre-instantiation of all possible testbed links is adopted as a means to allow quick instantiation of virtual networks; in this way, if such links can be shared between experiments, instantiation of individual links can be skipped.

2.5.6.2 Implementation: Pre-Instantiation

Pre-instantiation of all links implies a full mesh topology. As described above, the current PlanetLab implementation already shares a single UDP port for all inbound links, therefore only outbound links are required. Removing the need for all but a single inbound link means that a full mesh requires one outbound link between each PlanetLab node; at its current size of 1045 nodes, this equates to just over one million links (1,090,980). However, each individual router only requires a table storing 1,044 links, lookups for which can be handled by a simple hash table.

The set of all links is pre-generated by a script (`sora_mk_pl_links`) and placed in a file: the network ID for all links is set to `0.0.0.0`; the source and destination entity IDs are set to the primary IPv4 addresses of the source and destination PlanetLab nodes. The pre-generated file is compressed for periodic download to each PlanetLab router. The uncompressed file is approximately 309 MB; the compressed file is approximately 9.5 MB. Each router obtains its links via a regular expression file search (`grep`) for its source address and instantiates them at startup.

One data plane issue arises when sharing pre-instantiated links between virtual networks: when a packet is received, a lookup is performed on the three tuple: <network ID (from the packet), entity ID (of the router on the network), and the link ID (of the next hop in the path)>. However, for all shared links, the network ID is `0.0.0.0`, which means such queries would never succeed since the network in the packet control data does not match.

To address this issue, in shared environments, the link ID space is split: the highest 4,096 link IDs (of the 65,535 link ID space) are reserved for shared links; the lower part of the link space (zero to 61,439) are dedicated to virtual network use. That is, in shared testbed mode, when a packet arrives, if its next hop link ID is greater than 61,440, the network ID for lookup is set to the shared network (`0.0.0.0`) regardless of the packet network. With this split link ID space hack, both links can be used simultaneously, albeit at the expense of a smaller link ID space.

2.5.6.3 Implementation: Request Server

RPC queries are implemented via HTTPS-based XML-RPC. The link request service API has a single method `request_links(links, auth_token)`. First, the username and password contained in the `auth_token` structure are authenticated on PlanetLab via the HTTPS-based XML-RPC-based PLC-API [27]. On success, a network ID for the username is queried in the MySQL database; if none exists, one is created and associated in the `sora_users` table with the username and PlanetLab domain (it should be noted that the password is *not* stored for such queries; it is only stored for the "local" domain, which provides for local database authentication).

After user authentication succeeds, each link is then verified: currently all link requests are accepted provided the source port of the link is the default UDP source port used for PlanetLab. The link's network is reset to the one assigned to it. An SHA-1 [41] hash is made of a fixed representation of the link data; the hash value is signed using the request server's private RSA key; and an `RSASignedLink` is created using the given link and the signature. This is repeated for each link and the result is returned as a set of `RSASignedLinks` to the client. The server program implementing link request is `sora_sqldbd`; the client program is `sora_db`.

The given signed links may then be instantiated via the same process described above for normal links, using the `set_signed_outlinks(signed_links)` method.

2.6 Application Interface

As described in Section 2.3.3, the requirements for the application interface are: (1) ease of application-directed routing-aware application development and experimentation with the data and control plane; and (2) usability with existing network applications and protocols.

2.6.1 Challenges

The challenge for easing development and experimentation lies in allowing for convenient swapin and swapout of different configurations (e.g., methods for path scheduling and monitoring) and including a library for common tasks. The challenge for providing usability with existing applications and protocols while allowing application-directed routing features to be used simultaneously lies in facilitating ingress and egress of application data to and from virtual networks.

2.6.2 Approach

The approach adopted to allow swapin and swapout and a framework for implementing common tasks such as path scheduling and monitoring is a modular "conduit endpoint" structure and associated libraries for such tasks.

The approach adopted for facilitating convenient ingress and egress of packets to and from virtual networks is to provide two means for application interface: (1) a libc wrapper which intercepts system calls such as `socket`, `connect`, `send` and `recv`, ingresses sent packets based on a given criteria and egresses received packets to the application; and (2) an ingressing router, which is capable of acting as a default gateway (at layer two) on a network, intercepting all traffic inbound and outbound from a given host (or set of hosts) and performing ingress and egress to and from virtual networks based on some flexible criteria (e.g., source IP address and port number).

2.6.3 Path Conduit Endpoint Interface

As depicted in Figure 2.14, a *conduit* is an idealized construct for managing all paths between two communicating endpoints. Ideally, this structure would provide for monitoring of paths and the links that comprise them and perform path scheduling to optimize performance and efficiency. However, because a path is formed by individual links and the individual links are managed by routers distributed across the network, it is difficult for a single data structure to directly encompass and manage all such links.

2.6.3.1 Design

However, by performing path monitoring of traffic over the paths at both sides of this hypothetical path conduit and feeding information regarding their performance back to each other, a pair of "path conduit endpoints" may act as an approximation of this

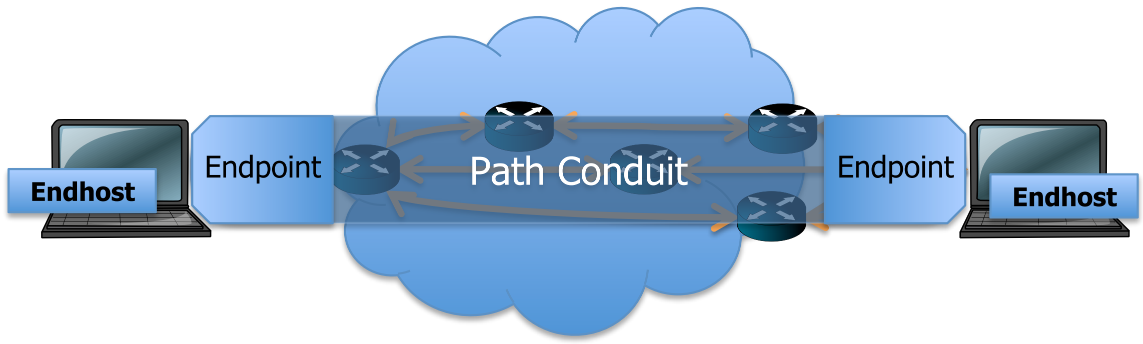


Figure 2.14: Path Conduit and Endpoints

idealized structure. Thus, a *path conduit endpoint* interface is a data construct for performing communications between two virtual network entities; it manages the functions of: *path scheduling*—selection of the next path to be used for a given packet; *path performance monitoring*—either passive or active monitoring of paths being used; and *path performance feedback*—sending monitored information back to the originating conduit endpoint such that it can be used in path scheduling.

Specifically, the path conduit endpoint structure provides a modular interface to the data and control planes described above for both inbound and outbound paths between a pair of communicating endpoints. A path conduit endpoint: obtains paths from the control plane, sends and receives packets on inbound and outbound links, performs monitoring of performance on both inbound and outbound paths, feeds information back to the sender regarding inbound paths, and may use collected path information to inform path scheduling based on a given policy. Header options processing, monitoring and feedback handling are performed by *packet processing modules* which may be added or removed; these modules are relatively small methods which, input a packet, perform a given task on it and return either the packet or an error.

An example conduit endpoint structure for implementing a multipath datagram service is depicted in Figure 2.15. As depicted, a path conduit endpoint comprises the following components: an *inbound link (inlink) set*, an *outbound link (outlink) set*, an *inlink reader*, an *outlink writer*, a *path scheduler*, a *path set*, an *input (pull) module chain* and an *outbound (push) module chain*.

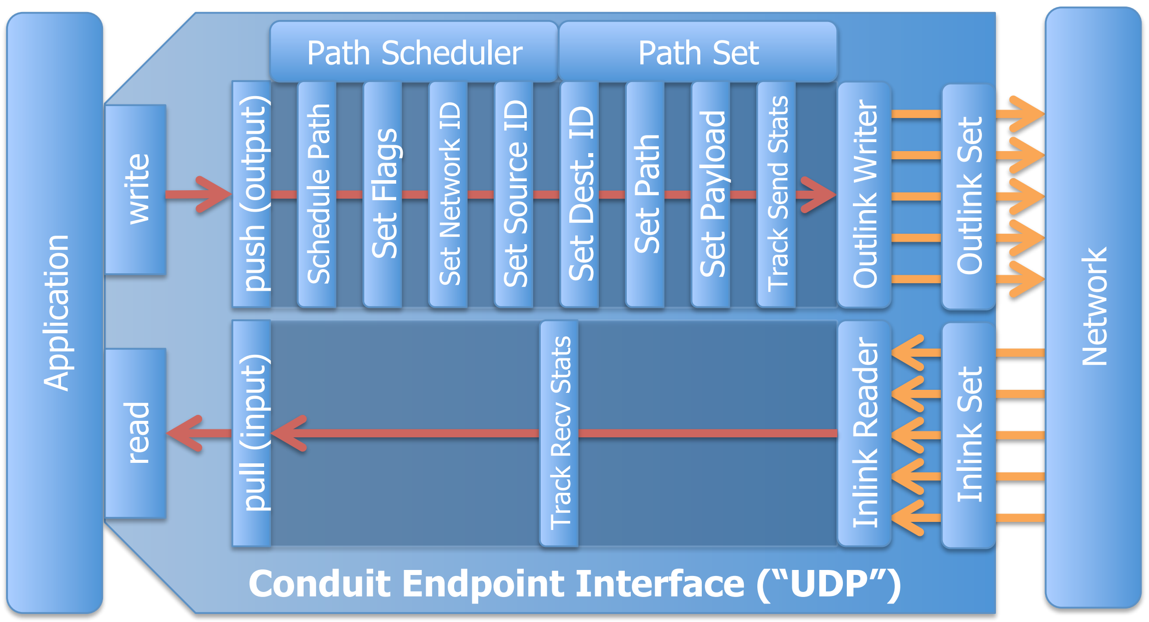


Figure 2.15: Path Conduit Endpoint Structure for "Multipath UDP"

The conduit endpoint exists mainly to provide packet send and receive methods to applications and transport layers. For example, when a packet is written to the conduit of Figure 2.15, it is "pushed" through the output (push) module chain. Each output module executes a specific action on the packet (i.e., scheduling a path; addition of header options: packet flags, network ID, destination ID and path; setting of the payload; tracking of sent packet statistics). The module chain ends at the outlink writer, which determines, based on the selected path, which outlink of the outlink set to write the packet to. It writes the packet to the given outlink and thereby sending it on the given path.

When a packet is to be read from the conduit, it is input (pulled) from the inlink reader. The inlink reader selects a random link with data, reads a packet from the link and returns it. The packet is then passed, for example, through the received packet statistics tracking module and back to the application.

With this structure, operations on packets can be handled automatically by modules, new combinations of packet operations can be tried easily without significant code modification and code can be reused via the modular interface.

2.6.3.2 Implementation

The conduit endpoint interface is implemented via a C++ template class `SoraUser-LevelConduit`. It is templatized to allow instantiation using a variety of components. Its five template parameters are: *path type*, *path scheduler type*, input/output *module chain pair type*, *inlink reader type* and *outlink writer type*. Each is explained below.

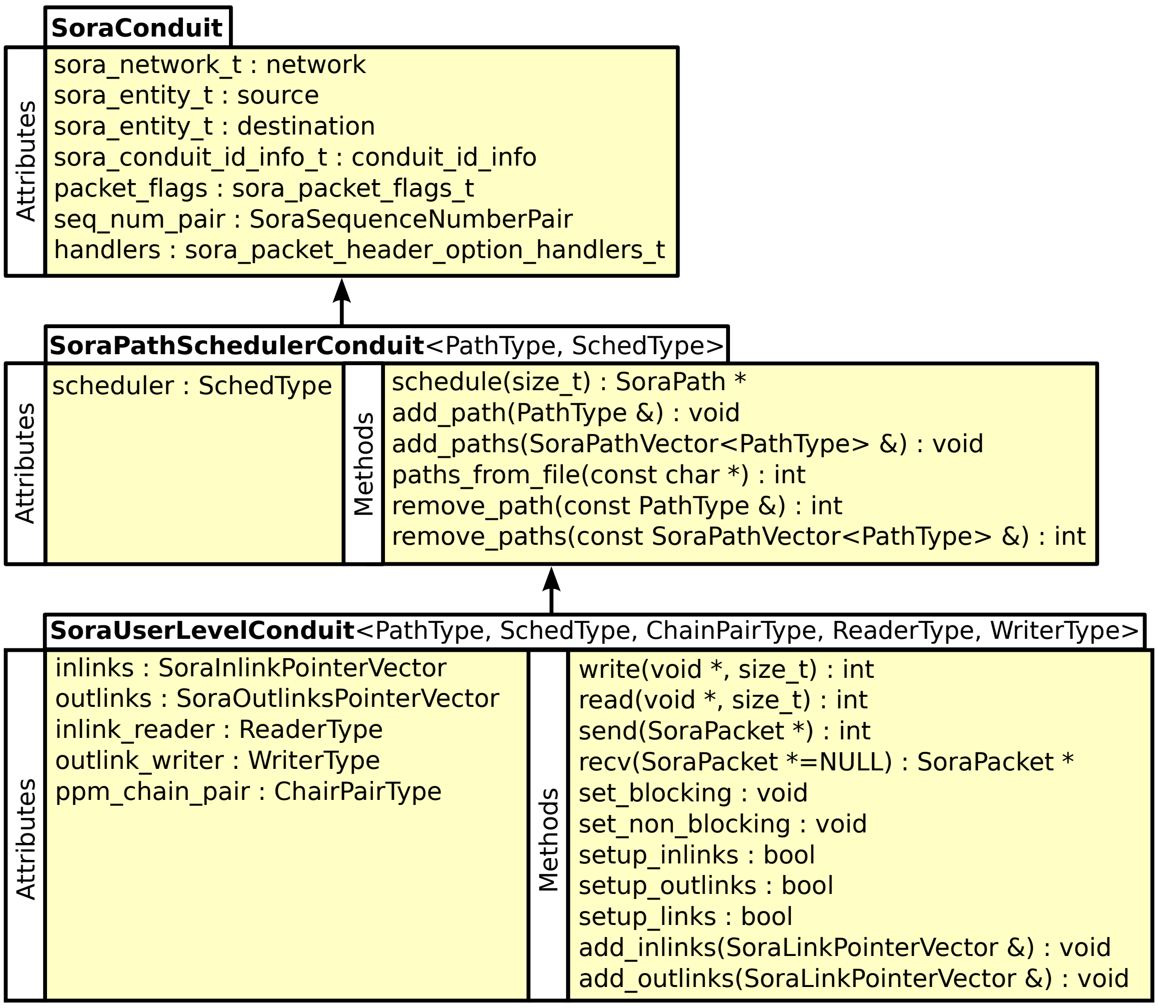

SoraConduit
Attributes
sora_network_t : network
sora_entity_t : source
sora_entity_t : destination
sora_conduit_id_info_t : conduit_id_info
packet_flags : sora_packet_flags_t
seq_num_pair : SoraSequenceNumberPair
handlers : sora_packet_header_option_handlers_t
SoraPathSchedulerConduit<PathType, SchedType>
Attributes
scheduler : SchedType
Methods
schedule(size_t) : SoraPath *
add_path(PathType &) : void
add_paths(SoraPathVector<PathType> &) : void
paths_from_file(const char *) : int
remove_path(const PathType &) : int
remove_paths(const SoraPathVector<PathType> &) : int
SoraUserLevelConduit<PathType, SchedType, ChainPairType, ReaderType, WriterType>
Attributes
inlinks : SoraInlinkPointerVector
outlinks : SoraOutlinksPointerVector
inlink_reader : ReaderType
outlink_writer : WriterType
ppm_chain_pair : ChairPairType
Methods
write(void *, size_t) : int
read(void *, size_t) : int
send(SoraPacket *) : int
recv(SoraPacket *=NULL) : SoraPacket *
set_blocking : void
set_non_blocking : void
setup_inlinks : bool
setup_outlinks : bool
setup_links : bool
add_inlinks(SoraLinkPointerVector &) : void
add_outlinks(SoraLinkPointerVector &) : void

Figure 2.16: `SoraUserLevelConduit` and Parent Classes

First, a *path scheduler* is a templatized class with one parameter: path type. The *path* type must be a subclass of `SoraPath`, which is the default template path type; `SoraPath` implements the basic functionality related to storing the link identifiers that comprise the path. Within the SORA framework, paths are means for path metrics to be stored by path monitoring and feedback receipt modules; different path types allow implementation of different metrics. For instance, one path type may track loss rate; another may track both loss rate and latency over the past 10 minutes. Subclasses of `SoraPath` include: `SoraPathEpochTotals`, which tracks loss and latency since conduit creation; `SoraPathTimeWindowTotals`, which tracks loss and latency on a time-window basis and `SoraPathGeneralTotals` which tracks both of these.

The path scheduler type determines choice of path scheduler implementations. All path schedulers inherit the `SoraPathScheduler` base class and thus have the same interface, which centers around the `schedule` method which returns a pointer to the next scheduled path object. Currently supported implementations include round robin (`SoraRoundRobinPathScheduler`) and random (`SoraRandomPathScheduler`); the default is the random scheduler.

An input/output *module chain pair* is a class formed by input packet and output packet processing module chains, as discussed above. Each module chain is a linked list of packet processing modules wherein each successive module performs some task and either calls the next module or returns an error value.

The module chain pair must be a subclass of `SoraPPMPushPullChainPair`. Modules may be allocated and added directly to its processing chains members `push_chain` and `pull_chain`. In order to perform more complex setup operations, it is often convenient to subclass this class. For example, monitored information (e.g., the loss of a given sequence number) may be fed back to the sender on the next available packet via an output packet processing module. To implement this, however, the loss monitoring module must have access to the loss feedback module, which requires the additional setup afforded by the constructor of a subclass. Packet processing modules are described below.

An *inlink reader* is a class which reads the next packet from the set of inlinks. It is, by default, implemented by the `SoraRandomInlinkReader` class, which randomly selects the next ready inlink. A random inlink selection policy efficiently avoids link read starvation and, unless a different reading policy is desired, there is no reason not to use this default.

An *outlink writer* is a class which determines the next hop link by referring to the next hop set in the selected path and is, by default, implemented by the `SoraOut-linkWriter` class. Since the outlink scheduling policy is effectively set by the path scheduler, there is little reason to change from the default, except, for instance, in the case of handling blocking writes, where the current framework will block.

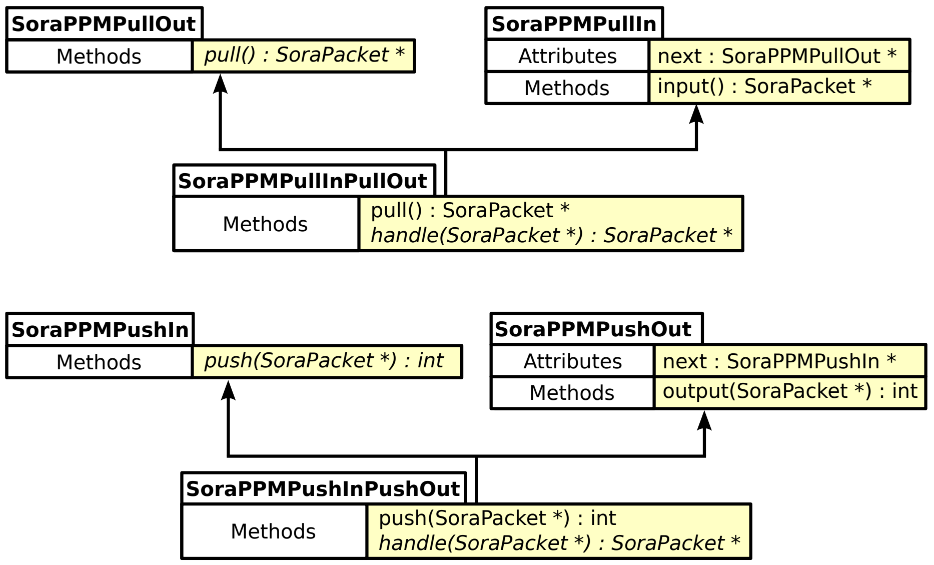


Figure 2.17: Packet Processing Module Base Classes

There are two types of packet processing modules, corresponding to the output and input chains: *push input/push output*, and *pull input/pull output*. The interfaces of these classes are implemented via the pure virtual classes `SoraPPMPushInPushOut` and `SoraPPMPullInPullOut`.

To implement a packet processing module, an implementor inherits one of these interfaces and then simply implements the `SoraPacket * handle(SoraPacket *)` method to perform tasks on the given packet. The `SoraPPMPushInPushOut` and `SoraPPMPullInPullOut` classes call the `handle` method with any packet that they are pushed (or pulled) a packet. These classes expect the implementor modules of the `handle` method to return a packet on success or `NULL` on failure. Packet processing modules should *never* deallocate a packet; this is the responsibility of the initiator of the pull or push chain.

Current inbound modules include `SoraPPMMonitorPacketLoss` for sequence number-based packet loss monitoring and `SoraPPMMonitorPathLatency` for unidirectional latency monitoring. Packet loss monitoring is performed by tracking packet sequence numbers and the paths they map to at the sender, monitoring excessive packet delays at the receiver and then feeding information back to the sender regarding excessively delayed packets. Efficient unidirectional path latency monitoring is performed using an algorithm similar to that recently published by Song et al. [147]. An analysis of performance of the tools provided for both packet loss monitoring and unidirectional path latency monitoring is provided in Section 2.7.1.3.

Current outbound modules include `SoraPPMSchedulePath` for path scheduling, `SoraPPMSetFlags` for setting flags options, `SoraPPMSetNetwork` for setting the network ID option, `SoraPPMSetSource` for setting the source entity ID option, `SoraPPMSetDestination` for setting the destination entity ID option, `SoraPPMSetPath` for setting the path option, `SoraPPMSetPayload` for setting the payload based on the amount of space available considering the link MTU and the size of the added options.

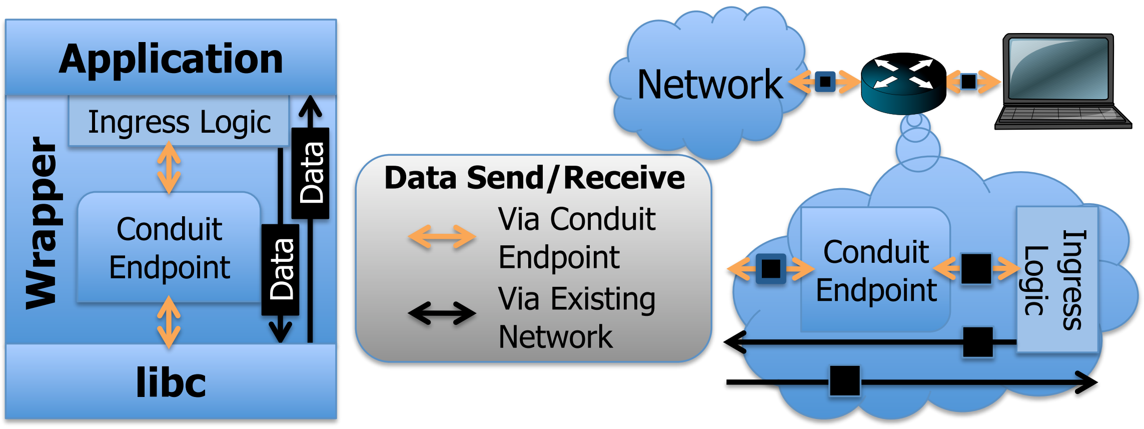


Figure 2.18: Two Ingress Types: Router and libc Wrapper

2.6.4 Packet Ingress/Egress: libc Wrapper

As depicted on the left side of Figure 2.18, packet ingress can be implemented by intercepting normal network-related system calls and redirecting them to use a virtual network.

2.6.4.1 Design

This interception and redirection of system calls is performed using a *wrapper library*. A *wrapper library* is defined as a dynamically loaded shared object file which overrides (*wraps*) existing shared library methods and modifies their behavior.

More specifically, Linux and a number of other operating systems are equipped with feature allowing a given shared object library to be loaded before any others are loaded. That is, the symbols defined in the pre-loaded library take precedence over those loaded later, thus allowing system call symbols to be overridden. Linux also allows applications to query for the location of the next instance of a given symbol (in the case of overridden system call symbols, the location of the *real* symbol), allowing it to be called as well from the same process. These two features of (1) redirecting calls to a given function and (2) being able to call the redirected function on demand allow system calls such as `socket` and `sendto` to be overridden based on certain criteria.

2.6.4.2 Implementation

In the current implementation, when a socket requesting UDP protocol traffic is requested, a conduit endpoint interface is created instead and a fake socket file descriptor (one with value 1023) is returned. Then, whenever the fake file descriptor is passed to any other relevant system call, the appropriate conduit endpoint interface call is invoked instead (e.g., `send` for `sendto`, `read` for `recv`, etc.). For other file descriptors (e.g., for files or TCP sockets), system calls are passed through directly.

This method has the advantage of not requiring setup and installation of any additional routers because ingress and egress happen within the wrapper. However, a number of limitations exist with the current implementation: (1) because no reliable bytestream implementation is currently available for use with the user-level conduit, only UDP sockets can currently be wrapped; (2) because there is no corresponding functionality in the conduit, the out-of-band messaging functions of `sendmsg` and `recvmsg` are not yet usable; and (3) the current implementation supports mapping of one socket per application.

Nevertheless, the interface is usable for a number of applications and all conduit-related performance tests below use `netperf` run via a wrapped UDP socket.

2.6.5 Packet Ingress/Egress: Router

As depicted on the right side of Figure 2.18, an ingressing router acts as a default gateway for one or more virtual network-unaware endhosts on a network. That is, it is a virtual network entity which accepts traffic from said endhosts and forwards it on their behalf, while ingressing select outbound traffic onto virtual networks and egressing any received virtual network inbound traffic to the endhosts.

2.6.5.1 Design

The design of this structure is based on extensions to the Click data plane router described above in Section 2.4.5. In order to act as a default gateway for endhosts, the Click data plane router is passed all traffic from them. As depicted on the right side of Figure 2.18, ingress logic is used to determine which packets to attempt virtual network ingress for. This ingress logic can use, for example, the source and destination IP address and/or port number to make the ingress determination. Packets which are not to be ingressed can be routed normally (e.g., passed directly to a next hop router).

For packets to be ingressed, first a mapping is maintained by the router between endhost IP addresses and virtual network addresses (network and entity identifiers). These mappings are maintained using the methods of the entity query service summarized in Figure 2.9 (`set_mappings`, `unset_mappings`, `reset_mappings` and `get_-mappings`). When addresses are found in the router's IP address to virtual address map for the IP source and destination address, a conduit endpoint interface is instantiated with the respective source and destination network and entity identifiers.

Queries are then made for inbound and outbound links. If inbound and outbound links are available, they are added to the conduit endpoint interface and outbound paths are obtained from the given source to the given destination. If outbound paths are available, they are added to the conduit endpoint interface and packets may then be encapsulated and sent via the virtual network. At the egress, packets are decapsulated and sent, as is, on the network to their destination. A similar ingress process occurs for any reply.

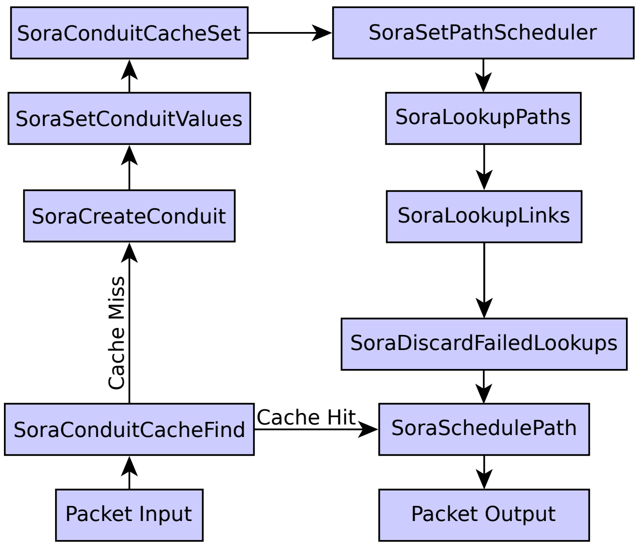


Figure 2.19: Click Ingressing Router Inbound Lookup Elements

2.6.5.2 Implementation

The Click ingress chain for implementing the queries described above and scheduling packets for sending as well as for egressing received packets is depicted in Figure 2.19.

For ingress, the first element `SoraConduitCacheFind` locates any cached conduit for the given source and destination IP and port pair; if found, it passes the packet directly to the path scheduler; otherwise, the packet passes first to the ingress lookup chain. The first element of the lookup chain `SoraCreateConduit` creates a conduit for the packet; the next modules perform default value setting (`SoraSetConduitValues`), caching (`SoraConduitCacheSet`) setting of the path scheduler type (`SoraSet-`

PathScheduler), path and link lookup (SoraLookupLinks, SoraLookupPaths). If any of the lookups fail, the packet is discarded; otherwise, the packet is passed to the path scheduler and out to the packet processing chain (not depicted), which creates a packet, adds any required options and sends to the output chain. The output chain is essentially the same as that depicted in Figure 2.7 starting with the SoraLookupNextHop element.

For egress, the conduit information in the packet control data (network, source and destination entity ID and conduit ID) is located, and the conduit cache is consulted to determine whether or not a conduit for the packet has already been created. If not, a conduit is created and cached so all further communication back from the receiver uses the same conduit. Then, the encaspulating header is stripped off and the packet is sent out on the network to its destination.

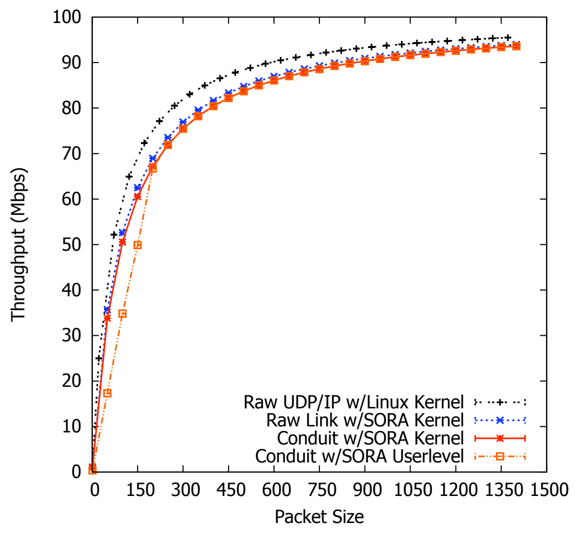
In this way, packets from any application that match a desired pattern can be ingressed onto and egressed off from a virtual network using a given conduit endpoint pair. The disadvantages to using the router are the need to install and configure extra network equipment and the fact that the current implementation only operates at user-level and therefore suffers from precisely the same performance bottleneck as the user-level router; this is discussed next, in Section 2.7.

2.7 Performance Analysis

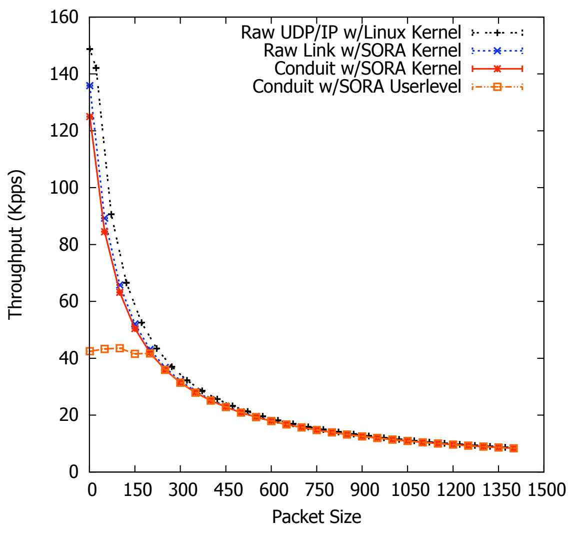
Performance analysis of the three components (data plane, control plane and application interface) is divided into two sections: first, the packet processing performance of the data plane in conjunction with the application interface is evaluated, followed by the control plane link and path discovery means.

2.7.1 Data Plane & Application Interface

Performance analysis was conducted in order to determine how the data plane, application interface and their components perform with actual packet traffic.



(a) Throughput (Mbps)



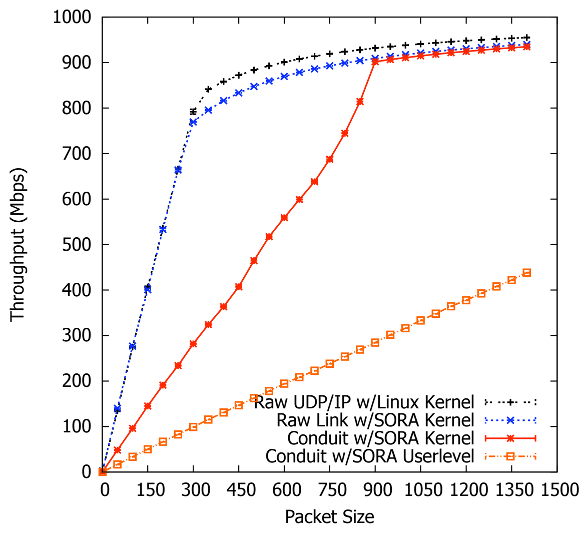
(b) Throughput (Kpps)

Figure 2.20: Throughput Performance (100 Mbps Network)

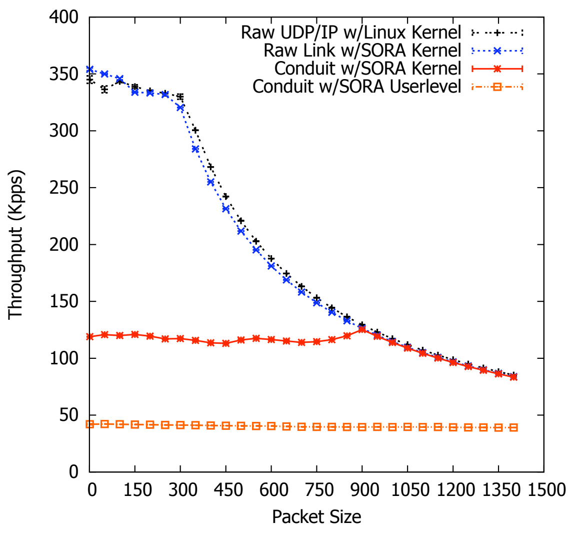
2.7.1.1 Processing Performance Testing Methodology & Environment

In order to evaluate the processing capabilities of both the router and application interface, both in terms of bits and packets per second, a testing environment like that depicted in Figure 2.22 comprising three nodes was established. All nodes were Dell r710's with an Intel Xeon 2.4 GHz (a Nehalem four core, each core having an 8 MB cache) CPU, 3 GB of RAM, running Linux 2.6.24.7-117 and Click 1.8. Moreover, all nodes were single-user and dedicated only to packet processing, meaning that no other processes were competing for processor or memory resources on these machines. The methodology for obtaining throughput results was to send packets from the source endpoint to the destination endpoint as quickly as possible (with no flow control) and monitor how many packets/bits were actually received.

Four test were performed: (1) *Raw UDP*, (2) *Raw SORA Kernel*, (3) *Conduit/SORA Kernel* and (4) *Conduit/SORA User*; each test was conducted five times for a duration of 10 seconds, with the average taken. These test parameters were chosen because measurements in 10 second intervals have been shown to provide relatively consistent results on networks with low jitter and loss rate like that used in the present test network [77], and because of the lack of variability in the data due to the routers and links being dedicated



(a) Throughput (Mbps)



(b) Throughput (Kpps)

Figure 2.21: Throughput Performance (Gbps Network)

to forwarding for this experiment alone. Raw UDP measures raw UDP throughput via `netperf` 2.4.5 [77] through an intermediate unmodified Linux kernel router; it provides a baseline estimate of the maximum throughput performance possible through the network. Raw SORA Kernel measures throughput to and from a single UDP link instance on either endpoint, with the intermediate router running the SORA Click router in kernel mode. Conduit/SORA Kernel measures throughput via `netperf` wrapped using the wrapper described in Section 2.6.4 with the intermediate router running the SORA Click router in kernel mode. Conduit/SORA User measures throughput via the same wrapped `netperf` means but with the intermediate router running the SORA Click router in user mode.

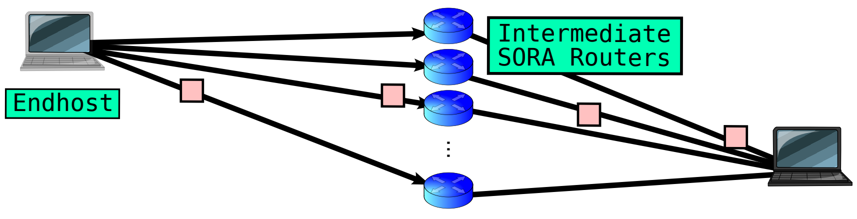


Figure 2.22: Throughput Testing Environment

The results for 100 Mbps environments, typical of most testbeds such as PlanetLab, are depicted in Figure 2.20; those for Gbps environments are depicted in Figure 2.21.

2.7.1.2 Packet Processing Performance Results

First, the throughput results in Figure 2.20 indicate that the performance of the SORA Click kernel router scales well against the baseline UDP through all packet sizes; thus, it does not present a bottleneck in typical testbed environments. Next, performance via the SORA conduit endpoint interface is nearly identical to that of the kernel router indicating that it also does not present a throughput bottleneck. However, as can be seen more clearly in the packet processing depiction of Figure 2.20b, throughput via the Click *user-level* router does present a bottleneck at packet sizes below 200 bytes or packet processing speeds between 40 and 45 Kpps.

Therefore, in environments such as Emulab, where kernel modifications are possible, the SORA conduit endpoint interface can provide performance on par with native IP routing for all packet sizes when used with the kernel router at 100 Mbps or less. However, when used with the user-level router, a performance bottleneck exists for smaller packets even for 100 Mbps networks, the exact extent of which will depend upon the packet processing capabilities (i.e., CPU and memory bus) of the given virtual network router.

Next, the same test was done using the same hardware but with Gbps-capable switches. The throughput results are depicted in Figure 2.21a; they indicate that the performance of the SORA Click kernel router also scales well against the baseline UDP through all packet sizes even at full Gbps speeds; thus, it does not present a bottleneck in Gbps environments. However, while performance of the SORA conduit endpoint interface is nearly identical to that of the kernel router in Figure 2.21b down to packet sizes of 900 bytes, beyond this point the packet processing speed of the conduit endpoint interface reaches a limit of about 140 Kpps for all packet sizes beyond. Moreover, the user-level router is, as above, consistently limited to 50 Kpps and is thus not well suited to Gbps environments.

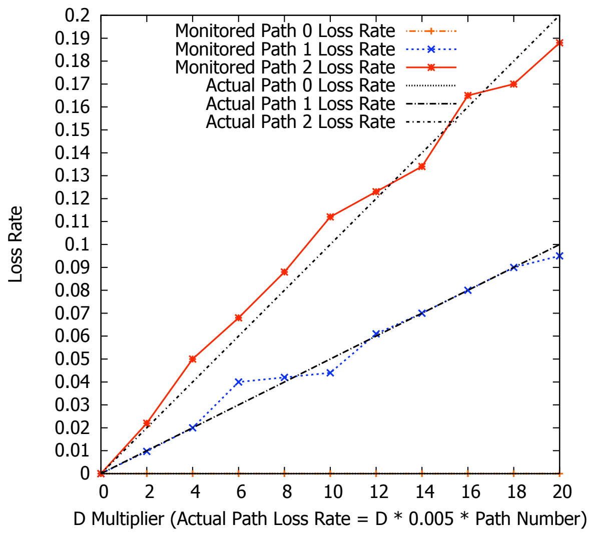
Therefore, while the SORA kernel router is capable of scaling with IP, when Gbps performance is required of the conduit endpoint interface, packet sizes must currently be kept above a certain minimum, which depends upon the packet processing capabilities of the given virtual network router.

The reason for this limitation on the conduit interface is largely due (1) to the extra data copy overhead required with the libc wrapper: it is passed a buffer via `send` and has no choice but to copy it into the UDP buffer, resulting in extra overhead not experienced by UDP; and (2) to the extra options processing that UDP does not have to perform. Given that the extra copy overhead is precisely the bottleneck experienced between the user-level and kernel-level Click router and the kernel-level implementation allowed the router to scale with IP for all packet sizes, it is believed that a kernel-level implementation of the conduit endpoint interface would also improve its performance similarly.

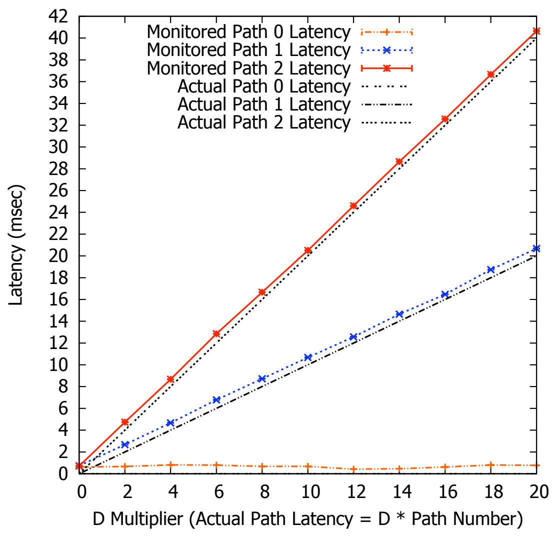
2.7.1.3 Path Monitoring Tools Testing Methodology & Environment

As discussed in Section 2.6.3.2, the application interface provides path performance monitoring libraries to ease development of application-directed routing applications and transport protocols; these include packet loss and unidirectional path latency.

In order to evaluate the accuracy of the monitored statistics provided by these utilities, a testing environment similar to that described above (depicted in Figure 2.22), but comprising three intermediate routers, and thus three paths, was established. Then, traffic sent over the network via a conduit endpoint pair was monitored using the utilities described above. Two separate tests were performed: one to measure loss rate monitoring performance and another to measure latency monitoring performance. In each test, the actual packet loss rate and latency on two of the three links forming the paths of the network were varied with each path being assigned unique latency and loss rates. Loss rate and latency to the links were assigned by numbering the paths $p = 0$ through 2 and using a value D to assign per-path loss rate of $D * 0.005 * p$ and per-path added latency of $D * p$ msec. In order to test for false error messages and latency error, one path was held constant (at zero added loss and latency) for all tests. The monitored loss rate and latency were then compared to the actual values.



(a) Loss Rate



(b) Unidirectional Latency

Figure 2.23: Monitoring Performance (Monitored vs. Actual)

2.7.1.4 Path Monitoring Tool Performance Results

The results are summarized in Figure 2.23. First, Figure 2.23a indicates that, as the loss rate is increased, the monitor is capable of providing an effective estimate. Next in Figure 2.23b latency was also shown to have low error with respect to the actual set values. Thus, results indicate that the loss rate and latency utilities provided for use with the application interface are effective at measuring loss rate and unidirectional latency.

2.7.2 Control Plane

Performance analysis was conducted in order to determine the performance (and thus convenience) of the control plane link and path discovery services in realistic environments. As a measure for what could be considered "convenient", query times in excess of one minute for network setup operations were regarded as "inconvenient".

2.7.2.1 Methodology & Testing Environment

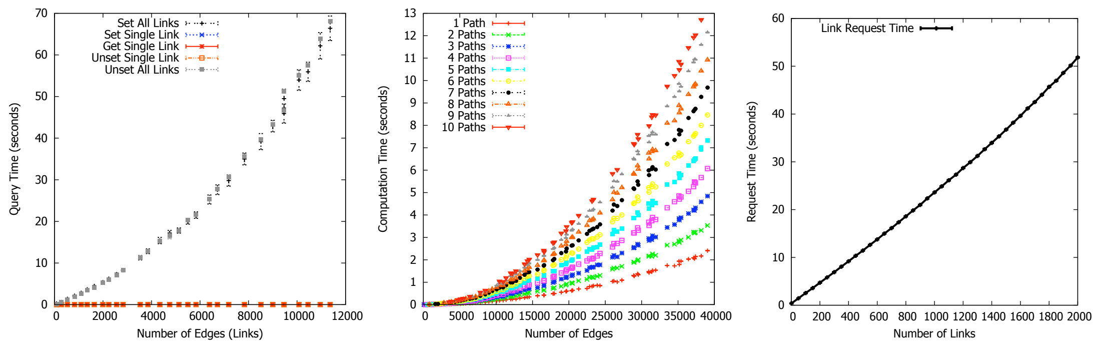
The test objective for both the link and path discovery services was query time. The methodology for obtaining both link and path query times was to (1) obtain a realistic graph, (2) generate graphs of varying size similar to the obtained graph, (3) generate a virtual network topology based on each graph and (4) test the performance of standard queries using the virtual network topology. As a realistic graph, a view of the current Internet AS graph was constructed using the Skitter datasets gathered by CAIDA [22] via RouteViews [154] for the period from January 1, 2008 to January 14, 2008. To generate graphs of varying size similar to this graph, the Orbis [102] graph generation package was used. The original graph from the CAIDA set contained 7,632 AS's (vertices) and 35,476 inter-AS connections (edges).

A total of 78 graphs were generated, ranging in size from a smallest of 32 vertices and 84 edges to a largest of 8,499 vertices and 39,060 edges. For the case of path computation, all 78 graphs were used. For the case of the link database, only 30 graphs were used, ranging in size from a smallest of 32 vertices and 84 edges to a largest of 2,543 vertices and 11,366 edges. (Only 30 graphs were used because the average setup time of largest graph exceeded the range of one minute.)

To evaluate link query performance, MySQL [114] version 5.0.51 was configured with the tables depicted in Figure 2.11 on a machine with 500 MB RAM and dual Intel Pentium D 2.8 GHz processors, each with a 1 MB cache. For each virtual network, the time required to: (1) insert each virtual network, (2) perform single link insertion and retrieval operations and (3) remove all links was measured 30 times. The average of each run is graphed with error bars indicating standard error in Figure 2.24a.

To evaluate path query performance, SORA was installed on an Intel Core 2 Duo 2 GHz processor with 4 MB cache, and 2 GB RAM running Mac OS X 10.4.8. For each virtual network graph, $k = 1$ to 10 paths were computed three times and the running time for each was measured. The average of each run is graphed with error bars indicating standard error in Figure 2.24b.

2.7.2.2 Results



(a) Link Query (b) Path Computation (c) Link Request

Figure 2.24: Link and Path Query and Link Request Service Performance

The link query results in Figure 2.24a indicate that single queries complete in relatively constant time across all network sizes, which is not unexpected given that databases of these sizes can easily fit into memory. The results also indicate that virtual networks of roughly 10,000 edges can be inserted in under one minute but that, given the seemingly polynomial growth curve, more than this will require significantly longer.

Upon investigation using *direct* database access (as opposed to going through XML-RPC), it was determined that XML-RPC overhead and the parsing of the link data was to blame for a substantial portion of the query processing time required and a more efficient implementation would yield lower query times. Nevertheless, 10,000 edge virtual network insertion time in under one minute is deemed adequate for most testbed-based virtual networks, which typically do not exceed 1,000 nodes.

The path query results in Figure 2.24b depict the expected performance of Dijkstra's algorithm. Further, they indicate that the Ruby path computation module is capable of computing seven disjoint paths on a graph with nearly 8,500 vertices and 40,000 edges (and up to 10 in under 13 seconds), with three paths possible through the largest graph in five seconds. Given that most testbed-based virtual networks have less than 1,000 nodes, the path computation module performance is adequate for most testbeds.

The link request query results in Figure 2.24c indicate that, as expected, link request query time increases linearly with the number of links. This is expected because each

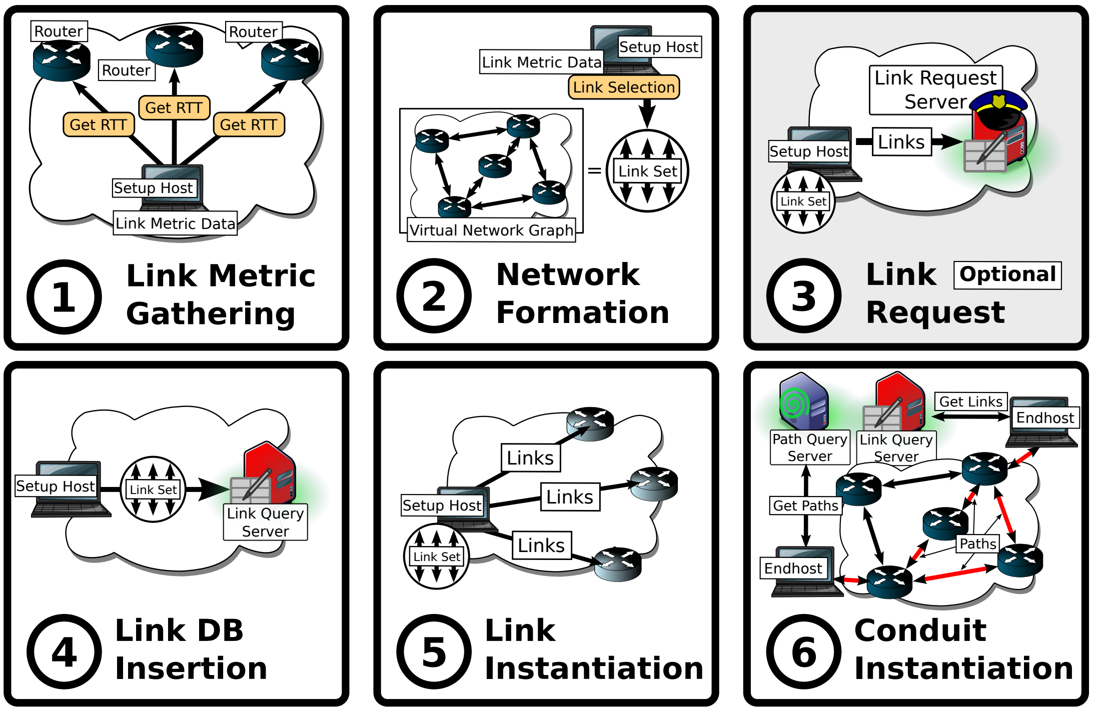


Figure 2.25: Virtual Network Creation Steps

link signature is computed individually in series and has the same run time. The results also indicate that, within the target duration of one minute (an average of 51.83 seconds), requests for 2,000 links can be handled. Given that with link pre-instantiation, the number of link requests required is expected to be far less than the network size, these results indicate that the link query service is suitable for large virtual networks relative to the standard of PlanetLab and the one-minute guideline discussed above.

2.8 Examples

As summarized in Figure 2.25, the general steps for creation and usage of a virtual network are *link metric gathering*, *virtual network graph formation*, *link request* (optional), *link database insertion*, *router link instantiation* and *conduit instantiation*. Notably, this assumes that the Click-based SORA data plane router described in Section 2.4.5.1 and the control plane daemon (`sora_entityd`) described in Section 2.5 is installed and running on each router.

This framework has been deployed and used in application-directed experiments on both Emulab and PlanetLab.

On PlanetLab, an early version of the framework was used to examine the feasibility of using packet dispersion across hundreds of PlanetLab-router-based paths with TCP/IP. Results [91] indicate that, for small numbers of paths, dispersion provides a reduction in the load on individual paths while still providing the reliability benefits of dispersion without greatly degrading performance; however, as the number of paths and the amount of reordering TCP must cope with is increased, performance decreases quickly.

On Emulab, the ingressing router described in Section 2.6.5 was also used to ingress packets onto a virtual network to test TCP/IP performance. Partially in response to the results from the above-described PlanetLab experiment, the performance of different TCP/IP implementations (e.g., RENO and Cubic) was evaluated while using multipath packet dispersion.

In these experiments [92, 94], TCP/IP traffic was dispersed across ten paths simultaneously; then, Emulab was used to add progressively more latency to the links forming these paths in order to induce out-of-order delivery. Because TCP confuses out-of-order delivery with packet loss, its performance is known to suffer greatly under such conditions. To cope with this, *a best-effort network layer reordering* was proposed and its performance evaluated. This service was shown to be capable of preserving near-linear throughput gain with respect to the number of paths in use even under heavy reordering, thereby providing significant performance improvements over unmodified TCP.

2.9 Future Work

The data plane, control plane and application interface are available for use now and provide an immediately usable solution to the problem of how to conveniently develop and test both existing and future applications and transport protocols on application-directed routing networks using the current network infrastructure, as proven both by experiment and usage case. Nevertheless, a number of extensions and improvements have been envisioned for the current implementation, as are described below.

2.9.1 Data Plane: Link Types

As described in Section 2.4.3.2, the framework currently implements two basic link types: UDP/IP and TCP/IP. Because UDP/IP is a datagram-oriented, best-effort transport protocol, UDP/IP-based links are well-suited for experiments which assume a best-effort network layer like that currently provided by the current Internet. Because TCP/IP provides reliable, stream oriented transport, it is well-suited for experiments which assume a connection-oriented network or data link layer, as well as in replicating and comparing parallel TCP/IP experimental results.

However, as described in Section 2.4.3 most any type of tunnel (link) may be implemented using the data plane framework by inheriting the `SoraLink` interface and implementing the required send, receive, setup and teardown methods. The main goals of extending the framework to support further link types are to: (1) support for a larger variety of testbed networks and (2) provide better packet utilization and thus better performance.

In order to support a larger variety of environments, future link support may be extended to include GRE-based tunnels [44], which are used by VINI [12] and CoreLab [115], and OpenFlow-based tunnels [108], which are used by GENI [58]. In order to provide better packet utilization and thus better throughput performance, the link interface may be extended to support embedding of the SORA header into existing packet options such as IPv6 options extensions. Implementation of links provide support for the application interface. In order to support these links in the SORA Click router, SORA Click elements for link receive and send (i.e., `SoraPLIface` and `SoraUDPIPSocket` for current UDP/IP links) would be implemented for the SORA router.

2.9.2 Control Plane: Extensions for Testbed Network Control

As described in Section 2.3, the current framework is not designed to replace existing testbed networks but to complement them—to extend existing testbed control and data planes, allowing creation and operation of application-directed data planes atop them. However, network setup and teardown on testbed environments can be be further eased via extensions to *directly* support link-based network setup and management of testbed networks.

For instance, when using Emulab with the current control plane, three basic steps must be performed before the SORA control plane can be used to instantiate links: (1) creation of virtual network link data representing the desired link topology and link metric characteristics; (2) manual creation of an experiment configuration, comprising routers (hosts), network topology and network latency and bandwidth characteristics to match the given virtual network configuration; and (3) manual instantiation of the network on the testbed and installation of SORA software on the routers.

However, network configuration steps (2) and (3) could be eliminated by extending the current SORA control plane to directly manipulate testbed control planes such as that of Emulab or CoreLab and integrating integrating instantiation features directly into the SORA control plane framework. In this case, on Emulab for instance, an extended SORA control plane would input a link file, create an Emulab experiment configuration based on the links and link metrics therein, and then instantiate the network.

Moreover, where underlying testbed networks support it, such a SORA control plane extension could also be used to *allow live manipulation of link characteristics such as bandwidth and latency*. This would make it possible to not only remotely query for live metric data but to remotely *set* live link metric characteristics.

2.9.3 Control Plane: Extensions for Live Router Link Monitoring

As described in Section 2.3, the goal of this framework is to enable application-directed routing on current networks. Herein, application-directed routing is defined as a packet routing method, whereby applications are given access to an arbitrary number of paths and are made fully responsible for enacting a routing policy as well as making performance decisions by monitoring path performance metrics; thus, link metric information is not used for path scheduling in application-directed routing. However, since live link monitoring information could conceivably be of use in examining different path scheduling schemes, another possible control plane extension is live router link monitoring.

For instance, this could be implemented via extensions to the existing control plane daemon (`sora_entityd`) to perform two additional steps: (1) regularly gather monitoring information regarding the state of all instantiated links; (2) make this information available to entities. To make this information available, three means are possible: a *query* means, a *callback* means and a shared measurement database.

With the query means, another XML-RPC-based method (e.g., `get_monitored_-link_metrics`) would be established for entities to query the router for information regarding any given monitored links (by specifying the network identifier, source entity identifier and link identifier of the given links).

With the callback means, the router would maintain a list of XML-RPC URLs and would provide periodic updates of monitored metric data by calling an XML-RPC method (e.g., `set_monitored_link_metrics`) for each URL with monitored metric data. The list of URLs could be maintained by having entities desiring metric updates register their URLs via a router XML-RPC method (e.g., `register_link_-metric_update_url`).

With the shared measurement database, each *router* would be configured with an XML-RPC URL which it uses to call a query method (e.g., `update_link_metrics`) with a set of links and their monitored metrics. This URL corresponds to a link metric database service which processes said query and updates its database with the given link metrics. Then, entities may query the same URL via a query method (e.g., `get_-monitored_link_metrics`) for updated link metrics from the database, instead of querying each router separately as with the query means described above.

2.9.4 Control Plane: Testbed Deployment

As described in Section 2.8, the current framework has already been used to obtain results on both PlanetLab and Emulab and its source code is freely available for download and deployment. The next step is using the shared testbed features of the control plane (described in Section 2.5.6) to deploy the data plane as a routing service on PlanetLab and CoreLab. This would place all possible links on these testbeds at the disposal of routing researchers, would represent the largest application-directed data plane currently available and would allow experimenters to perform application-directed experiments more quickly and easily than is currently possible.

2.9.5 API: Packet Processing Module Structure

As described in Section 2.6.3.2, the current packet processing module chain structure supports two types module operation: *push-input/push-output* and *pull-input/pull-output*. Currently, these respectively correspond to output and input modules and are sufficient for implementing modules which perform an operation on a packet and immediately return it (or return an error). However, in order to implement more complex modules such as those which perform queueing, it would be more convenient to have an interface which supported *mixed mode* operation. Specifically, a *pull-input/push-output* configuration would allow a module to pull (input) a packet from a source, buffer it for some period of time, and then push (output) it; conversely, a *push-input/pull-output* configuration would allow a module to be pushed (input) a packet from a source, and then buffer (queue) it until a later pull operation unbuffers (dequeues) it.

In support of mixed mode operation, two new classes are currently proposed as an extension: `SoraPPMPushInPullOut` and `SoraPPMPullInPushOut`. As described in Section 2.6.3.2 and specifically in Figure 2.17, the former class inherits the `SoraPPM-PushIn` and `SoraPPMPullOut` classes and implements the `push` method, which calls a pure virtual `handle` method to be implemented in a subclass; the latter class inherits the `SoraPPMPullIn` and `SoraPPMPushOut` classes and implements the `pull` methods, which calls a pure virtual `handle` method to be implemented in a subclass. Thus, these pure virtual class interfaces are to be inherited in constructing processing modules that implement packet buffer queues, packet reorderers and the like.

2.9.6 API: Path Schedulers

As described in Section 2.6.3.2, the framework currently supports two path schedulers: round-robin and random. However, many scheduling algorithms, such as weighted round-robin, weighted fair queueing and the like exist [45] to provide better performance under a variety of network conditions. The current framework allows such schedulers to be implemented and make use of different monitored path metrics (e.g., latency, loss rate, bandwidth, jitter, reordering and the like or some combination of thereof). Therefore, another future extension to the framework is implementation of support for a larger variety of schedulers. As is also described in Section 2.6.3.2, new schedulers are implemented by creating a subclass of `SoraPathScheduler` and implementing the scheduling logic within it to return the next path in the given path set.

2.9.7 API: Packet Processing Modules

As described in Section 2.6.3.2, current packet processing modules include those for packet options setting (e.g., path, network, source and destination identifiers, sequence numbers, path latency and loss information, etc.), payload setting as well as monitoring of statistics such as packet send and receive, packet loss, relative path latency and packet reordering. Moreover, modules can be created to perform new tasks; in particular, two types of extensions are envisioned for the current infrastructure: (1) support for further monitored metrics such as jitter and throughput in support of the path schedulers discussed above; and (2) modules supporting features such as packet reordering and, for instance, insertion of artificial packet delay via the mixed mode module interface discussed above; insertion of packet delay is useful in testing and experimentation because it allows controlled amounts of latency to be added on demand.

2.9.8 API: Gbps Performance

As discussed in Section 2.7, the kernel-level router and conduit currently provide performance on par with native IP for 100 Mbps networks and the kernel-level router scales with native IP even for Gbps networks. However, as depicted in Figure 2.21, the conduit implementation suffers from the same type of characteristic bottleneck on Gbps networks that the user-level router does on 100 Mbps networks. As can be seen between the performance of the user-level and kernel-level data plane routers, a kernel-level implementation eliminates the extra data copy and greatly improves performance. Therefore, in order to support experiments on a wider variety of environments, another proposed extension to the current framework is a kernel-level conduit endpoint interface implementation. It is believed that this will place the performance of the conduit endpoint on par with the native IP used as a baseline and that processing performance will then only be limited by the number and type of modules used with it.

2.9.9 API: Wrapper Improvements

As described in Section 2.6.4, the current libc wrapper implementation for application data ingress and egress has three limitations: (1) limitation to UDP sockets, (2) limitation to a single socket per application and (3) lack of support for out-of-band messaging functions (i.e., those provided by `sendmsg` and `recvmsg`). In order to better support experiments with existing applications, another proposed extension is aimed at addressing these issues.

The challenge in supporting TCP/IP sockets lies in the need for a conduit-based reliable stream protocol implementation to replace the functionality lost when TCP/IP sockets are redirected to conduit endpoint instances (i.e., a conduit supporting error, loss detection, retransmission and reordering); once this is available, TCP/IP socket data communication functions can be mapped to the methods of this conduit implementation.

The challenge in supporting multiple sockets per application lies in coping with the fact that UDP/IP applications, for instance, can change the source/destination socket address pair of an existing file descriptor, on demand. Since conduit endpoint instances depend upon the socket address, when such socket address changes are possible, it no longer suffices to simply map file descriptors to conduit endpoints since a new conduit must be created for each socket address pair. One approach is to maintain a list of conduit endpoint instances mapped to a single file descriptor. Then, when a new socket address pair is used, a new conduit endpoint instance is created and inserted into the list; when a send or receive is performed, the file descriptor is used to search for the corresponding conduit endpoint list and the located list is searched for the conduit endpoint instance corresponding to the given socket address; when a close is performed on the socket, all conduit endpoint instances in the corresponding list are destroyed.

The challenge in supporting out-of-band messaging functions lies in mapping the out-of-band message data to data carried in the packet but outside of the payload. One approach is to map such data to a packet header option. Then, when a message bearing such data is sent by an application via the wrapper, it is sent through the conduit via the given option; when a packet bearing such an option is received by the wrapper from the conduit, the option data is passed to the application as out-of-band message data.

2.10 Conclusion

This work described the SORA multipath virtual network framework, including an application-directed data plane, a control plane for configuring use of the data plane on large-scale testbed networks such as PlanetLab and an application interface for use in research and development of new application-directed routing applications and transport layers.

The data plane was shown to provide: (1) forwarding performance so as not to be a bottleneck on said networks, (2) optional forwarding for multiple experiments simultaneously and (3) ease of upper layer development via the path conduit endpoint interface.

The control plane was shown to provide: (1) convenient data plane link, link metric and path discovery, (2) convenient router link instantiation and (3) optional setup and management of forwarding for multiple simultaneous testbed experiments.

The application interface was shown to provide: (1) convenient application-directed routing-aware application development and experimentation via a modular control and data plane interface which allows quick component swapin and swapout and a library for common tasks; and (2) usability with existing network applications and protocols via two methods: a libc wrapper and an ingressing/egressing router.

Moreover, Section 2.8 described specifically how the data plane, control plane and application interface of the framework can be used in experiments on two major testbed network targets (Emulab and PlanetLab); Chapter 3 describes a concrete usage case studying the use of unmodified TCP/IP on application-directed networks on Emulab; finally, the framework is currently being used to develop and test new application-directed routing-aware transport protocols. Thus, the present framework attains its core goal of providing an immediately usable solution to the problem of how to conveniently develop and test both existing and future applications and transport protocols on application-directed routing networks using the current network infrastructure, as proven both by both experiment and a usage case.

However, a number of items for future work were discussed in the section above: namely, *link types* for the data plane; *testbed network control*, *live router link monitoring* and *PlanetLab-based deployment* for the control plane; and a modified *packet processing module structure*, alternate *path schedulers* as well as *packet processing module* additions, *Gbps performance* and *wrapper enhancements* for the application interface. However, these represent possible functionality extensions and enhancements to the current framework and are not necessary to begin using the framework immediately. Accordingly, even though many areas for future work exist, the software framework is deemed to be useful because it has been shown by experiment as well as a usage case to meet its core goals.

Nevertheless, the real test of any software framework is whether or not users find it useful. Source code for `libsora`, the SORA Click router and the SORA Ruby control plane library as well as documentation for their use are freely available under the GNU Public License at `sora.nakao-lab.org`. Moreover, development continues on many of the future work items listed above and it is hoped that future user requirements will continue to guide new development.

CHAPTER 3

SORA USAGE CASE: STUDYING PACKET REORDERING

3.1 Introduction

Demands for a future Internet include greater reliability and performance than can be obtained with the current routing model. Multipath routing, whereby network paths may be selected and simultaneously used on a per-packet basis, is well-suited for meeting these requirements because of the superior failover and bandwidth resource usage freedom it allows. One typical application of multipath routing that has long been proposed for realizing such reliability and performance gains is packet dispersion [106, 107], wherein packets in a given flow are striped across multiple paths.

However, the simultaneous use of larger numbers of paths causes packets to be delivered increasingly out-of-order. The performance of TCP (the Internet's most commonly used transport protocol) is known to degrade under precisely the type of out-of-order packet delivery encountered using multipath routing applications such as packet dispersion [14, 5]. While many performance improvements to address this issue with TCP have been suggested and a few have even been implemented, out-of-order delivery is a main issue preventing multipath routing from becoming a viable means for improving Internet performance and reliability.

While research is ongoing into next-generation transport layer replacements for TCP, which are capable of coping with reordering and taking fuller advantage of the improved access to multiple paths expected from a future Internet, given TCP's near ubiquitous deployment in networked computing systems from desktop computers to embedded devices, complete replacement of TCP is not possible on a short time scale. *The reality is that TCP will continue to be used well after deployment of a future Internet is complete.*

Therefore, research into techniques allowing *unmodified TCP* to take advantage of multipath routing applications such as packet dispersion are highly relevant for sustaining Internet growth.

This chapter investigates the feasibility of using a *best-effort network layer packet reordering service* to mitigate poor TCP performance in the presence of varying degrees of out-of-order packet delivery. Herein, packet dispersion is adopted as one highly typical cause of out-of-order packet delivery. However, as mentioned above, a number of possible causes exist; it should be noted that the reordering service described herein as well as its implementation are capable of aiding any out-of-order packet stream and are not particularly limited to use with multipath routing. Moreover, as will be described in Sections 3.3 and 3.5, the reordering service embodiment described herein requires *no end-host modifications*. While previous work on the subject [92] only simulated out-of-order delivery at the ingress, did not characterize the type of reordering observed and only tested one TCP variant, herein, these shortcomings are addressed and results presented from experiments to test the performance of two common TCP variants under packet dispersion with differing numbers of paths and amounts of inter-path latency variance, as might be expected from use of packet dispersion on a wide-area network.

The next section discusses the causes of and remedies for out-of-order packet delivery; Section 3.3, defines what is meant by a best-effort reordering; thereafter, related work is presented. In Section 3.5, details are provided of the experimental testing environment; thereafter, a packet reorderer implementation made for these experiments is described; finally, experimental results are presented and discussed and the chapter concludes in Section 3.8.

3.2 Causes and Effects of Out-of-Order Packet Delivery

By definition, out-of-order packet delivery (OOD) occurs when a source host sends a first packet prior to a second but the second packet arrives at its destination before the first. It is now commonly introduced by a variety of factors, including network operator use of traffic engineering to exploit multiple internal paths as well as switches employing parallel processing [14].

In multipath environments and particularly with packet dispersion, OOD is generally caused by inter-path differences in latency. These differences can be described more formally as follows. Let P be a set of network paths from a given source host to a given destination host. Let the latency of any path $\alpha \in P$ at the time a given packet instance a is sent via it be given by $\alpha.\texttt{latency}$; let the inter-send time between two packets a and b be $\Delta_s(a, b)$, where positive values of $\Delta_s(a, b)$ imply a was sent before b. Then the inter-receive time $\Delta_r(a, b)$ between packets a and b, sent consecutively and traversing paths α and β, respectively, is given by:

$$\Delta_r(a, b) = \beta.\texttt{latency} - \alpha.\texttt{latency} + \Delta_s(a, b) \tag{3.1}$$

By definition, in-order delivery can be guaranteed if and only if $\Delta_r(a, b) > 0$; that is, if

$$\beta.\texttt{latency} - \alpha.\texttt{latency} + \Delta_s(a, b) > 0 \tag{3.2}$$

Therefore, the following must also hold:

$$\alpha.\texttt{latency} - \beta.\texttt{latency} < \Delta_s(a, b) \tag{3.3}$$

3.2.1 Approaches to Addressing Out-of-Order Packet Delivery

Equation 3.3 suggests three basic methods for ensuring $\Delta_r(a, b) > 0$: (1) *at the sender*, by delaying packets in order to ensure large enough $\Delta_s(a, b)$ to cope with relatively small $\beta.\texttt{latency}$; (2) *in the network* by manipulating $\alpha.\texttt{latency}$ and $\beta.\texttt{latency}$; and (3) *at the receiver* by holding out-of-order packet b until the packet a arrives—effectively increasing $\beta.\texttt{latency}$.

The first option—delaying packets at the sender—is only possible when consistent information regarding both α.`latency` and β.`latency` is available, which is generally not the case on large inter-domain networks such as the Internet; the second option—controlling the network latency to ensure proper delivery—is not feasible on networks like the Internet where there is no centralized control over intermediate links; herein, focus is placed on the third option—effectively reordering the packet stream at the receiver.

3.3 Best-Effort Reordering

Reliable, connection-oriented transport services such as TCP typically ensure an ordered packet stream by including a monotonically increasing sequence number within the packet header and buffering out-of-sequence packets while waiting for slower packets to arrive. Excessively delayed packets are generally handled via explicit retransmission requests to the sender.

Another method is to use the same sequence number-based buffering scheme but instead of requesting retransmission, to perform a *best-effort reordering*. Here, packets are held just as before but a process is repeated whereby packets are buffered in sequence number order until the packets they depend upon arrive *or until a maximum hold time threshold elapses* since receipt of the earliest-sent packet in the buffer. When this occurs, the packet or packets that the earliest-sent packet in the buffer depend upon are declared lost and the earliest-sent packet is removed from the buffer and sent. This process is then repeated using the next earliest-sent packet in the buffer.

This method is *best-effort* in that no attempt is made to provide *reliable* packet delivery, which would introduce significant latency via retransmission requests. Moreover, it is possible to tune the maximum amount of latency the process adds by adjusting the maximum hold time threshold value. It is thus amenable for use with protocols such as TCP which are already providing reliable packet delivery and are sensitive to added latency. An algorithm to implement this method is formally presented in Section 3.6.

Best-effort reordering can be realized by a number of means and in multiple contexts. For instance, it is possible to use the scheme with TCP by using TCP's sequence numbers directly for the reordering. However, this method is transport-layer-dependent—it will not, for instance, benefit UDP streams. Another method, applicable to any transport layer instance, is to have a local router encapsulate input stream (e.g., TCP or UDP) packets, within UDP/IP packets and add per-stream, monotonically-increasing sequence number information between the two headers. Here, UDP and TCP streams are distinguished by the five-tuple of protocol identifier, source and destination IP addresses and port numbers (from the packet header); sequence number information is then maintained for each stream at the ingress. This allows existing transport layer instances to be used without modification and is the approach adopted in the experiments discussed herein.

3.4 Related Work

3.4.1 TCP and Reordering

TCP has performed packet reordering since its inception. While optimizations for performance have long assumed a single-path environment with practically no out-of-order delivery, more recently a number of TCP modifications have been proposed or implemented [14]. Most notably, selective acknowledgment (SACK) [18] allows packets to be acknowledged out-of-order and duplicate selective acknowledgment (DSACK) [54] allows for better performance recovery after duplicate acknowledgments have triggered congestion control [19, 52]. Nevertheless, TCP performance under reordering is still suboptimal and research continues at the transport layer [174, 39, 20].

3.4.2 Packet Dispersion / Link Bonding

Data dispersion across disjoint paths or links in a communications context has been proposed at the data link, transport and network layers.

At the data link layer, the concept is often referred to as link bonding (or inverse multiplexing) and has been realized in conjunction with dialup and ISDN [141], ATM [40], and even wired and wireless Ethernet [100, 83]. While data link layer-based techniques are known to provide good performance, it is not generally possible to use them with disparate link technologies (e.g., ATM with Ethernet), which precludes their use in end-to-end contexts.

Another approach, adopted by Adiseshu et al., is to place a layer of software at the endpoints—logically between the data link and network layers—and have this software perform striping of packets over multiple interfaces [1]. This approach differs from that presented herein; while it operates with disparate link layer technologies, it assumes the existence of multiple network interfaces; moreover, it makes no active attempt to ensure delivery of an ordered packet stream to the transport layer.

A number of attempts have also been made to employ varying forms of packet dispersion at the transport layer; examples include pTCP [69], which demultiplexes application data onto multiple host interfaces and mTCP [175], which stripes application data across multiple network paths while still attempting to maintain reliability and fairness. While these approaches do treat reordering, unlike the method presented herein, they do so by replacing TCP—requiring endhost modifications.

Packet dispersion at the network layer was first proposed in the 1970s [106] and has been revisited a number of times since [66]. Most recently, it has been suggested for use in wireless contexts by Chebrolu et al. [26]. The goal of their work is to exploit multiple wireless interfaces on hosts by dispersing packets using them. They suggest addressing reordering by scheduling sends (a sender-side solution, as discussed in Section 3.2.1) and also employing reordering using a buffering network proxy near the receiver.

The work presented herein in introducing reordering services at the network layer differs from this work in several ways: first, their focus is on hosts with multiple network interfaces, whereas the focus of this work is not limited to this and assumes a future network wherein multiple network paths are available even on single interface devices. Second, the effectiveness of the approach presented herein is demonstrated using empirical results from actual application data running on several TCP variants. Finally, this work investigates a greater degree of packet dispersion (using up to 10 simultaneous paths, as compared with three interfaces) as well as varying degrees of out-of-order delivery.

3.5 Implementation Environment

The environment used to conduct the experiments presented herein is comprised of a SORA-based multipath routing network implemented on the Emulab network testbed; this section explains SORA running within Emulab, the experimental network topology and configuration used in these experiments as well as the method by which out-of-order delivery is introduced to the packet stream.

3.5.1 Emulab

The network environment used to conduct the experiments presented herein was implemented on the Emulab testbed [162], hosted at the University of Utah. Emulab is a network testbed environment that allows researchers to create and perform experiments on dedicated networks of their own configuration—that is, on networks whose topology and link characteristics such as latency and loss rate are configurable and comprising machines running operating systems and software of the experimenter's own choosing. Emulab allows dedicated access to computing and network resources so that experiments are not subject to interference by resource competition from external sources.

Throughput through the SORA routers used in the present Emulab experiments was only limited by the packet processing capabilities of the underlying hosts. These experiments were conducted using hosts running Linux 2.6.29 with dual 3 GHz processors; routers and ingresses ran Click 1.6 in user mode.

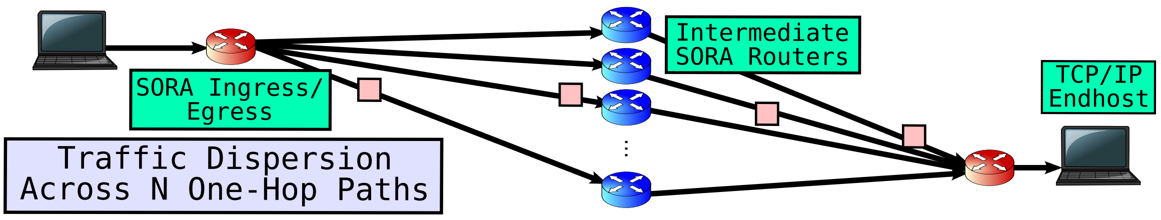

Intermediate SORA Routers
SORA Ingress/ Egress
TCP/IP Endhost
Traffic Dispersion Across N One-Hop Paths

Figure 3.1: Multipath Experiment Topology

3.5.2 Topology

In selecting a topology for the experimental network, the objectives were (1) to strictly control all sources of path latency and (2) to minimize factors which may secondarily contribute to TCP latency such as packet loss due to competition for shared resources (e.g., link capacity and buffer space at routers). Because such resource competition is exacerbated when paths are not edge- and vertex-disjoint, a multipath topology was chosen as depicted in Figure 3.1, wherein path latency can be strictly controlled by employing traffic shaping (on links adjacent to one of the ingress/egress routers) and resource competition can be minimized because all paths are edge-disjoint (i.e., share no non-duplex links) and only forward and reverse paths share a vertex (i.e., a router). To avoid having the maximum throughput achievable on the routers be a resource constraint, a maximum network throughput of 10 Mbps was adopted. To provide this with a maximum of 10 paths, the available bandwidth on each of the links adjacent to the ingress was set to 1 Mbps.

Finally, adopting round-robin path scheduling would result in consistent use of the same pattern of paths on the given network and thus the same packet latency pattern. Because this could result in differing results if the same latencies were used in a different pattern, *random* scheduling was employed, whereby the next path is scheduled randomly from among the paths in the current set. This eliminates the issue of a given scheduling pattern biasing the results, without changing the statistical distribution of scheduled paths.

3.5.3 Introducing Out-of-Order Packet Delivery

To study the effects of reordering on TCP throughput, a simple method was required to gradually increase the severity of out-of-order delivery introduced by the network through modification of the respective path latency values. Here, a gradual increase in latency variance from zero was sought, in which case the only reordering is that introduced by variations in the native network latency, to greater degrees of variance

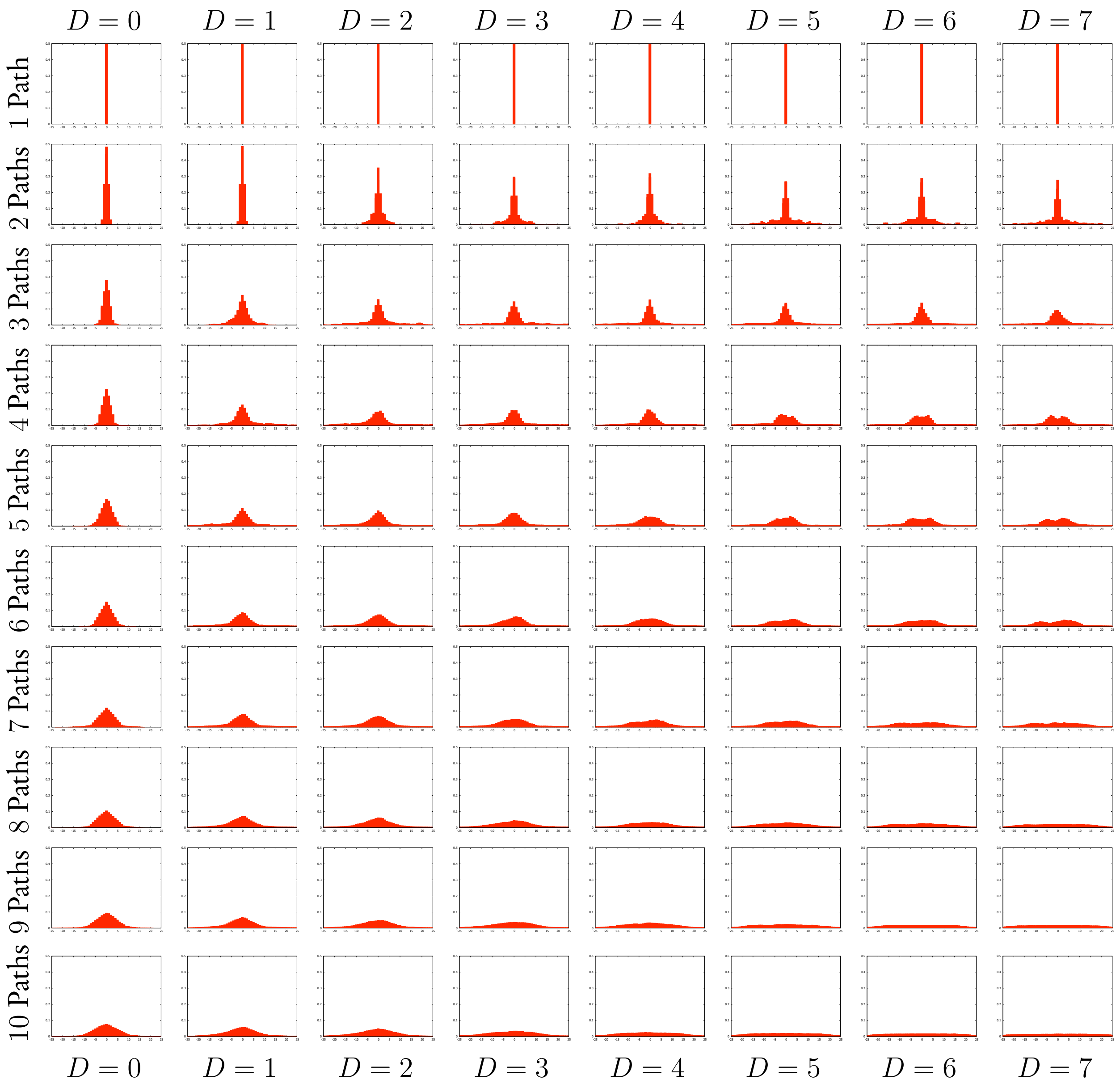


Figure 3.2: Effect of Varying Dispersion Degree and D Value on Reordering Density

and thus reordering, but with most cases still consistent with what might actually be experienced on a planet-wide network such as the Internet.

Accordingly, a set of delay configurations was adopted wherein the variance of the latency of the paths in the path set was gradually increased. To accomplish this, the paths in a full path set P were arbitrarily numbered from $i = 0$ to $|P| - 1$ and used a variable D defined over the positive integers to set the latency of path number i to $D * i$ msec. For example, with $D = 3$, the latency configuration on a four path set (p_0, p_1, p_2, p_3) would be $(0, 3, 6, 9)$. While Internet latency values are unlikely to be constant, this scheme may be viewed as application of the oft-used technique of adopting a mean value to model an unknown distribution.

While it is clear with this configuration that the statistical variance of the respective sets of latency values increase smoothly with D, an investigation was required to examine the extent of increase in out-of-order delivery said variance actually induced in the received packet stream. In the next section one metric which characterizes reordering is briefly examined and applied to this method.

3.5.4 Characterizing Out-of-Order Packet Delivery

A number of formal proposals have recently been made for metrics quantifying the severity of packet reordering [111, 76] induced by a network. RFC 5236 defines a statistical distribution that characterizes reordering and has been shown effective and readily implementable [123]: a *reordering density distribution*.

A *reordering density distribution* is a measure of how severely packets in a stream are displaced from their expected (in-order) arrival positions. It is a probability distribution indicating the likelihood that an arriving packet will be displaced a given number of arrival positions away from its expected position (zero). Thus it conveys both the likelihood of out-of-order delivery and the extent to which packets are displaced from their expected position. Note that in its computation, lost (or excessively delayed) packets are not counted as part of the reordering density distribution. In the reordering density distributions depicted in Figure 3.2, the x-axis represents the number of packet arrival positions distant a packet arrived from its expected position; the y-axis represents the frequency at which packets arrived a given distance from their expected positions.

Passive measurement of this metric was implemented on a given packet stream within the SORA multipath routing framework and the effects of the present method for introducing latency variance while dispersing packets over a 10 path network was measured. Reordering density was observed at the destination egress as both D and the dispersion degree (i.e., the number of simultaneous paths used) were increased. A portion of the results are summarized in Figure 3.2; therein, the top row depicts the $D = 0$ path latency configuration, wherein reordering is induced solely by the network itself as the number of paths is increased. In the two path case (far left) packets arrive nearly in order and the likelihood of out-of-order delivery increases as the number of paths increases (to the right). In the second row, D is increased to 2 and the likelihood of out-of-order delivery can be seen to increase as the histogram is seen to gradually "flatten out"; this trend continues in the final row with $D = 4$. Thus, the D-based configuration does produce a gradual transition from a reordering distribution wherein most packets arrive in sequence to one where packets begin to arrive significantly out of sequence.

3.6 Reorderer

The reorderer used in the experiments presented herein is a component of SORA and is implemented in C++. It relies on sequence numbers assigned by a SORA ingress and stored in the SORA packet header and uses a min-heap to buffer packets. A min-heap is a data structure internally organized such that the minimum heap element is always available for removal in $O(1)$ time and insertion (to maintain this property) requires $O(\lg n)$, where n is the heap size. Herein, the `MinPktHeap` class is used to describe the functionality of this heap; it has instance methods `insert`, `min`, `remove_min`, `is_empty` and `drain`, which respectively allow packet insertion, return and removal of packet with minimum sequence number, confirmation of whether the heap is empty and "draining" of the heap (described below). The reorderer uses two execution threads: an asynchronous packet receiving thread (i.e., running in the same thread as the packet router), which receives packets and places those received out of order into the heap, and a synchronous thread, which periodically "drains" the heap of expected or excessively delayed packets.

The algorithm which defines this "draining" of the packet heap is depicted in Algorithm 1 as an instance method of the `MinPktHeap` class. Here, E is the currently expected packet sequence number; M is the maximum hold time threshold value for the earliest-sent packet; the sequence number and receive time (e.g., in msec) of a given packet p are p.`seq_num` and p.`recv_time`, respectively; and `curr_time` indicates the current time. While the packet at heap top is the expected packet or has been waiting longer than M, the algorithm loops over the heap; within the loop body, it removes the packet at heap top and inserts it into the output queue if its sequence number is greater than or equal to E and drops it otherwise.

As depicted in Algorithm 2, the heap draining thread repeatedly runs the `drain` method on the given heap to remove packets from the packet buffer as necessary and, when no packet is buffered, sleeps until the packet with earliest sequence number must be removed or until it is awoken by the packet processing thread.

Here, S is a signaling semaphore that implements mutual exclusion locking and signaling. It is similar in spirit to the POSIX semaphore structures `pthread_mutex_t` and `pthread_cond_t` and their associated functions, as used in the implementation. The `Signal` class implements instance methods for blocking lock acquisition (`acquire_lock`) and release (`release_lock`), signal sending (`signal`), blocking sig-

Algorithm 1: `MinPktHeap` Method `drain`

```
   Input: PktQueue $O$, SeqNum $E$, Time $M$
1  while min.seq_num $==$ $E$ $||$
2      min.recv_time $+ M <$ curr_time do
3      $p =$ remove_min
4      if $p$.seq_num $\geq E$ then
5          $O$.enqueue($p$)
6          $E = p$.seq_num $+ 1$
7      end
8  end
```

nal receipt (await_signal) and blocking signal receipt with a maximum time value (await_signal_until). Packet queue class PktQueue implements instance methods for blocking packet read (dequeue) and non-blocking write (enqueue); enqueueing a packet into the output queue O causes it to be decapsulated and forwarded to its destination (an unmodified endhost).

More specifically, Algorithm 2 repeats the process of: (1) outputting the packet at heap top which depended on the newly-arrived packet or has been waiting longer than maximum hold time threshold M, (2) sleeping until such time as a packet at heap top has been waiting longer than M or (3) sleeping indefinitely or until a packet is inserted in the heap and it is awoken by the packet processing thread.

In Algorithm 3, the packet processing thread dequeues the next packet from packet input packet queue I and outputs it to the output packet queue O if the heap H is empty; else, it inserts the packet into H and notifies the drainer that there is now a packet waiting. The drainer wakes, runs drain on the heap and waits until the maximum hold value has expired for the given packet or until it is awoken again.

3.6.1 Maximum Buffer Hold Time

Determination of the maximum hold time value (M, above) is dependent upon the maximum possible end-to-end latency—the maximum amount of time a packet could be delayed in the network. Suboptimal selection of this value can hinder performance in two ways: (1) too large a value may needlessly increase network latency when packets are dropped in the network; and (2) too small a value may degrade performance when

Algorithm 2: Heap Draining Thread

```
  Input: MinPktHeap $H$, SeqNum $E$, Time $M$, Signal $S$
1 while true do
2     $S$.acquire_lock
3     $H$.drain($E$, $M$)
4     if $H$.is_empty then $S$.await_signal
5     else $S$.await_signal_until($H$.min.recv_time $+ M$)
6 end
```

Algorithm 3: Packet Processing Thread

```
Input: PktQueue $I$, PktQueue $O$, Time $M$
1  MinPktHeap $H$
2  SeqNum $E = 1$
3  Signal $S$
4  while true do
5      $p = I$.dequeue
6      $S$.acquire_lock
7      if $p$.seq_num $==$ $E$ && $H$.is_empty then
8          $O$.enqueue($p$)
9      else
10         $H$.insert($p$)
11         $S$.signal
12     end
13     $S$.release_lock
14 end
```

packets which were *not* dropped in the network but *only delayed* are effectively dropped by the reorderer.

In the absence of precise latency information regarding the path an expected packet is to traverse, an ideal balance between these two cases is a maximum hold time just larger than the maximum possible packet inter-receive time; that is, just larger than the difference between the largest and smallest possible path latency values. This value is ideal because it minimizes the amount of time spent waiting for dropped packets in the first case, and guarantees no packets will be effectively dropped by the reorderer in the second case. One widely-used estimate for minimum and maximum one-way latency values is half the minimum and maximum round-trip times between the two hosts, which can be obtained by continual monitoring.

When packet loss is persistent and relatively high, the reorderer tends to slow all packets to the latency of the slowest path and, as noted above, adds a correspondingly high degree of both latency and burstiness to the data stream, which is known, for instance, to cause TCP throughput reductions. Such packet loss is commonplace in many environments, *wireless* networks being one example. Accordingly, *a best-effort reordering service like that investigated herein is likely not well suited for such environments,* and it may be best to disable the service automatically when such loss is detected and reenable it after recovery.

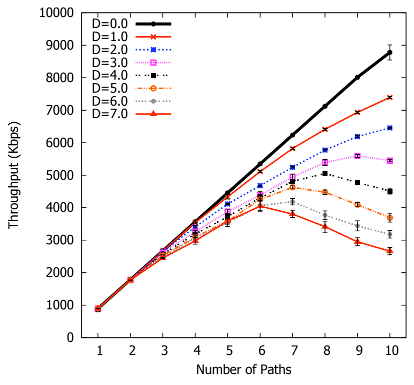
However, *in environments with negligible packet loss*, only the second case (selecting too small a value) is of relevance and there is little penalty for selecting too large a value for the maximum hold time. Thus, an extremely accurate estimate of the ideal value is unnecessary and it suffices to select any value highly likely to be larger than the maximum possible latency value (e.g., double the maximum observed latency).

The goal herein is to investigate what is possible with a simple best-effort packet reordering service in largely error- and loss-free environments like those provided by current wired networks, where loss generally only signals congestion. Accordingly, in the experiments described herein, the maximum hold threshold was set such that it was *always greater than the maximum possible inter-arrival time*. While this is possible here because the latency values were known, they may also be probed dynamically and updated according to changes during data transmission, making this approach feasible even when they are not known in advance.

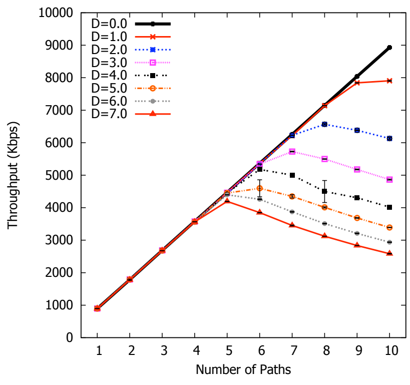
3.7 Results

The throughput achieved was tested with and without the use of the reorderer on two common TCP variants: Reno [43] and CUBIC [131], using the implementations of the Linux 2.6.29 kernel (note that these include NewReno [53] extensions). In all tests SACK and DSACK were enabled, and congestion window caching between TCP connections was disabled to avoid carry-over between experiments. TCP throughput was measured using the network performance utility `netperf` [77]; each data point is an average of five 30 second tests with error bars indicating the standard error in each test set. Let D be the delay configuration, as described in section 3.5.3 and N be the degree of packet dispersion (i.e., the number of paths used simultaneously). The results are summarized in the plots of Figure 3.3.

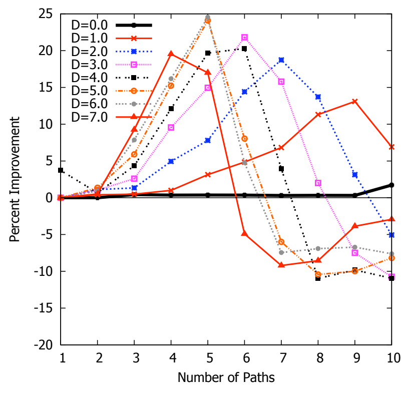
Therein, the plots in the leftmost column depict cases where packet dispersion was used *without* best-effort reordering; those in the middle column depict cases where packet



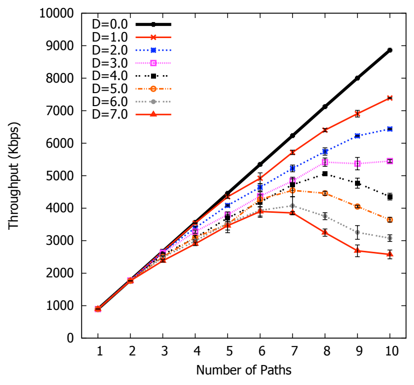
(a) Reno w/o RO, $W_{\mathtt{max}} = 32$ KB



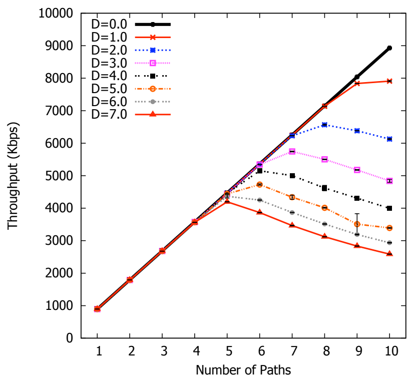
(b) Reno w/RO, $W_{\mathtt{max}} = 32$ KB



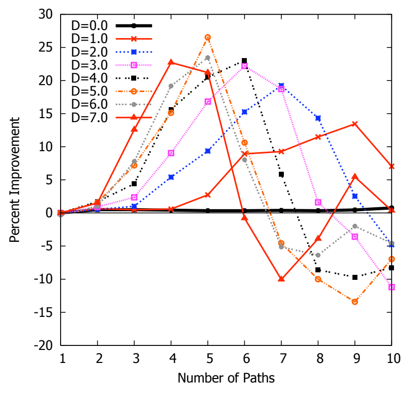
(c) Reno Speedup, $W_{\mathtt{max}} = 32$ KB



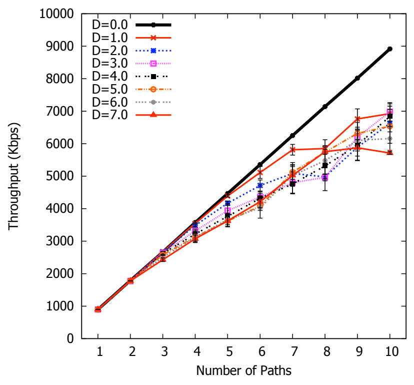
(d) CUBIC w/o RO, $W_{\mathtt{max}} = 32$ KB



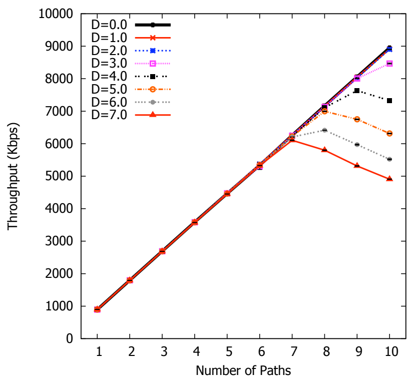
(e) CUBIC w/RO, $W_{\mathtt{max}} = 32$ KB



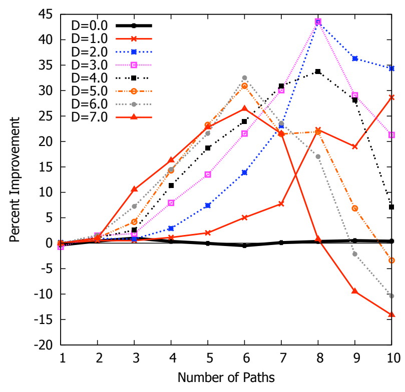
(f) CUBIC Speedup, $W_{\mathtt{max}} = 32$ KB



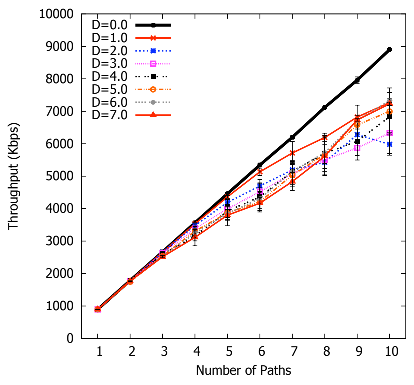
(g) CUBIC w/o RO, $W_{\mathtt{max}} = 64$ KB



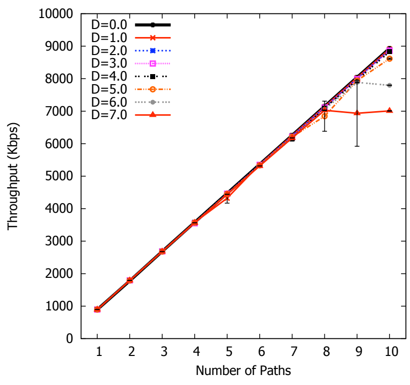
(h) CUBIC w/RO, $W_{\mathtt{max}} = 64$ KB



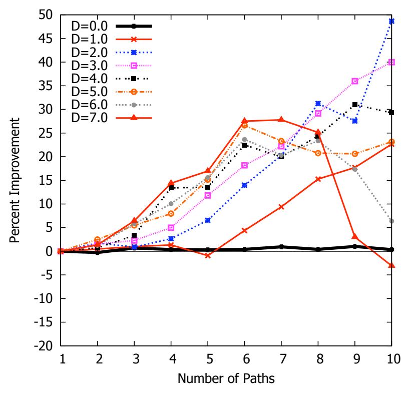
(i) CUBIC Speedup, $W_{\mathtt{max}} = 64$ KB



(j) CUBIC w/o RO, $W_{\mathtt{max}} = 96$ KB



(k) CUBIC w/RO, $W_{\mathtt{max}} = 96$ KB



(l) CUBIC Speedup, $W_{\mathtt{max}} = 96$ KB

Figure 3.3: Throughput Results With Varying Maximum Window Sizes

dispersion was used *with* best-effort reordering (note that "RO" stands for "reordering"). In the plots of both the left and middle columns, the x-axis is the number of paths used simultaneously for dispersion, the y-axis is the throughput obtained, and each line represents the throughput obtained using the value of D indicated in the legend. The plots in the rightmost column depict the speedup obtained by using the reorderer, calculated by dividing the throughput obtained with best-effort reordering by the throughput obtained without using it.

The first and second rows of plots depict the cases of Reno and CUBIC, respectively with a maximum window size $W_{\texttt{max}}$ of 32 kilobytes, a commonly used default socket buffer size; this setting was effected by setting the maximum TCP socket buffer size via `netperf` options, which use the `setsockopt()` system call to modify maximum socket send and receive buffer sizes. Notably, there is little difference between the performance of Reno and CUBIC—either with or without best-effort reordering. This was found to be the case with all the tests performed; while the degree of similarity is surprising, since both detect lost packets via the same threshold on the number of duplicate acknowledgements received, this is not entirely unexpected. Accordingly, since the results were also near identical for higher window sizes, for brevity, results are depicted only for CUBIC for the 64 and 96 kilobyte cases in the third and fourth rows, respectively.

3.7.1 Discussion

First, in the 32 kilobyte cases without best-effort reordering (Figures 3.3a and 3.3d), near linear performance is seen with respect to N in the case of $D = 0$, and for all D with $N \leq 2$. For $N \geq 3$ and $D \geq 1$, however, performance degrades as out-of-order delivery increases with D and N. Characteristically, the throughput curves tend to be concave—gradually rising to a given peak value and gradually falling thereafter—for example, the case of $D = 7$ peaks at $N = 6$, with a throughput of nearly 4 Mbps and gradually falls to just under 3 Mbps at $N = 10$.

However, as depicted in Figures 3.3b and 3.3e, the use of best-effort reordering extends this near linear performance with respect to N to all cases of $D \leq 5$. Nevertheless, thereafter the same type of performance degradation is visible as in the unreordered case. This is also clear from Figures 3.3c and 3.3f, where the speedup provided by best-effort reordering increases markedly for $N \geq 3$, particularly for larger D values but drops off sharply thereafter.

To understand this behavior, the effects of out-of-order delivery must be examined. For a given N, the best-effort packet reorderer must accumulate a group of packets and wait until the longest delayed among them is received—approximately $D * N$ msec plus the delay provided by the underlying network; it then sends the accumulated (and correctly ordered) group of packets out in one burst. Repetition of this process leads to packet bursts sent at regular intervals based on the $D * N$ msec maximum wait time. In the case of the best-effort reorderer, as D and N increase, the effective latency using packet dispersion tends to increase to the latency of the slowest path—roughly, $D * N$ msec. It has been shown by Padhye, et al. [121] that, with negligible packet loss, effective TCP throughput can be approximated by $W_{\mathtt{max}}/\mathtt{RTT}$. Therefore, it is reasonable to hypothesize that the latency introduced by waiting for the slowest path (i.e., the increase in RTT) is the source of the throughput reductions in the cases of larger D and N and that this can be overcome by increasing the maximum window size.

To test this hypothesis, the same experiment as above was conducted, but the maximum window size was increased from 32 kilobytes to 64 and then 96 kilobytes. It should be noted that these values are within the norm of those now in use: default socket buffer sizes now commonly range to hundreds of kilobytes, particularly on high bandwidth/high latency networks. The results are depicted in the plots of the third and fourth rows of Figure 3.3. First, notably the graphs in the cases without best-effort reordering (Figures 3.3g and 3.3j) are no longer concave, indicating that latency is no longer dominating $W_{\mathtt{max}}/\mathtt{RTT}$ for larger D and N. The respective peaks in the throughput curves in the reordered case with a 64 kilobyte window size are significantly higher than in the 32 kilobyte case, and in the 96 kilobyte case all but a few results ($D \geq 6$,

$N \geq 9$) provide throughput near linear with respect to N, indicating that this hypothesis is correct. Moreover, the speedup plots of the 64 and 96 kilobyte cases (Figures 3.3i and 3.3l) show the peak speedup moving to higher dispersion degrees—indicating that speedup over the non-reordered case is increasing with the number of paths. Further, while slowdowns (negative speedups) are not infrequent with smaller window sizes, they are negligible in the 96 kilobyte case. Because the throughput obtained at a maximum window size of 96 kilobytes was nearly identical to that obtained with zero added delay, further increases were unnecessary.

Thus, if the window size can be increased to compensate for the latency added by best-effort reordering it is possible to insulate TCP from the performance-degrading effects of even heavy reordering.

3.8 Conclusion and Future Work

This chapter discussed a method for insulating TCP from the performance-degrading effects of out-of-order packet delivery—a reality in multipath routing networks and particularly in those employing packet dispersion. It discussed the causes and effects of out-of-order delivery on TCP, provided an overview of a multipath testbed implementation as well as a reorderer implementation; it discussed a metric for quantifying the severity of reordering of the packet stream and used it to demonstrate that the method of adding increasing amounts of latency to the respective paths in the testbed gradually induced out-of-order packet delivery.

The results presented herein indicate that a simple reordering service implemented on ingress/egress routers can mitigate performance degradation and provide consistent performance without requiring endhost modifications: throughput was improved in nearly all test cases using a 96 kilobyte maximum send window, with speedups of more than 45% and most between 10% and 25%. While non-linear speedups as well as slowdowns were also encountered in some cases with smaller maximum window sizes, the results indicate that linear speedups can be expected if the window size can be increased to compensate for the latency added by the reorderer.

Given the role multipath routing networks may play in a future Internet, the widespread use of TCP and the difficulty of upgrading all existing devices, methods for helping TCP to take better advantage of multipath routing by coping with out-of-order packet delivery deserve further research. Examples include a method for dynamically optimizing the maximum hold time to allow correct classification of lost packets, methods for TCP-friendly path scheduling to avoid reordering and investigations of other delay configurations and topologies.

CHAPTER 4
CONCLUSION

It has been shown that a number of shortcomings exist in the interdomain routing regime of the current Internet which hinder its growth as well as the implementation of reliable, high performance services using it. These include poor control over interdomain traffic (e.g., selection of AS paths is limited to the options provided by neighbors), slow recovery in the face of faults (e.g., three minute average, 15 minute observed worst case and *minimum* 30 second projected convergence times), an inability to respond to differing application requirements (e.g., one-size-fits-all routing, where applications, endsystems and even endnetworks have no input regarding the path a packet is to traverse) as well as economic issues such as an inability for content producers to receive payment for the bandwidth content consumers consume when accessing their services, making many business models difficult or impossible, and an inability for ISP's to always be able to match user network charges with user traffic usage rates.

It has been demonstrated that many of these shortcomings have their roots in: (1) the Internet's conflation of control and data planes, (2) the Internet's dependence upon convergence and hop-by-hop routing and (3) the Internet's current payment structure. The conflation of control and data planes refers to the fact that nearly all forwarding decisions (e.g., best path decisions) are made by the same machines that perform the actual forwarding—the routers themselves. This has forced network architects to use hop-by-hop routing and distributed best-path computation protocols that require convergence. Hop-by-hop routing makes it difficult for network operators to perform traffic engineering or to set routing policies. Moreover, the dependence on convergence leads to slow recovery times because the result of each round of the best-path computation must be computed and redistributed. Finally, the problems described above with the current economic regime have been explained by its failure to allow payment for network services to flow along the entire path that data traverses.

Much of the research aimed at improving Internet policy, performance, reliability and even economic competition shares a common feature—it directly or indirectly provides some form of application-directed routing, whereby applications (and thus their users) are allowed choice in the paths their packets are to traverse. Therefore, it is reasonable to assume that, whatever architecture is adopted in a future Internet, it will allow applications choice in the paths their packets are to traverse.

However, no architecture exists to provide for *all* the requirements necessary to implement application-directed routing: a *data plane* to move packets across the network, a *control plane* to manage the data plane and an *application interface* to interact with both. While a number of technologies exist which could be used to implement an application-directed data plane (e.g., ATM, G/MPLS, IPv6 source routing and even overlay routing networks), there exists no convenient method to use them for application-directed research and development. Moreover, at present, no viable control plane exists to realize application-directed routing and no implementations or proposals exist for an application interface.

Moreover, even once data and control planes allowing some form of application-directed forwarding are available and an application interface is created for use with applications, a number of serious issues remain open problems; these include *path scheduling algorithms*, *path monitoring metrics and accompanying analysis algorithms*; multipath-aware *best path computation algorithms*; *means to prevent route oscillation*; and *means for allowing existing IPv4 and IPv6 applications to enjoy the benefits of path selection*.

However, it is difficult to perform the essential research required overcome these challenges without a data plane, control plane and application interface usable on current networks on which to perform application-directed routing research and development. While large-scale testbed networks exist which could aid in such research, what is currently lacking is the environment that would allow research and development of application-directed routing.

4.1 SORA Multipath Virtual Network Layer

The primary objective of SORA is to provide an R&D infrastructure for application-directed routing on large-scale testbeds; it comprises a *data plane* for executing application-directed forwarding, a *control plane* for management of said data plane, and an *application interface* to the application-directed data and control planes.

4.1.1 Data Plane

The data plane provides for application-directed packet forwarding on testbed networks; it has three general requirements: (1) forwarding performance so as not to be a bottleneck; (2) minimal network setup burden and router resource waste via *optional* support for routing for multiple experiments simultaneously; and (3) ease of development of upper layers.

SORA's data plane adopts the well-known technique of using virtual networks composed of tunnels (e.g., overlay networks) for data plane construction; it then allows applications to select from lists of such tunnels (paths) to send packets by. Because multiple virtual networks can exist on the same network, each with different topologies and routing structures [12], adoption of a virtual network-based structure allows the ability to optionally forward for multiple experiments simultaneously, satisfies the second requirement (configuration of shared data planes is handled by the control plane).

However, tunnels complicate network usage because there are a variety of tunnel types that can be in use. In order to provide for the third requirement of ease of development, SORA provides two data abstractions: a *link* abstraction, which hides the details of tunnel setup, teardown and data send and receive on various tunnel types; and a *packet* abstraction, which allows payload and control (options) data to be sent on links.

Implementations of these abstractions as well as supporting data types and methods are gathered in a shared library called `libsora`. Based on `libsora`, a full set of modular router elements for implementing SORA routers using the Click modular router framework were implemented. The router is currently functional in both kernel and user-level modes for operation on PlanetLab and Emulab environments as well as most any modern Linux version.

The packet processing performance of the data plane was analyzed; the results are presented in Section 2.7.1. Analysis indicates that the kernel router packet processing performance scales with IP on both 100 Mbps and Gbps environments, but the user-level router may present a bottleneck for smaller packet sizes on 100 Mbps and is not well suited for full Gbps networks. Thus the data plane's packet processing performance does not present a throughput bottleneck when kernel access is available (e.g., Emulab). Moreover, bitrates on the major target testbed environments of interest (e.g., PlanetLab, OneLab) currently do not yet exceed 100 Mbps, therefore the data plane performance meets its performance requirements on these networks.

4.1.2 Control Plane

The control plane requirements are: (1) convenient link, link metric and path discovery, (2) convenient router link instantiation and (3) minimization of network setup burden and router resource waste by *optionally* supporting setup and management of forwarding for multiple testbed experiments simultaneously. In particular, the challenge in supporting the third requirement lies in creating a mechanism for sharing the resources of an existing testbed such as PlanetLab for the creation of many virtual networks, while ensuring only authorized use of the testbed.

The control plane is implemented via a Ruby-based programming library, which includes link, path and supporting data type implementations as well as methods for their manipulation. The control plane implementation provides: (1) *RPC-queryable link database, link metric and path computation services*; (2) *RPC-queryable link instantiation service*; and (3) a means for providing testbed router and link sharing by *pre-instantiating all testbed links and centrally authenticating all other (testbed-external) links*, thus allowing approved links to be instantiated by experimenters directly and avoiding the need for any central link instantiation or storage.

The performance of the control plane link and path discovery and link request service components were analyzed; the results are presented in Section 2.7.2. Analysis indicates that the link database is capable of providing fast retrieval for all link database sizes tested and sub-one-minute insertion times for networks of 10,000 links, which is believed to be suitable for all current testbed networks. The path discovery service is capable of computing up to 10 disjoint paths through graphs with nearly 8,500 vertices and 40,000 edges in under 13 seconds, with three paths possible through the largest graph in five seconds, which is also believed to be suitable for all current testbed networks. The link request service was capable of servicing 2,000 link requests in under one minute (an average of 51.83 seconds); given that in shared environments, most links are pre-instantiated and link requests are only required for external virtual network entities, this is also deemed suitable for use with even the largest of current testbeds.

4.1.3 Application Interface

The requirements for the application interface are: (1) ease of application-directed routing-aware application development and experimentation with the data and control plane; and (2) usability with existing network applications and protocols.

Ease of development and experimentation was provided via the conduit endpoint abstraction: a modular programming interface, allowing quick addition and removal of packet processing components as well as a library for common tasks such as packet loss and path latency monitoring and feedback. Usability was provided for via two means: (1) a libc wrapper, which automatically instantiates a conduit endpoint and performs send and receive via the conduit when inbound and outbound traffic match certain parameters (e.g., IP address or port number); and (2) an ingressing router, which is an extension to the Click user-level data plane router and acts as a default gateway for hosts on a network and also instantiates a conduit endpoint and performs virtual network ingress when outbound traffic matches certain parameters.

The packet processing performance of the conduit endpoint interface was analyzed; the results are presented in Section 2.7.1. For access via the ingressing router, since it runs using the user-level router, it shares the performance bottleneck of the user-level router. For access via the libc wrapper, analysis indicates that the conduit is capable of scaling with native IP for all packet sizes on 100 Mbps networks. However, while the conduit interface is capable of Gbps speeds for nearly half the range of packet sizes tested, for smaller packet sizes on Gbps networks, the conduit does present a bottleneck. Thus the packet processing performance of the libc wrapper-based conduit endpoint interface does not present a throughput bottleneck for testbed networks such as PlanetLab, which do not yet operate at Gbps speeds and is thus deemed to meet the requirements for their use; however, it may not be well suited for full Gbps testbeds.

4.2 SORA Usage Case: Best-Effort Network Layer Reordering

The SORA prototype (API, conduit endpoint interface and router) was used to extend multipath benefits to existing IPv4 applications and transport protocols such as TCP and UDP via application-directed routing while insulating them from the known detrimental effects of packet reordering via a scheme called best-effort network layer reordering.

Here, the automated ingress and egress feature discussed above was used to divert traffic onto a SORA virtual network and a min-heap-based buffer was added to the conduit interface in order to buffer early-arriving packets at the receiver, while waiting for later arrivals, with the goal of delivering an in-order packet sequence to the waiting endhost. In order to cope with packet loss, a configurable maximum hold timer controlled how long packets should be waited for in the queue before giving up on them and passing along the next-received packet. Results indicate that, when sufficient TCP window size is available to cope with the network latency, best-effort network layer reordering is able to improve performance by more than 45% with typical gains in the range of 15 to 25%.

4.3 Future Work Summary

While, as described above, the current framework offers an immediately usable solution to the core problem of how to conveniently develop and test both existing and future applications and transport protocols on application-directed routing networks using current networks, a number of areas exist for improving and extending the current infrastructure. These are summarized below but are described concretely in Section 2.9.

4.3.1 Data Plane: Link Types

The data plane framework currently implements two link types: UDP/IP and TCP/IP. One natural step forward is extending the current set of supported links to include GRE and IP-in-IP, among others, in order to allow more convenient use on testbed networks such as VINI [12] and CoreLab [115], which support optional data plane construction using GRE tunnels and GENI, which uses OpenFlow [108] for network control.

4.3.2 Control Plane: Extensions for Testbed Network Control

The current framework is intended to extend existing testbed control and data planes, allowing creation and operation of application-directed data planes atop them; it is not intended to replace them. However, one next step toward easing data plane setup on testbed networks is to add extensions to the current control plane to *directly* support link-based network setup and management of testbed networks. For instance, such an extension would allow direct instantiation of a testbed network from a virtual network link set. Moreover, on testbed networks (e.g., Emulab) which support setting of network characteristics (link metrics) such as latency or throughput, it would also allow live manipulation of such metrics.

4.3.3 Control Plane: Extensions for Live Router Link Monitoring

The current framework is intended to enable application-directed routing, which is defined as a packet routing method, whereby applications are given access to an arbitrary number of paths and are made fully responsible for enacting a routing policy as well as making performance decisions by monitoring path performance metrics; thus, link metric information is not used for path scheduling in application-directed routing. However, since live link monitoring information could conceivably be of use in examining different path scheduling schemes, another possible control plane extension is live router link monitoring, whereby routers monitor a set of links and provide monitored data to entities.

4.3.4 Control Plane: Testbed Deployment

The current framework has already been used to obtain results on both PlanetLab and Emulab and its source code is freely available for download and deployment. The next step is the deployment of shared testbed features of the control plane (described in Section 2.5.6) to establish a shared data plane as a routing service on testbed networks such as PlanetLab and CoreLab.

4.3.5 API: Packet Processing Module Structure

The current packet processing module chain structure supports two types module operation, which respectively correspond to output and input modules; it is sufficient for implementing modules which perform an operation on a packet and immediately return it (or return an error). However, in order to implement more complex modules such as those which perform queueing and buffering, it would be more convenient to have an interface which supported *mixed mode* operation: specifically, a *pull-input/push-output* configuration would allow a module to pull (input) a packet from a source, buffer it for some period of time, and then push (output) it; conversely, a *push-input/pull-output* configuration would allow a module to be pushed (input) a packet from a source, and buffer (queue) it until a later pull operation unbuffers (dequeues) it.

4.3.6 API: Path Schedulers

The framework currently supports two path schedulers: round-robin and random. However, many scheduling algorithms exist to provide better performance under a variety of network conditions. Therefore, another future extension to the framework is implementation of support for a larger variety of schedulers.

4.3.7 API: Packet Processing Modules

As described above, some conduit endpoint interface packet processing modules are provided for basic tasks. However, one natural addition is further packet processing modules to support, for example, monitoring of conduit performance metrics beyond loss rate and latency, to include, for example, jitter.

4.3.8 API: Gbps Performance

As discussed above and demonstrated in Figure 2.21, the conduit implementation suffers from the same type of characteristic bottleneck on Gbps networks that the user-level router does on 100 Mbit networks. As can be seen between the performance of the user-level and kernel data plane routers, a kernel-level implementation eliminates the extra data copy and greatly improves performance. Therefore, another next step forward is a kernel-level conduit implementation, which is expected to be on par with the native IP used as a baseline and only limited by the number of options it is made to process.

4.3.9 API: Wrapper Improvements

As described in Section 2.6.4, work is also progressing on addressing the current conduit endpoint interface libc wrapper limitation to use with UDP. This mainly involves two components: (1) the addition of reliable stream-oriented transport to SORA; and (2) mapping socket address pairs to conduits.

4.4 Summary

This dissertation described the SORA multipath virtual network layer; it provides: (1) a data plane supporting fast application-directed forwarding on a variety of current networks; (2) a control plane to allow convenient setup and management of the data plane, including a shared mode allowing resource usage reductions and easier virtual network setup and usage when multiple virtual networks use the same testbed; and (3) an application interface which eases application-directed routing-aware application development and experimentation with the data and control plane and provides usability with existing network applications and protocols.

While a number of areas for future work exist with respect to improving and extending the current framework defined by these components, the framework offers an immediately usable solution to its core problem of how to conveniently develop and test both existing and future applications and transport protocols on application-directed routing networks using current large-scale testbed networks, as was proven both by experiment and usage case. Source code for its components as well as documentation for their use are freely available under the GNU Public License at `sora.nakao-lab.org`.

BIBLIOGRAPHY

[1] Adiseshu, Hari, Varghese, George, and Parulkar, Guru. An Architecture for Packet-Striping Protocols. *ACM Trans. Comput. Syst. 17*, 4 (1999), 249–287.

[2] Agarwal, S., Chuah, Chen-Nee, and Katz, R.H. OPCA: Robust Interdomain Policy Routing and Traffic Control. *2003 IEEE Conference on Open Architectures and Network Programming* (April 2003), 55–64.

[3] AKARI. AKARI Architecture Project. `akari-project.nict.go.jp`.

[4] Andersen, David, Balakrishnan, Hari, Kaashoek, Frans, and Morris, Robert. Resilient Overlay Networks. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 131–145.

[5] Arthur, Colin M., Lehane, Andrew, and Harle, David. Keeping Order: Determining the Effect of TCP Packet Reordering. *Networking and Services, 2007. ICNS. Third International Conference on* (19-25 June 2007), 116–116.

[6] Awater, Geert A., and Kruys, Jan. Wireless ATM: An Overview. *Mob. Netw. Appl. 1*, 3 (1996), 235–243.

[7] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V., and Swallow, G. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209 (Proposed Standard), Dec. 2001. Updated by RFCs 3936, 4420, 4874, 5151, 5420.

[8] Ballani, Hitesh, Francis, Paul, and Zhang, Xinyang. A Study of Prefix Hijacking and Interception in the Internet. *SIGCOMM Comput. Commun. Rev. 37*, 4 (2007), 265–276.

[9] Banerjee, A., Drake, J., Lang, J.P., Turner, B., Kompella, K., and Rekhter, Y. Generalized Multiprotocol Label Switching: an Overview of Routing and Management Enhancements. *Communications Magazine, IEEE 39*, 1 (January 2001), 144–150.

[10] Banerjee, A., Drake, L., Lang, L., Turner, B., Awduche, D., Berger, L., Kompella, K., and Rekhter, Y. Generalized Multiprotocol Label Switching: an Overview of Signaling Enhancements and Recovery Techniques. *Communications Magazine, IEEE 39*, 7 (July 2001), 144–151.

[11] Barrénd, Séstien, Raiciu, Costin, Bonaventure, Olivier, and Handley, Mark. Experimenting with Multipath TCP. In *SIGCOMM 2010 Demo* (September 2010). http://conferences.sigcomm.org/sigcomm/2010/papers/sigcomm/p443.pdf.

[12] Bavier, Andy, Feamster, Nick, Huang, Mark, Peterson, Larry, and Rexford, Jennifer. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proceedings of the ACM SIGCOMM Conference* (September 2006).

[13] Bellman, Richard. On a Routing Problem. *Quarterly of Applied Mathematics 16*, 1 (1958), 87–90.

[14] Bennett, J.C.R., Partridge, C., and Shectman, N. Packet Reordering Is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking (TON) 7*, 6 (1999), 789–798.

[15] Bhattacharjee, Bobby, Calvert, Ken, Griffioen, Jim, Spring, Neil, and Sterbenz, James. Postmodern Internetwork Architecture. Tech. rep., 2006.

[16] Black, U. *X.25 and Related Protocols*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[17] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. An Architecture for Differentiated Service. RFC 2475 (Informational), Dec. 1998. Updated by RFC 3260.

[18] Blanton, E., Allman, M., Fall, K., and Wang, L. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), Apr. 2003.

[19] Blanton, Ethan, and Allman, Mark. On Making TCP More Robust to Packet Reordering. *SIGCOMM CCR 32*, 1 (2002), 20–30.

[20] Bohacek, Stephan, Hespanha, Joao P., Lee, Junsoo, Lim, Chansook, and Obraczka, Katia. A New TCP for Persistent Packet Reordering. *IEEE/ACM Trans. Netw. 14*, 2 (2006), 369–382.

[21] Braden, R., Clark, D., and Shenker, S. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), June 1994.

[22] CAIDA Macroscopic Topology Project Team. CAIDA Skitter AS Links Topology (collection). `http://imdc.datcat.org/collection/1-000W-X=CAIDA+skitter+AS+Links+Topology`.

[23] Campbell-Kelly, M., and Garcia-Swartz, D.D. The History of the Internet: The Missing Narratives.

[24] Castineyra, I., Chiappa, N., and Steenstrup, M. The Nimrod Routing Architecture. RFC 1992 (Informational), Aug. 1996.

[25] Cerf, V., Dalal, Y., and Sunshine, C. Specification of Internet Transmission Control Program. RFC 675, Dec. 1974.

[26] Chebrolu, Kameswari, Raman, Bhaskaran, and Rao, Ramesh R. A Network Layer Approach to Enable TCP Over Multiple Interfaces. *Wirel. Netw. 11*, 5 (2005), 637–650.

[27] Chun, Brent, Culler, David, Roscoe, Timothy, Bavier, Andy, Peterson, Larry, Wawrzoniak, Mike, and Bowman, Mic. PlanetLab: an Overlay Testbed for Broad-Coverage Services. *SIGCOMM Computer Communications Review 33*, 3 (2003), 3–12.

[28] Clark, D. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM '88: Symposium Proceedings on Communications Architectures and Protocols* (New York, NY, USA, 1988), ACM Press, pp. 106–114.

[29] Clarke, Ian, Sandberg, Oskar, Wiley, Brandon, and Hong, Theodore W. Freenet: a Distributed Anonymous Information Storage and Retrieval System. In *International Workshop on Designing Privacy Enhancing Technologies* (New York, NY, USA, 2001), Springer-Verlag New York, Inc., pp. 46–66.

[30] Clipsham, W.W., Glave, F.E., and Narraway, M.L. DATAPAC Network Overview. In *Proc. ICCC* (1976), pp. 131–136.

[31] Cohen, Bram. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems* (Berkeley, CA, USA, June 2003).

[32] Coltun, R., Ferguson, D., Moy, J., and Lindem, A. OSPF for IPv6. RFC 5340 (Proposed Standard), July 2008.

[33] Cormen, L. Introduction to Algorithms. *McGraw-Hill Company* (2000).

[34] Crowcroft, Jon, Hand, Steven, Mortier, Richard, Roscoe, Timothy, and Warfield, Andrew. QoS's Downfall: at the Bottom, or Not at All! In *RIPQoS '03: Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS* (New York, NY, USA, 2003), ACM Press, pp. 109–114.

[35] Dabek, Frank, Kaashoek, M. Frans, Karger, David, Morris, Robert, and Stoica, Ion. Wide-Area Cooperative Storage with CFS. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 202–215.

[36] Dasgupta, S., de Oliveira, J.C., and Vasseur, J.P. Path-Computation-Element-Based Architecture for Interdomain MPLS/GMPLS Traffic Engineering: Overview and Performance. *IEEE Network 21*, 4 (2007), 38–45.

[37] Deering, S., and Hinden, R. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), Dec. 1998. Updated by RFC 5095.

[38] Deloche, G. ARPA Network Functional Specifications. RFC 8, May 1969.

[39] Dimopoulos, Peter, Zeephongsekul, Panlop, and Tari, Zahir. Multipath Aware TCP (MATCP). *ISCC* (2006), 981–988.

[40] Duncanson, J. Inverse Multiplexing. *Communications Magazine, IEEE 32*, 4 (Apr 1994), 34–41.

[41] Eastlake 3rd, D., and Jones, P. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), Sept. 2001. Updated by RFC 4634.

[42] Estan, Cristian, Akella, Aditya, and Banerjee, Suman. Achieving Good End-to-End Service Using Bill-Pay. In *Hotnets-V* (New York, NY, USA, 2006), ACM.

[43] Fall, K., and Floyd, S. Simulation-Based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM CCR 26*, 3 (1996), 21.

[44] Farinacci, D., Li, T., Hanks, S., Meyer, D., and Traina, P. Generic Routing Encapsulation (GRE). RFC 2784 (Proposed Standard), Mar. 2000. Updated by RFC 2890.

[45] Fattah, H., and Leung, C. An Overview of Scheduling Algorithms in Wireless Multimedia Networks. *Wireless Communications, IEEE 9*, 5 (2002), 76 – 83.

[46] Feamster, N., Borkenhagen, J., and Rexford, J. Techniques for Interdomain Traffic Engineering. *Computer Communications Review 33*, 5 (2003).

[47] Feamster, Nick, Andersen, David G., Balakrishnan, Hari, and Kaashoek, Frans. Measuring the Effects of Internet Path Faults on Reactive Routing. In *ACM Sigmetrics - Performance 2003* (San Diego, CA, June 2003).

[48] Feamster, Nick, and Balakrishnan, Hari. Towards a Logic for Wide-Area Internet Routing. In *FDNA '03: Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture* (New York, NY, USA, 2003), ACM Press, pp. 289–300.

[49] Feamster, Nick, and Balakrishnan, Hari. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symp. on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005).

[50] Feamster, Nick, Balakrishnan, Hari, Rexford, Jennifer, Shaikh, Aman, and van der Merwe, Jacobus. The Case for Separating Routing from Routers. In *FDNA '04: Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture* (New York, NY, USA, 2004), ACM Press, pp. 5–12.

[51] FIRE. FIRE - Future Internet Research & Experimentation. `cordis.europa.eu/fp7/ict/fire`.

[52] Floyd, S. A Report on Recent Developments in TCP Congestion Control. *Comm. Magazine, IEEE 39*, 4 (Apr 2001), 84–90.

[53] Floyd, S., and Henderson, T. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), Apr. 1999. Obsoleted by RFC 3782.

[54] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[55] Ford, Jr., L R, and Fulkerson, D. R. *Flows in Networks*. Princeton University Press, 1962.

[56] Garcia-Lunes-Aceves, J. J. Loop-free Routing Using Diffusing Computations. *IEEE/ACM Trans. Netw. 1*, 1 (1993), 130–141.

[57] Gavras, Anastasius, Karila, Arto, Fdida, Serge, May, Martin, and Potts, Martin. Future Internet Research and Experimentation: the FIRE Initiative. *SIGCOMM Comput. Commun. Rev. 37*, 3 (2007), 89–92.

[58] GENI. Global Environment for Network Innnovations. `www.geni.net`.

[59] Godfrey, P., Ganichev, I., Shenker, S., and Stoica, I. Pathlet Routing. *ACM SIGCOMM Computer Communication Review 39*, 4 (2009), 111–122.

[60] Gray, James P. Services Provided to Users of SNA Networks. In *SIGCOMM '79: Proceedings of the Sixth Symposium on Data Communications* (New York, NY, USA, 1979), ACM Press, pp. 52–62.

[61] Greenberg, Albert, Hjalmtysson, Gisli, Maltz, David A., Myers, Andy, Rexford, Jennifer, Xie, Geoffrey, Yan, Hong, Zhan, Jibin, and Zhang, Hui. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Comput. Commun. Rev. 35*, 5 (2005), 41–54.

[62] Griffin, T.G., and Sobrinho, J.L. Metarouting. In *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2005), ACM, ACM Press, pp. 1–12.

[63] Griffin, Timothy G., and Wilfong, Gordon. An Analysis of BGP Convergence Properties. In *SIGCOMM '99: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 1999), ACM Press, pp. 277–288.

[64] Grossman, D. New Terminology and Clarifications for Diffserv. RFC 3260 (Informational), Apr. 2002.

[65] Gummadi, P. Krishna, Saroiu, Stefan, and Gribble, Steven D. A Measurement Study of Napster and Gnutella as Examples of Peer-to-Peer File Sharing Systems. *SIGCOMM Comput. Commun. Rev. 32*, 1 (2002), 82–82.

[66] Gustafsson, E. Karlsson, G. A Literature Survey on Traffic Dispersion. In *IEEE Network* (1997), IEEE, pp. 28–32.

[67] Hand, Steven, and Roscoe, Timothy. Mnemosyne: Peer-to-Peer Steganographic Storage. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 130–140.

[68] Herzog, S. RSVP Extensions for Policy Control. RFC 2750 (Proposed Standard), Jan. 2000.

[69] Hsieh, Hung-Yun, and Sivakumar, R. pTCP: An End-to-End Transport Layer Protocol for Striped Connections. *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on* (12-15 Nov. 2002), 24–33.

[70] Hsieh, Wen-Ning, and Gitman, Israel. Routing Strategies in Computer Networks. *IEEE Computer 1*, 1 (1984), 46–56.

[71] Huebsch, Ryan, Hellerstein, Joseph M., Lanham, Nick, Loo, Boon Thau, Shenker, Scott, and Stoica, Ion. Querying the Internet with PIER. In *VLDB* (2003), pp. 321–332.

[72] ISO. ISO Standard, IS 7498, Information Processing Systems – Open Systems Interconnection – Basic Reference Model. *ISO* (1985).

[73] Iyengar, Janardhan R., Amer, Paul D., and Stewart, Randall R. Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. *IEEE/ACM Trans. Netw. 14*, 5 (2006), 951–964.

[74] Jacobson, V., Smetters, D.K., Briggs, N.H., Plass, M.F., Stewart, P., Thornton, J.D., and Braynard, R.L. VoCCN: Voice-over Content-Centric Networks. In *Proceedings of the 2009 Workshop on Re-Architecting the Internet* (2009), ACM, pp. 1–6.

[75] Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., and Braynard, R.L. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies* (2009), ACM, pp. 1–12.

[76] Jayasumana, A., Piratla, N., Banka, T., Bare, A., and Whitner, R. Improved Packet Reordering Metrics. RFC 5236 (Informational), June 2008.

[77] Jones, Rick. Netperf. `www.netperf.org`.

[78] Joseph, Dilip, Kannan, Jayanthkumar, Kubota, Ayumu, Lakshminarayanan, Karthik, Stoica, Ion, and Wehrle, Klaus. OCALA: An Architecture for Supporting Legacy Applications over Overlays. Tech. Rep. UCB/CSD-05-1397, EECS Department, University of California, Berkeley, 2005.

[79] Kaat, M. Overview of 1999 IAB Network Layer Workshop. RFC 2956 (Informational), Oct. 2000.

[80] Kalmanek, Charles. A Retrospective View of ATM. *SIGCOMM Comput. Commun. Rev. 32*, 5 (2002), 13–19.

[81] Kaur, H. Tahilramani, Kalyanaraman, S., Weiss, A., Kanwar, S., and Gandhi, A. BANANAS: an Evolutionary Framework for Explicit and Multipath Routing in the Internet. In *FDNA '03: Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture* (New York, NY, USA, 2003), ACM Press, pp. 277–288.

[82] Keshav, S. *An Engineering Approach to Computer Networking*. Addison-Wesley, 1997.

[83] Kim, Sung-Ho, and Ko, Young-Bae. Wireless Bonding for Maximizing Throughput in Multi-Radio Mesh Networks. *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on* (19-23 March 2007), 570–576.

[84] Kohler, Eddie, Morris, Robert, Chen, Benjie, Jannotti, John, and Kaashoek, M. Frans. The Click Modular Router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[85] Kompella, K., and Lang, J. Procedures for Modifying the Resource reSerVation Protocol (RSVP). RFC 3936 (Best Current Practice), Oct. 2004.

[86] Koponen, T., Chawla, M., Chun, B.G., Ermolinskiy, A., Kim, K.H., Shenker, S., and Stoica, I. A Data-Oriented (and Beyond) Network Architecture. *ACM SIGCOMM Computer Communication Review 37*, 4 (2007), 192.

[87] Koren, Debby. Are We Ready for IPv6? Is IPv6 Ready for Us? *Int. J. Netw. Manag. 15*, 1 (2005), 61–66.

[88] Kubiatowicz, John, Bindel, David, Chen, Yan, Czerwinski, Steven, Eaton, Patrick, Geels, Dennis, Gummadi, Ramakrishna, Rhea, Sean, Weatherspoon, Hakim, Wells, Chris, and Zhao, Ben. OceanStore: an Architecture for Global-Scale Persistent Storage. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ACM Press, pp. 190–201.

[89] Labovitz, Craig, Ahuja, Abha, Bose, Abhijit, and Jahanian, Farnam. Delayed Internet Routing Convergence. *IEEE/ACM Trans. Netw. 9*, 3 (2001), 293–306.

[90] Labovitz, Craig, Ahuja, Abha, and Jahanian, Farnam. Experimental Study of Internet Stability and Wide-Area Network Failures. In *Proc. International Symposium on Fault-Tolerant Computing* (1999).

[91] Lane, John Russell, and Nakao, Akihiro. SORA: A Shared Overlay Routing Architecture. In *ROADS Warsaw 2007: Proceedings of the 2007 Workshop on Real Overlay and Distributed Systems, Warsaw* (New York, NY, USA, 2007), ACM Press.

[92] Lane, John Russell, and Nakao, Akihiro. Best-Effort Network Layer Packet Reordering in Support of Multipath Overlay Packet Dispersion. *Globecom 2008, Proceedings of* (December 2008).

[93] Lane, John Russell, and Nakao, Akihiro. Path Brokering for End-Host Path Selection: Toward a Path-Centric Billing Method for a Multipath Internet. *Proceedings of ReArch 2008* (Dec. 2008).

[94] Lane, John Russell, and Nakao, Akihiro. On Best-Effort Packet Reordering for Mitigating the Effects of Out-of-Order Delivery on Unmodified TCP. *IEICE Transactions on Communications 93*, 5 (2010), 1095–1103.

[95] Langar, Rami, Grand, Gwendal Le, and Tohme, Samir. Fast Handoff Process in Micro-Mobile MPLS Protocol for Micro-Mobility Management in Next Generation Networks. In *WONS '05: Proceedings of the Second Annual Conference on Wireless On-demand Network Systems and Services (WONS'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 252–257.

[96] Lee, Sung-Ju, Banerjee, Sujata, Sharma, Puneet, Yalagandula, Praveen, and Basu, Sujoy. Bandwidth-Aware Routing in Overlay Networks. *INFOCOM 2008* (2008).

[97] Lehman, T., Sobieski, J., and Jabbari, B. DRAGON: a Framework for Service Provisioning in Heterogeneous Grid Networks. *Communications Magazine, IEEE 44*, 3 (March 2006), 84–90.

[98] Li, Zhi, and Mohapatra, P. QRON: QoS-Aware Routing in Overlay Networks. *IEEE Journal on Selected Areas in Communications 22*, 1 (January 2004), 29–40.

[99] Liao, Jianxin, Wang, Jingyu, and Zhu, Xiaomin. cmpSCTP: an Extension of SCTP to Support Concurrent Multi-Path Transfer. In *ICC* (2008), pp. 5762–5766.

[100] Linux Channel Bonding. `sourceforge.net/projects/bonding`.

[101] Liu, Hang, and Raychaudhuri, Dipankar. Label Switched Multi-path Forwarding in Wireless Ad-Hoc Networks. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 248–252.

[102] Mahadevan, Priya, Hubble, Calvin, Krioukov, Dmitri, Huffaker, Bradley, and Vahdat, Amin. Orbis: Rescaling Degree Correlations to Generate Annotated Internet Topologies. In *SIGCOMM '07: Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2007), ACM, pp. 325–336.

[103] Mahajan, Ratul, Wetherall, David, and Anderson, Tom. Understanding BGP Misconfiguration. In *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2002), ACM Press, pp. 3–16.

[104] Malkin, G. RIP Version 2. RFC 2453 (Standard), Nov. 1998. Updated by RFC 4822.

[105] Marques, P., and Dupont, F. Use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain Routing. RFC 2545 (Proposed Standard), Mar. 1999.

[106] Maxemchuk, N. F. Dispersity Routing. *Proc. IEEE Int'l Comm. Conf. (ICC '75) 1*, 1 (June 1975), 10–13.

[107] Maxemchuk, N.F. Dispersity Routing on ATM Networks. *INFOCOM '93. Proceedings of the Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies 1* (1993), 347–357.

[108] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[109] Mills, D.L. Exterior Gateway Protocol Formal Specification. RFC 904 (Historic), Apr. 1984.

[110] Montenegro, G. Reverse Tunneling for Mobile IP, revised. RFC 3024 (Proposed Standard), Jan. 2001.

[111] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and Perser, J. Packet Reordering Metrics. RFC 4737 (Proposed Standard), Nov. 2006.

[112] Moy, J. OSPF Version 2. RFC 2328 (Standard), Apr. 1998.

[113] Murphy, S. BGP Security Vulnerabilities Analysis. RFC 4272 (Informational), Jan. 2006.

[114] MySQL, AB. MySQL, 2004.

[115] Nakao, A., Ozaki, R., and Nishida, Y. CoreLab: an Emerging Network Testbed Employing Hosted Virtual Machine Monitor. In *Proceedings of the 2008 ACM CoNEXT Conference* (2008), ACM New York, NY, USA.

[116] Narten, T., Draves, R., and Krishnan, S. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard), Sept. 2007.

[117] Odlyzko, Andrew. The Delusions of Net Neutrality. *36th Telecommunications Policy Research Conf. Proceedings* (Sept. 2008).

[118] OneLab. OneLab - Testbeds for Future Internet Research & Experimentation. `www.onelab.eu`.

[119] Oran, D. OSI IS-IS Intra-domain Routing Protocol. RFC 1142 (Informational), Feb. 1990.

[120] Owezarski, Philippe. Does IPv6 Improve the Scalability of the Internet? In *IDMS/PROMS 2002: Proceedings of the Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems* (London, UK, 2002), Springer-Verlag, pp. 130–140.

[121] Padhye, Jitendra, Firoiu, Victor, Towsley, Don, and Kurose, Jim. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, architectures, and Protocols for Computer Communication* (New York, NY, USA, 1998), ACM, pp. 303–314.

[122] Pelsser, C., and Bonaventure, O. RSVP-TE Extensions for Interdomain LSPs, October 2002. Work in progress, draft-pelsser-rsvp-te-interdomain-lsp-00.txt.

[123] Piratla, N.M., and Jayasumana, A.P. Metrics for Packet Reordering—a Comparative Analysis. *International Journal of Communication Systems 21*, 1 (2008), 99–113.

[124] Postel, J. User Datagram Protocol. RFC 768 (Standard), Aug. 1980.

[125] Postel, J. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[126] Pouzin, L., and AndrGe, E. *Cyclades Computer Network: Toward Layered Network Architectures*. Elsevier Science Inc. New York, NY, USA, 1982.

[127] Ratnasamy, Sylvia, Francis, Paul, Handley, Mark, Karp, Richard, and Schenker, Scott. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), ACM Press, pp. 161–172.

[128] Ratnasamy, Sylvia, Handley, Mark, Karp, Richard M., and Shenker, Scott. Application-Level Multicast Using Content-Addressable Networks. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication* (London, UK, 2001), Springer-Verlag, pp. 14–29.

[129] Rekhter, J. EGP and Policy Based Routing in the New NSFNET Backbone. RFC 1092, Feb. 1989.

[130] Rekhter, Y., and Li, T. A Border Gateway Protocol 4 (BGP-4). RFC 1654 (Proposed Standard), July 1994. Obsoleted by RFC 1771.

[131] Rhee, I., and Xu, L. CUBIC: A New TCP-Friendly High-Speed TCP Variant. In *Proc. PFLDnet* (2005), vol. 2005.

[132] Roberts, L.G. The TELENET Network; The Benefits of Public Packet Service. In *Proceedings of the 3rd International Conference on Computer Communications* (1976), pp. 239–245.

[133] Rosen, E., Viswanathan, A., and Callon, R. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), Jan. 2001.

[134] Rosen, E.C. Exterior Gateway Protocol (EGP). RFC 827, Oct. 1982. Updated by RFC 904.

[135] Rowstron, Antony, and Druschel, Peter. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 188–201.

[136] Rowstron, Antony I. T., and Druschel, Peter. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg* (London, UK, 2001), Springer-Verlag, pp. 329–350.

[137] Rowstron, Antony I. T., Kermarrec, Anne-Marie, Castro, Miguel, and Druschel, Peter. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication* (London, UK, 2001), Springer-Verlag, pp. 30–43.

[138] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst. 2*, 4 (1984), 277–288.

[139] Seamonson, L., and Rosen, E.C. "STUB" Exterior Gateway Protocol. RFC 888, Jan. 1984. Updated by RFC 904.

[140] Sethom, Kaouthar, Afifi, Hossam, and Pujolle, Guy. Wireless MPLS: a New Layer 2.5 Micro-Mobility Scheme. In *MobiWac '04: Proceedings of the Second International Workshop on Mobility Management & Wireless Access Protocols* (New York, NY, USA, 2004), ACM Press, pp. 64–71.

[141] Sklower, K., Lloyd, B., McGregor, G., Carr, D., and Coradetti, T. The PPP Multilink Protocol (MP). RFC 1990 (Draft Standard), Aug. 1996.

[142] Skype. Skype home page. `www.skype.com`.

[143] Smallwood, J.E. An Introduction to ATM. *Computing & Control Engineering Journal 9*, 5 (October 1998), 233–245.

[144] Smith, P. *Frame Relay: Principles and Applications*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1993.

[145] Snoeren, Alex C., and Raghavan, Barath. Decoupling Policy from Mechanism in Internet Routing. *SIGCOMM Comput. Commun. Rev. 34*, 1 (2004), 81–86.

[146] Sobrinho, J.L. Network Routing with Path Vector Protocols: Theory and Applications. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2003), ACM Press, pp. 49–60.

[147] Song, Fei, Zhang, Hongke, Zhang, Sidong, Ramos, Fernando M. V., and Crowcroft, Jon. Relative Delay Estimator for SCTP-Based Concurrent Multipath Transfer. In *GLOBECOM* (2010), pp. 1–6.

[148] Stewart, R. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007.

[149] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and Paxson, V. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), Oct. 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[150] Stoica, Ion, Adkins, Daniel, Zhuang, Shelley, Shenker, Scott, and Surana, Sonesh. Internet Indirection Infrastructure. *IEEE/ACM Trans. Netw. 12*, 2 (2004), 205–218.

[151] Stoica, Ion, Morris, Robert, Karger, David, Kaashoek, M. Frans, and Balakrishnan, Hari. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), ACM Press, pp. 149–160.

[152] Sunshine, Carl A. Source Routing in Computer Networks. *SIGCOMM Comput. Commun. Rev. 7*, 1 (1977), 29–33.

[153] Teixeira, Renata, Shaikh, Aman, Griffin, Tim, and Rexford, Jennifer. Dynamics of Hot-Potato Routing in IP Networks. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2004), ACM Press, pp. 307–319.

[154] The University of Oregon. Routeviews Project. `www.routeviews.org`.

[155] Touch, J. Dynamic Internet Overlay Deployment and Management Using the X-Bone. In *ICNP '00: Proceedings of the 2000 International Conference on Network Protocols* (Washington, DC, USA, 2000), IEEE Computer Society, p. 59.

[156] Touch, Joseph D., Wang, Yu-Shun, Pingali, Venkata, Eggert, Lars, Zhou, Runfang, and Finn, Gregory G. A Global X-Bone for Network Experiments. In *TRIDENTCOM '05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities (TRIDENTCOM'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 194–203.

[157] Tymes, LR. Routing and Flow Control in TYMNET. *Communications, IEEE Transactions on 29*, 4 (1981), 392–398.

[158] Valancius, Vytautas, Feamster, Nick, Johari, Ramesh, and Vazirani, Vijay. MINT: A Market for INternet Transit. *Proceedings of ReArch 2008* (Dec. 2008).

[159] Wan, T., and van Oorschot, P.C. Analysis of BGP Prefix Origins During Google's May 2005 Outage. *Proc. of Security in Systems and Networks* (2006).

[160] Wecker, S. DNA: the Digital Network Architecture. *Communications, IEEE Transactions on [Legacy, Pre-1988] 28*, 4 (1980), 510–526.

[161] Weiser, Mark. Whatever Happened to the Next-Generation Internet? *Commun. ACM 44*, 9 (2001), 61–69.

[162] White, Brian, Lepreau, Jay, Stoller, Leigh, Ricci, Robert, Guruprasad, Shashi, Newbold, Mac, Hibler, Mike, Barb, Chad, and Joglekar, Abhijeet. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[163] Winer, David. XML-RPC Specification. `www.xmlrpc.com/spec`.

[164] Xie, Kaiduan, Wong, V.W.S., and Leung, V.C.M. Support of Micro-Mobility in MPLS-Based Wireless Access Networks. *IEEE Wireless Communications and Networking Conference (WCNC 2003) 2* (March 2003), 16–20.

[165] XML-RPC-C. XML-RPC-C Library Website. `xmlrpc-c.sourceforge.net`.

[166] Xu, Wen, and Rexford, Jennifer. MIRO: Multi-path Interdomain Routing. In *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2006), ACM, pp. 171–182.

[167] Yan, Hong, Maltz, David A., Ng, T. S. Eugene, Gogineni, Hemant, Zhang, Hui, and Cai, Zheng. Tesseract: A 4D Network Control Plane. In *NSDI* (2007), http://www.usenix.org/events/nsdi07/tech/yan.html.

[168] Yang, Xi, Lehman, Tom, Tracy, Chris, Sobieski, Jerry, Gong, Shujia, Torab, Payam, and Jabbari, Bijan. Policy-Based Resource Management and Service Provisioning in GMPLS Networks. *The First IEEE Workshop on Adaptive Policy-based Management in Network Management and Control, Proceedings of 1*, 1 (April 2006).

[169] Yang, Xiaowei. NIRA: a New Internet Routing Architecture. In *FDNA '03: Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture* (New York, NY, USA, 2003), ACM Press, pp. 301–312.

[170] Yang, Xiaowei. An Internet Architecture for User-Controlled Routes. *NSF FIND Proposal* (2007).

[171] Yang, Xiaowei, Clark, David, and Berger, Arthur. NIRA: a New Routing Architecture. *IEEE/ACM Transactions on Networking* (2007).

[172] Yang, Xiaowei, and Wetherall, David. Source Selectable Path Diversity via Routing Deflections. In *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2006), ACM, pp. 159–170.

[173] Zhang, Lixia, Deering, S., Estrin, D., Shenker, S., and Zappala, D. RSVP: a New Resource Reservation Protocol. *Communications Magazine, IEEE 40*, 5 (May 2002), 116–127.

[174] Zhang, Ming, Karp, Brad, Floyd, Sally, and Peterson, Larry. RR-TCP: A Reordering-Robust TCP with DSACK. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols* (Washington, DC, USA, 2003), p. 95.

[175] Zhang, Ming, Lai, Junwen, Krishnamurthy, Arvind, Peterson, Larry, and Wang, Randolph. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. In *USENIX 2004 Conference Proceedings* (2004), pp. 99–112.

[176] Zhao, B.Y., Huang, Ling, Stribling, J., Rhea, S.C., Joseph, A.D., and Kubiatowicz, J.D. Tapestry: a Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (January 2004), 41–53.

[177] Zhuang, Shelley Q., Zhao, Ben Y., Joseph, Anthony D., Katz, Randy H., and Kubiatowicz, John D. Bayeux: an Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination. In *NOSSDAV '01: Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (New York, NY, USA, 2001), ACM Press, pp. 11–20.

PUBLICATIONS RELATED TO SORA

1. *SORA: A Shared Overlay Routing Architecture – Enabling Multipath Routing on PlanetLab and Beyond*, ACM ROADS 2007, Warsaw, Poland
2. *End-Host Path Monitoring and Selection Supporting Packet Dispersion on Multipath Overlay Networks*, CFI 2008, Seoul, Korea
3. *Best Effort Network Layer Packet Reordering in Support of Multipath Overlay Packet Dispersion*, IEEE GlobeCom 2008, New Orleans, USA
4. *Path Brokering for End-Host Path Selection: Toward a Path-Centric Billing Model for a Multipath Internet*, ACM ReArch 2008, Madrid, Spain
5. *On Best-Effort Packet Reordering for Mitigating the Effects of Out-Of-Order Delivery on Unmodified TCP*, IEICE Transactions on Communications, Special Issue on Technology and Architecture for Sustainable Growth of the Internet, Vol. 93, No. 5, May 2010

Appendices

APPENDIX A

FUNDAMENTAL SORA CLICK ELEMENTS

A.1 Elements

A.1.1 Packet Header Processing Elements

SoraPacketHToN()
Agnostic. Converts the SORA packet header pointed to by the SORA packet annotation pointer from host to network byte order.

SoraPacketNToH()
Agnostic. Converts the SORA packet header pointed to by the SORA packet annotation pointer from network to host byte order.

SoraSetChecksum()
Agnostic. Computes the checksum of the SORA packet header pointed to by the packet annotation pointer and sets the computed value therein.

SoraCheckHeader()
Agnostic. Performs basic verification of the SORA packet header sizes and computes the packet checksum and verifies that it matches the checksum in the packet.

SoraUDPIPMarkHeader()
Agnostic. Given a UDP/IP packet as input, creates pointers to the start of the SORA packet header struct and SORA payload in the Click packet annotation.

SoraStripHeader()
Agnostic. Sets the Click packet start pointer to point to the SORA payload (i.e., the encapsulated packet).

SoraUnstripHeader()
Agnostic. If a pointer address to the SORA packet header remains in the Click packet annotation, sets the Click packet start pointer to point to the given pointer address.

SoraIncrementHop()
Agnostic. Increments the current link identifier in the SORA path option in the SORA packet header pointed to by the SORA packet annotation pointer.

A.1.2 Packet Option Handling Elements

SoraCreatePacketClass()
Agnostic. Creates a SoraPacket object and sets a pointer to it in the Click packet annotation.

SoraCreatePacketStruct()
Agnostic. Obtains the SoraPacket object from the Click packet annotation and uses it to create a header struct (`sora_packet_t`) and sets a pointer to it in the Click packet annotation; deallocates the SoraPacket object.

SoraSetConduitIDInfoOption()
Agnostic. Adds the conduit ID information option to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

SoraSetDestinationOption()
Agnostic. Adds the destination option to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

SoraSetFlagsOption()
Agnostic. Adds the packet flags to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

SoraSetNetworkOption()
Agnostic. Adds the network option to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

SoraSetPathOption()
Agnostic. Adds the path option to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

SoraSetSourceOption()
Agnostic. Adds the source option to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

SoraSetSequenceNumberOption()
Agnostic. Adds the sequence number option to the SoraPacket object pointed to in the Click packet annotation using information from the conduit.

A.1.3 Conduit Handling Elements

SoraCreateConduit(`OVERWRITE` *boolean*)
Agnostic. Creates a new path conduit endpoint object and sets its pointer address in the Click annotation; if `OVERWRITE` is true, creates a conduit even if a non-zero value exists for the conduit pointer address value in the Click annotation; if `OVERWRITE` is false, refuses to create a new conduit.

SoraSetConduitValues(`ID_ONLY` *boolean*, `IS_INBOUND` *boolean*)
Agnostic. Sets initial default values for the path conduit endpoint instance pointed to by the conduit endpoint pointer address value stored in the Click annotation, including source and destination conduit endpoint identifiers; if `ID_ONLY` is true, only sets the conduit endpoint identifiers; if `IS_INBOUND` is true, sets the conduit values based on an initial inbound packet (i.e., with source and destination swapped).

SoraConduitCacheFind(`USE_IP` *boolean*, `USE_PORT` *boolean*, `IS_INBOUND` *boolean*)
Port 0: Agnostic; Port 1: Push. Looks up the conduit endpoint instance in the conduit cache for the given packet; if it finds it, it sets the conduit endpoint pointer address value in the Click annotation and outputs the packet on port zero; if it does not find it, it sends the packet out port one; if both `USE_IP` and `USE_PORT` are set, the lookup key is the source and destination IP addresses, ports and the protocol; if `USE_IP` is set and `USE_PORT` is not, the lookup key is only the source and destination IP addresses; if `USE_IP` is not set, an inbound packet is assumed and the lookup key is the conduit endpoint identifiers in the packet; if `IS_INBOUND` is true, an inbound packet is assumed for purposes of creating the lookup key based on IP addresses (i.e., the source and destination addresses are swapped).

SoraConduitCacheSet(`USE_IP` *boolean*, `USE_PORT` *boolean*, `IS_INBOUND` *boolean*, `OVERWRITE` *boolean*)
Agnostic. Inserts a pointer to the conduit endpoint instance referred to by the conduit endpoint pointer address value in the Click annotation into the path conduit endpoint cache; if both `USE_IP` and `USE_PORT` are set, the lookup key is the source and destination IP addresses, ports and the protocol; if `USE_IP` is set and `USE_PORT` is not, the lookup key is only the source and destination IP addresses; if `USE_IP` is not set, an inbound packet is assumed and the lookup key is the conduit endpoint identifiers in the packet; if `IS_INBOUND` is true, an inbound packet is assumed for purposes of creating the lookup key based on IP addresses (i.e., the source and destination addresses are swapped); if `OVERWRITE` is true, stores the conduit even if a mapping already exists for the given key in the conduit cache.

SoraFilterUnsetConduits()
Input 0: Agnostic; Output 0: Agnostic, 1: Push. Verifies that the conduit referred to by the conduit endpoint pointer address value in the Click annotation has minimum requirements to begin sending packets with: a network identifier, a source entity identifier, a destination entity identifier and at least one path. Packets meeting these requirements are output on port zero; unset packets are output on port 1; this element is generally used in the lookup pipeline.

A.1.4 Control Plane Service Lookup

SoraSetPacketState(`STATE` *string*)
Agnostic. Sets the packet ingress lookup state to the given argument, which should be one of `LIVE`, `LOOKUP` or `PASSTHROUGH`.

SoraLookupLinks(`DIRECTION` *string*, `LOOKUP_TARGET` *string*, `STORAGE_TARGET` *string*, `PATH_SOURCE` *string*, `URL` *string*, `USERNAME` *string*, `PASSWORD` *string*)

Input 1: Agnostic, Outputs 1-3: Push. Performs link lookup on a state database. The `DIRECTION` argument is one of `INBOUND` or `OUTBOUND` and determines the type of links to search for. The `LOOKUP_TARGET` is one of `SOURCE`, `DESTINATION`, `FIRST_HOP` or `NEXT_HOP`. If the lookup target is one of `SOURCE` or `DESTINATION`, lookup is performed on all outbound (source) or inbound (destination) links, otherwise, a hop-based lookup is performed: the path is consulted and the first hop or next hop link is looked up, depending upon the argument given. The `STORAGE_TARGET` argument should be one of `CONDUIT` or `ANNO` and determines where retrieved links are stored: in the conduit or in the annotation. The `PATH_SOURCE` argument should be one of `CONDUIT` or `ANNO` and determines where the path should be obtained in the case of a hop-based lookup: the conduit or packet annotation. The `URL` argument should contain the base URL of an XML-RPC query server; it overrides the system default. The `USERNAME` and `PASSWORD` arguments specify the username and password required to access the query server.

SoraLookupPaths(`NUM_PATHS` *integer*, `WINDOW_LENGTH` *integer*, `NUM_WINDOWS` *integer*, `URL` *string*, `USERNAME` *string*, `PASSWORD` *string*)

Input 1: Agnostic, Outputs 1-3: Push. Performs paths lookup using the source and destination entity identifiers stored in the conduit. The `WINDOW_LENGTH` and `NUM_WINDOWS` arguments control the size and length of the statistics windows maintained by the `SoraPath` classes created by the element upon successful path query. The `URL` argument should contain the base URL of an XML-RPC query server; it overrides the system default.

SoraDiscardFailedLookups(`VERBOSE` *boolean*)

Agnostic. Examines the ingress lookup state and discards all non-live packets. Outputs a message on discard if the `VERBOSE` option is set.

A.1.5 Encapsulation

SoraUDPIPEncap()

Input 1: Agnostic, Output 1: Agnostic, 2: Push. Encapsulates an input packet in a UDP/IP header by adding the UDP/IP header just above it. Reads a UDP/IP hop link pointer from the packet annotation and writes the UDP information therein to the packet header. For direct use with Click interfaces. For using sockets (e.g., on PlanetLab), see `SoraUDPIPSocket`.

SoraUDPIPSocket(`SOURCE_PORT` *IP port*)
Input 1: Agnostic, Output 0. Encapsulates an input packet in a UDP/IP header by adding the UDP/IP header just above it. Reads a UDP/IP next hop link pointer from the packet annotation and sends the packet to the destination IP and port noted therein using the given source port which defaults to the SORA default outbound port (48484). For use with sockets, therefore this also deallocates the packet and has no output port. For using Click interfaces, see `SoraUDPIPEncap`.

A.1.6 Path Scheduling

SoraSetPathScheduler(`TYPE` *string*)
Agnostic. Creates a scheduler for the conduit referred to in the given packet's annotation. The type of the scheduler is determined by the `TYPE` option and is one of `ROUND_-ROBIN` or `RANDOM`.

SoraSchedulePath(`PROBES` *boolean*)
Input 1: Agnostic, Output 1: Push. Schedules a packet using the scheduler within the conduit endpoint interface pointed to by the packet's pointer annotation. Whether the scheduler schedules a live or a probe packet is determined by the `PROBES` option (when it is true, probe packets will be scheduled, otherwise live packets will be scheduled).

A.1.7 Packet Filtering

SoraClassifier(`SRC` *address*, `DST` *address*, `SRC_ANNO` *address*, `DST_ANNO` *address*, `FIRST_HOP` *integer*, `NEXT_HOP` *integer*, `-`)
Input 1: Agnostic, Outputs: Push. Implements basic classifier functionality similar to Click's own `Classifier` and `IPClassifier` elements. Each argument specifies a match target (the given argument) and an output port (the position of the argument in the argument list); packets with a field matching the given argument are sent out its corresponding output port. The special argument "-" matches all packets. Note that `SRC` and `DST` match against the packet source and destination identifiers; `SRC_ANNO` and `DST_ANNO` match against the annotation pointer source and destination identifiers.

SoraLocalTargetFilter(`USE_CONDUIT` *boolean*, `ADDRESSES` *IP address list*, `URL` *string*, `USERNAME` *string*, `PASSWORD` *string*)
Input 1: Agnostic, Output 1: Agnostic, 2: Push. Attempts to determine the local interface IP addresses; uses those from the `ADDRESSES` option, if available. The element also tries to load the address mapping cache using those addresses. When input a packet, it examines the destination address pointed to either in the conduit or the packet (depending upon the configuration variable `USE_CONDUIT`) as well as the path and outputs a packet on its second port if the packet is either destined for one of the local addresses or if this is the last hop, and on its first port otherwise.

A.1.8 Packet Services

SoraReorderPackets(`MAX_HOLD` *integer*, `STRICT` *boolean*)

Input: Agnostic, Outputs: Push. Accepts a packet on its input port and, if it has a sequence number available, locates the conduit endpoint instance referenced in the packet annotation and uses it to locate the reorderer instance for the given conduit endpoint. If a reorderer instance is found, it updates the reorderer with the packet. If a reorderer instance is not found, it creates one with the given arguments for `max_hold` and strict reordering.

APPENDIX B

SORA CONTROL PLANE PROGRAMS

Name	Purpose
`sora_compute_paths`	Computes paths based on input links
`sora_cpp`	Processes a SORA Click router input file
`sora_db`	Interfaces with the SORA link database
`sora_entity`	Interfaces with SORA entity-based services
`sora_entityd`	Handles services for entities
`sora_gencert`	Generates an RSA certificate for use with XML-RPC
`sora_genkey`	Generates an RSA key for use with link signatures
`sora_get_rtts`	Obtains RTTs from a set of hosts
`sora_instantiate_links`	Instantiates a set of links (in parallel)
`sora_mk_pl_links`	Generates a full mesh link set for PlanetLab
`sora_pcs`	Interfaces with the path computation service
`sora_pcsd`	Handles path computation service
`sora_rtts_to_links`	Converts a set of RTT values into links
`sora_sign_links`	Signs links in a link file
`sora_sqldbd`	Handles link database and link request services
`sora_verify_links`	Verifies links against an link signing RSA key
`sora_wrap`	Performs libc wrapping for automated ingress

Table B.1: SORA Control Plane Program Summary