

A Narrow Waist for Multipath Routing

Murtaza Motiwala*, Mukarram bin Tariq*, Bilal Anwer*, David Andersen†, Nick Feamster*

*School of Computer Science, Georgia Tech †Carnegie Mellon University

ABSTRACT

Many applications can use multipath routing to improve reliability or throughput, and many multipath routing protocols exist. Despite this diversity of mechanisms and applications, no common interface exists to allow an application to select these paths. This paper presents a design for such a common interface, called *path bits*. Path bits are a small string of opaque, semantic-free bits in a packet header; these bits have a simple property: changing a packet’s path bits should, with high probability, result in the packet taking a different path to the destination. This paper presents the design of path bits and demonstrates that they are simple enough to be easily implemented in both hardware and software and expressive enough to support a variety of applications that can benefit from multipath routing. We present both hardware and software implementations of multipath routing protocols that implement the path bits abstraction, as well as implementations of applications that can use this abstraction with only small modifications.

1. Introduction

Networked applications in the wide-area, enterprise, and data center can all benefit from network protocols that allow traffic to be sent over multiple paths en route to a destination. Such *multipath routing* can improve performance for networked applications by improving throughput or by exposing redundant paths in the face of link and node failures in the network. Applications that can benefit from multipath routing range from real-time applications such as network voice and video to bulk-transfer applications. Just as the many applications for multipath routing have many goals—throughput, low latency, or resilience, to name a few—so are there many ways of implementing multipath routing, each of which may be more or less beneficial to different classes of applications [11, 20, 31, 33].

Both real-world adoption and research into multipath routing is hampered by the lack of a common interface for accessing multiple paths in the network. Each implementation of multipath routing has typically come with its own, unique way of allowing applications to specify a path to use. The unfortunate consequence of this is that there is neither a standard multipath interface, nor a set of applications ready to make use of any multipath mechanisms that could become available. The lack of a flexible, widely-applicable interface thus makes it difficult to

create usable multipath implementations and applications, and also raises high barriers for researchers attempting to compare and contrast the benefits of different approaches.

The premise of this paper is that many multipath implementations can be realized—or nearly realized—by exposing a properly constructed common interface to applications. This common interface provides a *narrow waist for multipath routing*: Below the narrow waist, multipath routing schemes can evolve, and network service providers can replace one multipath routing scheme for another. Above the waist, any application can gain access to multipath routing capabilities, as long as its socket layer conforms to the interface specified by this narrow waist.

To be successful, this narrow waist must meet four requirements: It must be *general* enough to support a wide set of applications, *powerful* enough to take advantage of the capabilities provided by multipath mechanisms, and *easy to adopt* in applications without major rewriting. Finally, it must admit *efficient implementation* in networking hardware and software.

We present the design and implementation of such a narrow waist based upon two high-level design choices. At the network layer, path choice is expressed by an opaque *path bits* interface that provides a simple but powerful semantics: two packets with the same destination address, but with different path bits, will, with high probability, take different paths to the destination. At the host layer, we provide an interface for applications to express their intended use of multipath routing so that the networking stack can automatically set the path bits as appropriate.

Path bits represent an explicit choice about the amount of control that an end system can have over the paths that its traffic takes en route to a destination. Path bits do not explicitly select hops along the path, but instead correspond to *some* path. Making path bits opaque provides an interface to applications that both divorces the interface from any specific implementation of multipath routing and balances control over traffic forwarding between end systems and network operators.

The host and network stack expose an interface that allows the application either to explicitly set path bits in packets or to specify generally when the network stack should change path bits automatically. To enable an end system to request that the network layer automatically set bits, we expose API calls to allow the application to specify conditions under which it would request a new path; this same interface also allows the end system to

request multiple paths. A monitoring daemon then monitors these metrics and sets *path bits* on behalf of applications at higher layers. Like other aspects of the system, we have designed the interface between the monitoring daemon and both the applications and path selection to be general enough to support a variety of multipath monitoring strategies.

This paper makes the following contributions:

- We develop a general interface for multipath routing that supports both a variety of applications with minimal modifications to the applications themselves and many implementations. We demonstrate the generality of our interface by showing how many different multipath implementations can map to it.
- We implement an interface to allow end hosts to send traffic along multiple paths en route to a destination. As part of this framework, we develop a general end-host monitoring framework so that end-host applications can specify path selection criteria (*i.e.*, number of paths, performance characteristics of paths) and rely on the network to automatically adjust paths when they do not meet the desired criteria.
- We demonstrate the generality of the interface by developing implementations of three multipath routing mechanisms on four hardware and software router implementations (*i.e.*, Click [16], NetFPGA [1], OpenFlow [21], Intel IXP). Table 1 summarizes implementations, which are publicly available on our project Web site [2].
- We evaluate these implementations to show the benefits of this design for failure recovery at end hosts and for bulk transfer applications. We demonstrate the design’s flexibility by running a common monitoring algorithm for different multi-path schemes.

The rest of the paper is organized as follows. Section 2 surveys existing multipath routing protocols and technologies. Section 3 describes the design goals for a multipath narrow waist, and Section 4 describes the design of the narrow waist itself, including the design of the functions that reside at the end system and at the network device. Section 5 describes our implementation of the host component of this narrow waist, and Section 6 describes software and hardware implementations of path bits on network devices. Section 7 builds and demonstrates two types of applications with path bits: a failure recovery application, and a bulk transfer application. Section 8 concludes with a summary and possible avenues for future work using these implementations.

2. The Case for a Narrow Waist

We describe several multipath routing proposals and motivate the need for a consistent, narrow interface between applications and multipath routing implementations.

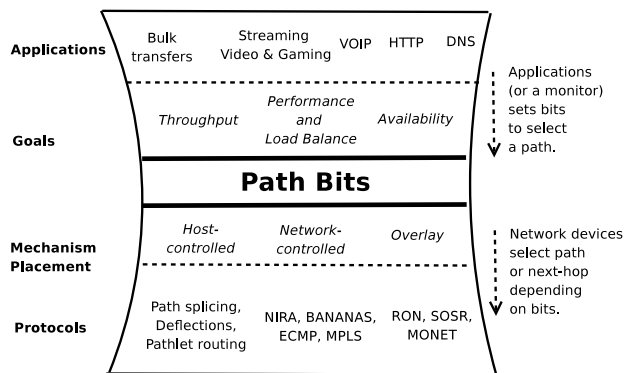


Figure 1: Hourglass model.

2.1 Different Applications and Goals

Applications differ in both their requirements and their ability to take advantage of multiple paths. This list of application types is not meant to be definitive or exhaustive, but merely to illustrate that many applications could credibly benefit from using different criteria for the ways they make use of paths; this list is likely to evolve with changing and new application requirements. As such, a well-structured path selection architecture should be as agnostic as possible to the criteria applications use to select paths and the ways that applications use the available paths. Consider several representative examples:

2.1.1 Performance and load balancing

Duplicate or redundant streams: VoIP VoIP traffic must ultimately travel from one source to one receiver. Compared to bulk data transmission, VoIP streams are still relatively low-bandwidth. A VoIP application might therefore wish to use multiple source-destination paths concurrently to avoid quality degradation due to packet losses or path failures. It is worth noting that VoIP applications might also seek to minimize end-to-end latency as a part of their path selection.

Better throughput or load balance: Bulk transfers Several systems load balance data transfers across multiple TCP streams from a sender to a receiver to improve throughput [15, 26, 34]. Such systems require a way to gain access to multiple, hopefully independent sender-receiver paths. Unlike the VoIP example, bulk transfers are less likely to care about latency, preferring instead high-bandwidth paths.

2.1.2 Availability

Duplicate requests: DNS The Domain Name System can often benefit from sending queries over different paths to different servers, allowing it to avoid failed DNS servers as well as failed paths [5]. DNS requests are relatively

Multipath Scheme		Path Bits
Path Splicing [20]		IPv4: IP ID field is used to store the “splicing bits” TTL field is used to index in to those bits at each hop. IPv6: the bits could be included as a separate IPv6 extension header
Routing Deflections [33]		IPv4: IP ID field and TTL fields are used to calculate the index into the deflection table at the router
ECMP [14]		Hash the (src ip, dst ip, path bits) tuple instead of just (src ip, dst ip) to select one of the equal cost paths
Pathlet Routing [11]	Routing	Encode each pathlets (<i>i.e.</i> , a sequence of virtual nodes) onto a set of opaque bits
MIRO [31]		End systems can set bits to select the appropriate interdomain tunnel. These bits could be included in the IP ID field, as an interdomain MPLS tag [7], or as a separate header

Table 1: Mapping multipath routing protocols to path bits; we have implemented the ones shown in bold.

infrequent, small, and critical to establishing communication. Clients may wish to send multiple queries in parallel.

Fast recovery from failure: Video streaming or SCTP

A client may not wish to duplicate a large video stream, but may still desire fast fail over to a new path if the current one fails. While such clients might also use multiple paths concurrently, they may not wish to pay the implementation complexity of doing so.

Initial path selection only: Small HTTP requests

Selecting a path based upon the success of a TCP SYN packet provides a simple, effective way to load balance requests upon multiple paths and avoid many path failures [3, 5, 13]. Although this technique primarily works for small, stateless requests such as HTTP traffic, its simplicity makes it an attractive choice in many cases.

2.2 Different Path Selection Mechanisms

Multipath routing exposes multiple paths for each destination to each end system. Multipath routing can improve failure recovery by allowing end systems to react to failures more quickly than the underlying routing system would. If a multipath routing system allows an end host to use multiple paths simultaneously, it can also help improve throughput. Multipath routing can also improve robustness or security for end-system applications [29]. Multipath routing can be used to describe either interdomain or intradomain routing schemes; in this paper, we use the term “multipath routing” to refer to any scheme that can expose multiple paths between endpoints, regardless of whether it is implemented with interdomain or intradomain routing protocols. This section surveys various multipath routing schemes.

2.2.1 Mechanisms

Source-controlled The first class of mechanisms expose control to the source over the path that its traffic takes

through the network. Here, the source annotates the packet with additional information to signal to the network something about the path the traffic should take. The type of control ranges from implicit—from an opaque interface much like path bits—to explicit source routing. Implicit approaches include routing deflections [33] and Path Splicing [20]. Other methods, such as Pathlet Routing [11], provide explicit control to the source over the path that traffic takes en route to the destination, by specifying a path as a sequence of virtual nodes (“vnodes”).

Network-controlled, intradomain The second class of multi-path mechanisms operate entirely within the network. We liberally include methods such as IP fast re-routing [24] and MPLS fast re-routing [8], which maintain multiple paths to act as backups in response to failures. Using equal cost multipath routing (ECMP), the network dynamically load-balances traffic along equal cost intradomain paths. Our prior experience and that of others has shown that equal cost does not necessarily mean equal behavior—a failure may affect only one link of an ECMP load balanced group. ECMP could support a path-bits-based selection mechanism if routers select paths based upon the path bits in addition to the conventional four-tuple (source IP, source port, destination IP, destination port). In other network-controlled multipath schemes, nodes along the path annotate the packets themselves with failure information; other routers along this path use this information to route around paths that include the failed node or link [18].

Network-controlled, interdomain In contrast to the prior group, most systems for inter-domain multipath routing remain in the realm of research. In MIRO [31], networks send requests to neighboring networks to obtain access to alternate paths. MIRO, however, requires the networks themselves to request these additional paths through the control plane, and these alternate paths are not established in advance; such a technique may not scale in practice. Similarly, R-BGP [17] and Anomaly-Cognizant Forwarding (ACF) [10] add functions to the network that enable it to send traffic over a backup path when a failure occurs; neither of these approaches provides control directly to applications or end systems, although it is possible that path bits could augment these schemes to allow an end-host to explicitly select a backup path, even when path failure does not occur. BANANAS allows networks to stitch together an inter-domain path using a single Path ID, in the same way that a single MPLS label can be used in interdomain MPLS. This Path ID or MPLS label is similar in spirit to path bits. NIRA [32] allows edge networks some control over end-to-end paths, but path selection in NIRA

is explicit, assumes that hosts use provider-based addressing, and requires significant changes to packet headers.

Overlays Finally, overlay networks (e.g., RON [4], SOSR [13], MONET [5]) avoid the issue of changing the networking and routing layers by allowing end-hosts to tunnel traffic through different intermediate nodes. Path bits can serve as an interface to these overlays, as well: either end hosts, or the overlay nodes themselves, can set bits indicating the tunnel or intermediate node that traffic should take en route to the destination.

2.2.2 Implementation choices

Just as there are many multipath mechanisms that can be mapped to the path bits interface, there also exist numerous options for implementing these mechanisms. The diversity of available hardware and software platforms provide useful insight for creating path selection mechanisms that can be implemented efficiently on the routing platforms of today and, hopefully, into the future.

Hardware Several popular options exist for implementing multipath routing protocols in hardware; Section 6 describes our implementations of these hardware mechanisms. Supercharged PlanetLab offers programmable network processors [27]. OpenFlow [21, 30] allows a controller to install flow table entries that can match on certain fields in each packet header and direct traffic out certain switch ports. Such an abstraction could be used to implement a multipath routing scheme based on path bits: extra bits in a packet-header field (e.g., MAC address, IP ID, VLAN ID) could be used to index into different flow-table entries. Similarly, path bits could demultiplex packets into different forwarding tables on a NetFPGA card [1]. In fact, we have already implemented virtual forwarding tables in hardware for virtual routers [6]; we have this function instead perform demultiplexing based on path bits.

Software Multipath routing can also be implemented in software elements. We have implemented a forwarding module in Click [16] that uses path bits embedded in the IP ID field to index into different forwarding tables.

3. Design Goals

Our design has two goals: (1) Decouple the end systems and multipath routing mechanism so that multipath mechanisms can evolve independently from the applications that use them. Additionally, decouple the interface from the mechanisms that use the interface, so that the multipath protocols themselves can evolve independently of the interface. (2) Provide a simple interface to applications and afford a simple implementation. The following sections explore these design goals—and requirements that follow from these goals—in more detail.

3.1 Decouple End Hosts from Protocols

The interface should expose the most important capabilities of multipath routing mechanisms and protocols in an agnostic manner. The interface should be easy to use within applications, and should not require end hosts to maintain knowledge about the state of the network beyond the paths it is actively using.

Most existing multipath protocols base path selection upon information embedded in the data packets or provided through a separate control channel. Regardless of the control mechanism, many routing protocols require this information to be *specific*: It either directly tells the routers which path to select (e.g., classical source routing), or indicates to the routers a set of properties that the chosen path must satisfy (e.g., the original ToS bits in the IP header). A specific path interface induces an undesirable coupling between the application and the semantics of the path selection mechanism: applications built to this interface could assume a degree of control not available using other mechanisms, and the network could become reliant upon receiving this information from applications. Future applications would be forced to provide sufficient information for this interface, and future path selection mechanisms would be required to provide (a superset of) the existing semantics.

Our multipath routing framework, path bits, provides syntactic and semantic isolation between applications and multipath mechanisms by imposing only two, very minimal, requirements upon the multipath mechanism:

Ability for end systems to use multiple available paths

The interface should allow end systems to explore available paths in the network. Just as many multipath routing protocols do not expose all the routes that may be available in the native network, the path bits interface may not necessarily expose all the paths that the multipath protocol allows, but path bits should allow exploring enough paths to be *useful* for the variety of application requirements discussed earlier. It is not necessary for the interface to allow end systems to request specific paths, but the end system should be able to iterate through a set of paths and find at least one that meets its requirements.

Per-host consistent path selection The network must interpret the path selection information such that the same path preference expression will result in the same path choice until a routing reconfiguration occurs. Consistent path selection allows hosts to learn path properties and to ensure that flows that should receive similar treatment will follow the same path. The path bits do *not* specify that this information be consistent across end systems, even if those systems are within the same subnet. This choice has a pragmatic basis—ECMP and similar mechanisms already violate such a guarantee—and an engineering basis—requiring hosts to explore to find a well-

behaved path can, should the multipath mechanism desire it, help distribute end system traffic over a variety of paths. It does, however, prevent shared path information schemes such as SPAND [23] from working.

With these requirements, we note two specific goals that the “waist” should achieve:

Interoperation among different multipath protocols

The interface should allow different networks to provide different multi-path routing protocols, while permitting an end-host to take advantage of multipath routing across two or more such domains. A key requirement for interoperability is ensuring that the interface makes minimal assumptions about how the path selection information is interpreted within the network. We believe that the two requirements stated above do not impose an excess burden, and we show through evaluation that the path bits interface can be used to select paths in several different multi-path implementations. Backwards compatibility with the traditional single (shortest) path routing schemes is also one of our goals, but it follows naturally from the above requirement.

Balance control between end hosts and the network

The interface should balance control between the end systems and routing protocols. End systems should be able to easily achieve goals such as improved availability or load balance, but without introducing the aforementioned concerns about unpredictable traffic load or introducing the possibility of oscillatory path selection behavior.

3.2 Simple Interface and Implementation

The path bits interface should be simple, easy to use at the end systems, and easy to interpret and implement for multipath routing schemes in the network.

Ease of use at end systems The first aspect of ease of use is hiding the complexity of the underlying multipath routing scheme from higher layers, which path bits accomplish. The second aspect is creating an interface that is itself easy to use and allows programmers to easily manipulate familiar abstractions. For example, the programmer may wish to bind a flow to a path, or to change this binding. Another application may wish to split a flow onto multiple paths to improve throughput. Yet others may desire reliable host-to-host or host-to-service communication. The interface should allow these to be accomplished without difficulty.

Efficiently implementable in the network. The data-plane components of the interface should be easy to implement efficiently on routers and switches. This implies that decoding any packet-carried information should be straightforward, and that path bits should not require large

amounts of state in the routers and switches beyond that required by the underlying multipath scheme.

Scalable access to large numbers of paths. The architecture should scale well with multipath mechanisms that expose large numbers of paths. The memory requirements and processing required to map from the path bits to routes should remain constant as the number of available paths increases.

4. Design

Based on the design goals in Section 3, we describe the design of an interface that allows applications running on end systems to access multiple paths through the network using an opaque sequence of bits that we call *path bits*. The design of the path-bit interface entails design choices involving what the bits should represent and how many bits are needed for the application to request additional paths from the network. After discussing the design trade-offs involving the bits themselves (Section 4.1), we describe how simple modifications to end hosts (Section 4.2) and to the network devices (Section 4.3) can provide hosts access to multiple paths through the network.

4.1 Path Bits

Path bits are opaque bits that end-systems include in packet headers to control the forwarding decision made by routers on the path. Path bits can be added in a separate header or an additional header. In this section, we describe the options for (1) what semantics the bits could carry and how much control they should give to end hosts over the paths; (2) how many bits should be used for the interface.

4.1.1 Decision #1: Level of control

The first question concerning the design of path bits is what semantics they should carry, and how explicit they should be about the route that traffic takes to a destination. At one extreme, these bits might explicitly denote an entire source route; at the other extreme, the bits might simply encode the desire for a source to have a new path.

To provide a simple, scalable interface to hosts and to balance control between end hosts and the network, *path bits should be opaque*. They should not expose semantics as to which particular path is being used. Rather, they should provide the property that *changing the bits will, with high probability, yield a different path to the destination*. This abstraction is based on the insight that end systems typically do not care about the specific sequence of hops that traffic takes through the network, as long as they can have easy access to a set of good paths or alternately can avoid bad (*e.g.*, lossy or failed) paths [4, 13].¹

¹One often-hypothesized requirement is that a path might wish to avoid going through a particular location or country; such an application is compatible with a path bits specification. In contrast, expressing a re-

Opaque path bits still leave considerable flexibility as to the actual semantics of the bits themselves. The bits might explicitly encode information about how each node along the path should forward traffic (*e.g.*, having a fixed number of bits per hop), or they might simply encode a request for the network to change the path (a request that could be encoded in a small number of bits, or even a single bit).

Opaque bits satisfy our two high-level design goals: (1) decoupling end hosts from the specific multi-path protocols; and (2) providing a simple interface. Opaque bits allow individual networks to retain control over how routers and switches interpret bits and allow them to apply their own policies regarding security, traffic engineering, etc. Opaque bits provide a basis for randomization mechanisms to prevent oscillations when many end systems simultaneously switch between the same paths in response to congestion.

While our design is, at some level, a clean-slate notion—it requires modification to a large number of components—a backwards compatible implementation of path bits can be embedded in the packets in such a way so that networks or routers that do not support multi-path still forward the packets based on their destination IP address. For example, path bits could be embedded in the IP ID or TOS fields of the IPv4 header (as shown in Section 6); in the case of IPv6, packets already include a flow label that could be used to carry path bits or we can have a separate IPv6 extensions or Options header to incorporate path bits.

4.1.2 Decision #2: Number of bits

Path bits should provide end hosts as much control over paths as is necessary to access a diverse set of paths, without imposing needless overhead. The mechanism for interpreting path bits could, in principle, work for anywhere from a single bit to $l \log_2 k$ bits, where l represents the maximum number of hops along any network path and each hop along the path has as many as k bifurcations. The *one-bit* mechanism offers only coarse control, but is obviously compact. The opposite extreme, the *full bits* mechanism—encoding each hop as a sequence of bits—instead offers maximal flexibility because the bits encode, for each hop, the forwarding choice to be made at that hop.

Practically, increasing the number of bits provides diminishing returns as it exceeds either the number of paths provided by the underlying multipath routing scheme, or the number of paths that behave differently (*e.g.*, the number of failure-independent or bottleneck-independent paths). Picking a *specific* number is obviously an engineering tradeoff, but these practical limits suggest, first, that the number should be constant—not based upon the path length—and that it should be small, but not so small *quirement for* a particular route is more difficult, but such requirements mostly arise in a functional context, such as wishing to route traffic through a firewall [28], a capability beyond the intent of our architecture to provide.

that it forces implementations to maintain complex mappings between the bits and the resulting path. Based upon our implementations of existing schemes, we elect to initially use sixteen bits for the path bits.

Table 1 details several mappings of these bits to proposed multipath mechanisms, several of which we implement in Section 6. We briefly illustrate here how many multipath functions can be achieved using a small number of path bits. Consider the case where full bits are used, but the bits are opaque. In this case, the end system has no information about what will actually happen at each hop, so changing the bits at each hop has the same effect as changing a smaller number of path bits at random. One way to implement this mechanism with a smaller number of bits is to embed in the packet header a small number of bits which form the input to a hash function at each router. Specifically, each node along a path might select a next-hop indexed between 1 and n where the index is $H(s, d, i, p)$, where (s, d) represents the source-destination pair, i is the node’s identifier, and p are the path bits. Such a mechanism yields per-hop control over the path but allows path choice to be implemented in a smaller number of bits.

4.2 End-System Interface

End systems (or applications running on end systems) need an interface and mechanism to set path bits when they require a different path. As noted earlier, our goal is to keep this interface simple, but also to enable application programmers to use it at a familiar level of abstraction. To do so, the end system network stack maintains a mapping of flows to the bits used by the flow. It then provides an interface for either the application or a higher-level protocol (*e.g.*, TCP) to modify these mappings. We have implemented one such mapping (Section 5) using Click [16] running as a kernel module to replace the networking stack and incorporate a table that maps flows to path bits.

Because many applications may have similar requirements for path quality (and therefore, similar criteria for selecting and changing paths), this interface leads naturally to allowing a shared monitoring application to evaluate path quality and change path bits appropriately. As a result, even unmodified applications can benefit from, *e.g.*, rapid path switching to alternate paths. Multipath-aware applications, or those with specific requirements such as concurrent multipath use, can instead use the lower-level flow-binding mechanisms.

4.3 Network Support

Routers interpret the path bits in a packet to direct traffic to the appropriate outgoing interface. We strive to implement a mechanism such that, (1) with high probability, packets with different path bits will traverse paths to the destination; and (2) packets with the same path bits will traverse the same path en route to the destination (barring

route reconvergence in the face of failures). The simplicity of this abstraction means that a variety of network devices can implement different multipath schemes with only minimal additional functionality in the forwarding plane.

We provide network support for different multipath routing protocols by having the bits serve as an index into different forwarding table entries and changing *how* the path bits are used to index into the different forwarding table entries. For example, routing deflections [33] could be implemented at a router by using the path bits to switch forwarding to a pre-computed next-hop that is closer downstream to the destination. Path splicing [20] could be implemented by using the bits as an index into one of k pre-computed slices. Pathlet routing [11] could be implemented by pre-computing labels for pathlets, and setting up those paths in the forwarding tables, and using the bits to index into different pathlets. A variant of ECMP could be implemented by installing multiple paths of almost equal length into the forwarding table and using the path bits to select one of multiple possible next hops. Table 1 describes how to map some of the multi-path routing schemes to the path bits interface.

A key design decision for implementing the narrow waist at network devices is *which* bits the devices should use to index into different forwarding table entries. Options include the IP ID and TOS fields in the IP header to the VLAN ID in the layer-two header. The main criteria are: (1) every network device along the path that interprets path bits should interpret the same set of bits; (2) the path bits should not affect other functions in the network. For most of our implementations in Section 6, we use the IP ID and TOS fields; because the current implementation of OpenFlow does not allow rules for these fields, we use the VLAN ID for our OpenFlow implementation.

5. End-System Implementation

End-system support for path bits consists of an interface for applications to specify the performance metrics that are relevant for them and a monitoring daemon that monitors the path quality on behalf of the applications. Figure 2 shows an overview of the extensions we make to end hosts to support the path bits interface. The end-system implementation has three components: (1) a socket capture library; (2) a path monitoring daemon; and (3) a kernel interface. In this section, we describe and evaluate each aspect of the implementation.

Our goals in evaluating the end-system component are to show that the interface is simple and general, that the interface to the monitoring daemon permits a variety of monitoring strategies, and that the function enabled by path bits can be implemented simply and efficiently.

5.1 Socket Capture Library

Applications that wish to use multiple paths may do so without modifications. To keep track of a set of desirable

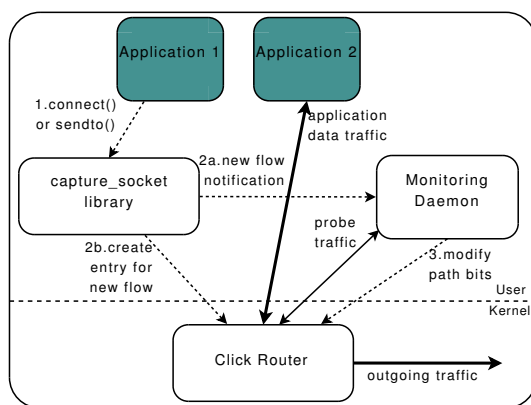


Figure 2: Design of the different components of the failure recovery system at the end system. Applications supply policy configuration files to the monitoring daemon which uses the information to monitor performance of paths. Click router adds appropriate routing bits to data traffic as specified by the monitoring daemon.

paths for each application, the system provides a socket capture library that intercepts `connect()` calls for TCP and `sendto()` calls for UDP flows. The library determines when new flows are initiated from the end host and works transparent to the application as shown in Figure 2.

5.2 Kernel Interface

Our current implementation runs Click [16] as a kernel module to replace the Linux network stack. The Click module consists of a table that stores information about which path bits are currently used for a particular flow. The Click module captures packets from the network interface, matches the packet flow identifier with the entries in its table, and applies the corresponding path bits if there is a match. The Click module also provides an RPC interface for the monitoring agent to read, modify, and delete path bits from the table. The module also provides support for wild-card entries: the source and destination ports could be wild cards to enable matching for all packets going to the destination. The module considers specific matches to be higher priority than wild card matches.

5.3 Path Monitoring Daemon

The monitoring daemon runs as a separate process that monitors paths either actively or passively based on the requirements specified by the applications. It updates the path bits in the kernel to reflect the current set of best paths for each application that satisfy the performance requirements specified by the applications. The monitoring daemon in our prototype uses active probes to monitor paths; the framework is general enough to support different monitoring strategies. Our monitoring daemon is intended mostly as a proof-of-concept to ensure that the *interface* works and is general, and not to design or evaluate path monitoring strategies, deferring such tasks to future work.

The monitoring daemon receives a notification from the socket capture library when a new flow is initiated by

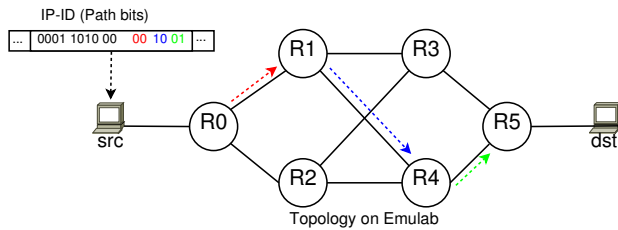


Figure 3: Topology used for Emulab experiments. The source and destination nodes are Emulab hosts connected to other Emulab nodes acting as routers. The different colored arrows show the next hop for reaching the destination along an example spliced path.

an application. The current implementation receives the identification of each flow (the four tuple $\langle \text{src ip}, \text{dst ip}, \text{src port}, \text{dst port} \rangle$), as well as acceptable thresholds for the latency and loss rate which the flow for each application type. The application can, if it chooses to use the notification interface explicitly, specify if the flow should use a single best path or multiple paths simultaneously.

The daemon records performance information in a monitoring table for a set of paths (*i.e.*, paths with different path bits). The application can specify to the monitoring daemon how many alternate paths to monitor on its behalf. The daemon then periodically sends probes on each of the monitored paths and updates the performance information in the table. If the performance of the current path fails to meet the latency and loss rate thresholds specified by the application, the daemon communicates with the kernel module via RPC to initiate a path switch. The daemon replaces paths in its monitoring table that do not meet the thresholds with a new paths that are selected via a selection of random path bits. The monitoring daemon interacts with the Click kernel module on the end system to set or modify path bits for the monitored flows.

6. Network Implementations

To understand the generality, power, and efficiency of the path bits interface, we implemented three multipath routing schemes on a mix of four different platforms: software-based implementations in Click, and hardware implementations on NetFPGA, OpenFlow [21], and the Intel IXP network processor. One of the main goals of this exercise is to demonstrate that the simple interface provided by path bits—and its function as an index into multiple forwarding tables—can be used to implement a variety of multipath routing schemes in very few lines of code and using only modest resources. In all cases, we embed path bits in the IP ID field in the IP header. The TTL field value is used to index into the appropriate set of bits in the IP ID field by the routers along the path. All of our implementations, along with details on configuration, are available on our project Web page [2].

6.1 Click Software Router

We implement three multipath schemes as Click elements: Path Splicing, Routing Deflections, and a modified version of ECMP that we call ECMP++. The Click modules for these are available for use by researchers on our Web site [2].

6.1.1 Path Splicing

Figure 3 shows the experimental topology with an example of a spliced path. The IP ID field encodes the forwarding tree that each router along the path should use to forward the packet en route to the destination. Routers read the appropriate bit position in the IP ID field according to the index in the TTL field; the IP ID serves as an index into the forwarding table (“slice”) that the router should use to forward the packet. In our experiments, we use the TTL field instead of the TOS field because switches in the Emulab testbed filter packets with certain TOS values. The code for the Path Splicing implementation consists of a new Click element that reads the IP ID field in the packets and selects the appropriate routing table to use for forwarding the packet. *This element (GetSlice) required only seven semicolon-containing lines of C++*, as shown in Figure 4(a). Lines 7–10 extract the forwarding table index number (or slice) from the IP ID and TTL fields of the packet. The Click element then outputs the packet on the appropriate output port of the element (Line 11). configuration file

Each router has a Click configuration file that specifies the connections for the multiple forwarding tables at the router and connects the output of the Click element *GetSlice* with the appropriate forwarding table. For example, Figure 4(b) shows the Click configuration for router R2 in Figure 3. Lines 16–18 direct the output of *GetSlice* to the appropriate routing table, whose connections are specified for example in Lines 20–22 in Figure 4(b).

6.1.2 Routing deflections

Routing Deflections [33] uses the bits in a similar manner as Path Splicing. Each router has a *deflection set* consisting of next hops that may be used for a packet depending on the packet’s previous hop and destination address. Thus, the deflection set is a function of (*ingress interface, destination ip address*). The IP ID and TTL fields index into the deflection set at each router, as described in [33]. We implemented Routing Deflections by precomputing the deflection set for each router and including it in the Click configuration files. We also implemented a Click element that reads the IP ID and TTL fields to output the index number in the deflection set. This Click element, shown in Figure 5(a), is about nine semicolon-containing lines of C++. Lines 7–12 compute the *tag*, which is extracted only if the TTL is greater than 160 and less than 200. The tag is then used to compute the index into the

```

1 void GetSlice::push(int port, Packet *p_in)
2 {
3     const click_ip *ip_in = p_in->ip_header()
4     ;
5     assert(ip_in);
6
7     //extract the slice number based on TTL
8     uint16_t my_id = ntohs(ip_in->ip_id);
9     uint8_t ttl = ip_in->ip_ttl;
10    uint8_t index = ttl % 8;
11    uint16_t slice = ((my_id & (0x0003 << (2
12    * index))) >> (2 * index)) & 0x0003;
13    output(slice).push(p_in);
14 }

```

(a) Click element code.

```

1 elementclass RTable { $dest0,$dest1 |
2     input -> rtable :: RadixIPLookup(
3     $dest0 0,
4     $dest1 1,
5     0/0 2.);
6     rtable[0] -> [0] output;
7     rtable[1] -> [1] output;
8     rtable[2] -> [2] output; }
9
10 slicer::GetSlice;
11 //Create 4 routing tables
12 rtable0 :: RTable(src_ip, dst_ip);
13 rtable1 :: RTable(src_ip, dst_ip);
14 ...
15 //GetSlice connected to appropriate table
16 slicer[0] -> DecIPTTL-> rtable0;
17 slicer[1] -> DecIPTTL-> rtable1;
18 ...
19 //Connections for table 0
20 rtable0[0] -> to_router0
21 rtable0[1] -> to_router4
22 rtable0[2] -> Unstrip(14) -> ToHost();
23 .... //Three more tables

```

(b) Click configuration file.

Figure 4: Path Splicing.

deflection set for the incoming interface and destination IP address (Line 13).

Figure 5(b) shows the Click configuration for the Routing Deflections elements at Router *R2*. The configuration is similar to the Path Splicing configuration, except the configuration now specifies a deflection set for each neighbor, as opposed to individual routing tables per slice. Lines 3–7 configure the *GetDeflection* element and the connections to the routing tables (*deflection set*) for packets coming from neighbor *R0*.

6.1.3 ECMP++

ECMP++ is a version of ECMP in which the outgoing interface for a packet is decided based on hash of (*src ip*, *dst ip*, *path bits*) in the packet header. The path bits are included as part of the IP ID field in the packet header; the TTL field can be used similar to the implementation of Path Splicing to help the routers index in the IP ID field to read the path bits corresponding to the router. Figure 6 shows the implementation of this Click element; in the interest of space, we have not shown the Click configuration for this setup. It is similar to that for other multipath

```

1 void GetDeflection::push(int port, Packet *
2     p_in)
3 {
4     const click_ip *ip_in = p_in->ip_header()
5     ;
6     assert(ip_in);
7
8     uint16_t my_id = ntohs(ip_in->ip_id);
9     my_id = my_id & 0x03ff; //use only the
10    //last 10 bits
11    uint8_t ttl = ip_in->ip_ttl;
12    uint16_t tag = 0;
13    if(ttl > 160 && ttl < 200) {
14        tag = my_id;
15    }
16    uint16_t deflection = (tag % prime) %
17    //size;
18    output(deflection).push(p_in);
19 }

```

(a) Click element code.

```

1 ....
2 //Deflection set for neighbor R0
3 deflect_0 :: GetDeflection(2);
4 rtable_R0_0 :: RTable(src_ip, dst_ip);
5 rtable_R0_1 :: RTable(src_ip, dst_ip);
6 deflect_0[0] -> DecIPTTL -> rtable_R0_0;
7 deflect_0[1] -> DecIPTTL -> rtable_R0_1;
8
9 //Deflection set for neighbor R3
10 deflect_3 :: GetDeflection(2);
11 rtable_R3_0 :: RTable(src_ip, dst_ip);
12 rtable_R3_1 :: RTable(src_ip, dst_ip);
13 ...
14
15 //Connections for pkts coming from neighbor
16 //R0
17 rtable_R0_0[0] -> to_router0;
18 rtable_R0_0[1] -> to_router4;
19 rtable_R0_0[2] -> Unstrip(14) -> ToHost();
20
21 rtable_R0_1[0] -> to_router0;
22 rtable_R0_1[1] -> to_router3;
23 rtable_R0_1[2] -> Unstrip(14) -> ToHost();
24 .... //Similarly for other neighbors

```

(b) Click configuration file for router R2.

Figure 5: Routing Deflections.

routing implementations. Lines 6–9 are same as in implementation of Path Splicing, while Lines 11–14 determine the output port (either 0 or 1) by hashing the ip addresses with the path bits.

6.2 NetFPGA

We implemented Path Splicing and Routing Deflections using NetFPGA [1]. Our implementations are loosely based on the implementation for building a fast, virtualized data plane with NetFPGA [6]. We implemented these schemes on the Xilinx Virtex-II Pro 50 FPGA.

6.2.1 Path Splicing

The implementation instantiates four forwarding tables and four ARP tables in the base router. Because a destination IP address can exist multiple forwarding tables, the implementation requires separate ARP tables to have different ARP entries for same IP address.

```

1 void GetECMP::push(int port, Packet *p_in)
2 {
3     const click_ip *ip_in = p_in->ip_header();
4     assert(ip_in);
5
6     uint16_t my_id = ntohs(ip_in->ip_id);
7     uint8_t ttl = ip_in->ip_ttl;
8     uint8_t index = ttl % 8;
9     uint16_t bits = ((my_id & (0x0003 << (2 *
10         index))) >> (2 * index)) & 0x0003;
11
12     int src_addr = ip_in->ip_src.s_addr;
13     int dst_addr = ip_in->ip_dst.s_addr;
14     int hash = Hash(src_addr, dst_addr, bits);
15     hash = hash%2;
16     output(hash).push(p_in);
17 }

```

Figure 6: Click element for ECMP++.

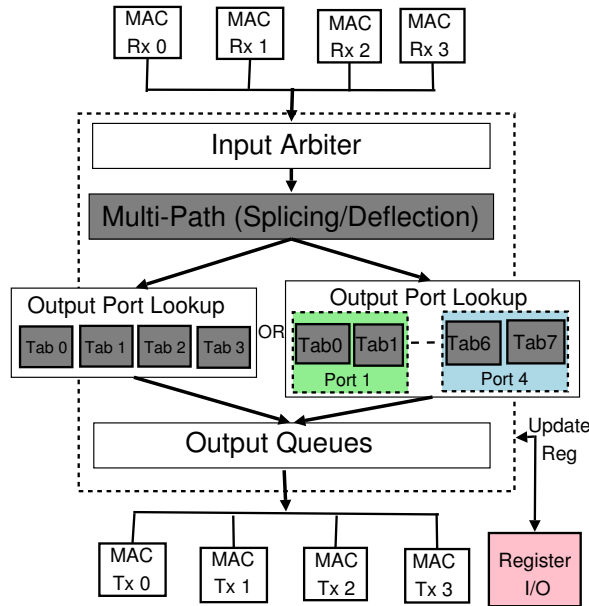


Figure 7: Router pipeline for the NetFPGA implementation of Path Splicing.

The implementation uses four 32-entry TCAMs and four 32-entry ARP tables. The lookup modules are implemented using SRL16e while ARP CAMs are implemented using dual port Block RAM (BRAM). These 32-entry tables correspond to available resources on the NetFPGA base router implementation: the card has one 32-entry TCAM longest-prefix match module with its lookup table and one 32-entry ARP table.

Our implementation creates the path bits using three bits from the TTL field plus the IP ID field. The high three bits are used to divide the lower 16 bits into eight entries. Each entry determines which of the four routing tables should be used at each hop. We use on-chip memory to store the forwarding tables. The base router implementation used in this design is the reference router implementation from the NetFPGA group [1]. Figure 7 shows the base router implementation and the modules that are added or modified

in the reference design. The path splicer performs collects the path bits and informs the output port lookup module which forwarding table to use to determine the next hop.

By separating forwarding table selection from forwarding, the *multipath module* in Figure 7 allows designers to use any kind of path bits selection mechanism that can act on the packet header. We use this feature to ease the implementation of both Path Splicing and Routing Deflections. In Section 4.3, we observed that the common denominator among different multipath schemes is to have separate forwarding tables for different paths; different multipath algorithms merely change how the path bits are used to select among these forwarding tables. Our *multipath* module can be used easily to implement a new multipath routing scheme by changing the table selection register shown in Figure 7.

Resource utilization We use Xilinx’s ISE 9.2i to determine the fraction of the FPGA used for our implementation. Path Splicing accounts for approximately 62% of LUTs on the Virtex-II Pro FPGA that is available on the NetFPGA card for logic, as shown in Table 2. Of this percentage, approximately 7.2% of LUTs are used for shift registers. Route through is responsible for 7.1% of four input LUTs. This implementation uses 162 BlockRAMs, or 69% of available BRAMs. A more efficient implementation can reduce the resource utilization further, however, our focus here was towards a more generic design to ease implementation of different multipath mechanisms.

Performance Figure 8 shows the NetFPGA based testbed we used to test our Path Splicing implementation. We used the NetFPGA-based hardware packet generator [9] developed by the NetFPGA group to send and receive traffic. All nodes shown in Figure 8 are rack-mountable servers with one NetFPGA card installed in each of them. Router R0 in Figure 8 allowed us to measure the effectiveness of splicing, whereas routers R0 and R2 helped us determine the efficiency of the implementation at line rates. This topology, though minimal, allows us to study whether the basic implementation is functional.

Figure 9 shows the packet forwarding rates of this NetFPGA-based implementation, as observed at the sink node. While observing these rates, no packet loss occurred on any of the nodes shown in Figure 8. The blue bars show the results when we sent two flows with same destination IP address but using different path bits to direct them to different intermediate router. Packets from one flow were sent to R2 via R1, while the other went directly to R2.

In another iteration, we introduced four different flows in the network, such that all four forwarding tables at router R0 and R2 were looked up with equal probability. Four is the maximum number of forwarding tables sup-

Router	Total LUTs	% LUT Utilization	Total BRAMs Usage
Base Router	23K	45%	123
Path Splicing	30K	62%	162
Deflections	35K	73%	194

Table 2: Multi-path scheme’s hardware resource usage

ported in our current implementation but this can easily be increased to eight forwarding tables.

Both these experiments show that multipath routing schemes like Path Splicing can be implemented using hardware without sacrificing performance. Path splitting of flows on $R0$ shows that our implementation is efficient to split traffic while looking up in two different forwarding tables at line rate. In second iteration where we increased the number of flows, it resulted in lookups in all four forwarding tables but still there was no performance degradation. From our experiences in [6] we are confident that this number can be increased to eight for multi-path routing without sacrificing line rate performance.

6.2.2 Routing deflections

Design and implementation We modify the *multipath* module, as shown in Figure 7 to design the pipeline for implementing Routing Deflections. We implement a slightly modified design for Routing Deflections to make it more similar to the Path Splicing implementation, where each router looks up the appropriate path bits from the IP ID field based on the value of the packet’s TTL. Since the NetFPGA [1] card has four Ethernet ports, we can instantiate eight forwarding tables with eight ARP tables on it. For each incoming packet the input port appends its port number to the packet.

Each incoming packet’s port number, TTL and IP ID fields determine which routing table will be looked up for the particular packet. Four input ports divide eight forwarding tables into four sets of twos. Then we use TTL and IP ID field to select one of the two forwarding table for that particular packet. For this we use four LSB bits of TTL field to index into 16-bit field of IP ID. Depending upon the value in the IP ID field either first or 2nd table is selected for the particular packet.

Resource utilization and performance Routing Deflections implementation in hardware uses 73% of available LUTs on the FPGA as shown in Table 2. Of this 73% LUTs, 12% LUTs are used by shift registers. Route through is responsible for 6.4% of total 73% LUTs being utilized. BlockRAM usage accounts for 83% of available BRAMs on FPGA.

This increase in BRAM and LUT usage can be attributed to increased number of Forwarding and ARP tables in routing deflections implementation. Since we are in regression testing phase therefore we are not reporting forwarding rates for this implementation but we can

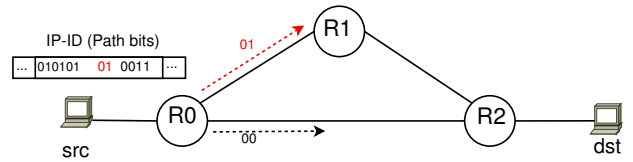


Figure 8: NetFPGA-based router testbed topology.

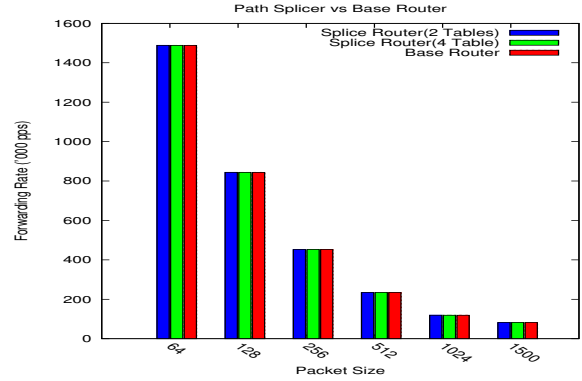


Figure 9: Path Splicing router performance with varying load compared with base router.

say with some confidence that this implementation should not result in any performance degradation considering the pipelined architecture being used in design.

6.3 OpenFlow

We describe our design and implementation of Path Splicing using OpenFlow. OpenFlow is suited for deployment in an enterprise network or in a datacenter network environment, where all the network elements (*i.e.*, the routers and switches) are owned by a single entity. OpenFlow setup consists of switches which support the OpenFlow specification and an OpenFlow controller. The controller has full view of the topology and can communicate with the OpenFlow switches to install forwarding rules. The controller also receives any frames which do not match the forwarding rules in the switches.

Design Using the switch topology information, the OpenFlow controller computes multiple spanning trees for the network topology and installs the appropriate forwarding rules in the OpenFlow switches. When a new end host sends a frame to the switch that it is attached to (*e.g.*, for an ARP request), the switch will not find a matching rule in the flow table and forwards the frame to the controller. The controller uses this frame to learn the location (*i.e.*, switch and port) of the host. The controller then installs multiple forwarding rules on *all* the switches in the network, based on the spanning trees that it has computed; there is one forwarding rule per spanning tree. Subsequently, as other hosts send frame to this host, the switches forward them using the rules installed by the controller. To avoid loops for broadcast frames, the controller installs rules to

forward broadcast messages from each host on only one of the spanning trees.

Because current OpenFlow switches allow specifying forwarding rules based on limited number of fields in the Ethernet frames or IP packets, we cannot use the IP ID or TTL fields for the path bits. Instead, we use the source VLAN ID tag field to carry the path bits; this ID actually lends itself to a natural mapping between a single network that provides multiple paths and a network that is overlaid with multiple networks. Using the VLAN ID field also preserves semantics at layer two and higher, since no other fields in the Ethernet frame or IP header are modified. Unfortunately, unlike the IP header fields, the VLAN ID is not modifiable by applications. To allow hosts to modify the VLAN ID, hosts must implement a module that copies *path bits* to the VLAN ID tag field in the packets when they are sent on the network. This design is feasible, because an enterprise (or datacenter operator) has much tighter control over the host operating systems.

We have implemented the above design with a custom NOX controller [12] and reference software switches for a three switch and two host network topology, similar to NetFPGA testbed topology shown in Figure 8. We are in the process of implementing the host modifications to allow setting the VLAN ID.

Resource utilization Implementing Path Splicing using OpenFlow requires additional space in the switch flow tables, as well as additional communication overhead with the controller. If there are k trees and N active hosts in the network, then we need kN rules in the flow table of every switch. By comparison, a classical learning switch that is part of a single spanning tree maintains N entries in its bridge table. Second, the network incurs overhead in terms of communication with the controller. If there are M switches in the network, the controller sends kM messages: one message per switch per spanning tree. The host’s switch only forwards the first frame from a host to the controller, so this overhead is fixed. Our current implementation does not refresh the rules or expunge stale entries; these functions are important if hosts are silent for extended periods, leave the network, or relocate in the network. Implementing these features requires kMN messages per refresh cycle across all switches. The network in Figure 8 has 4 rules (2 trees, 2 hosts) in switches R0 and R2. R1 has only 2 forwarding rules, one per host, because it is part of only one of the spanning trees. We can optionally install rules on R2 so that it drops frames carrying VLAN ID of the other spanning tree.

6.4 Network Processors

We also developed an implementation of Path Splicing using the programmable Network Processors (Intel IXP) from the Open Network Lab [22, 30]. The implementation consists of a plugin for the Intel IXP network processors

```

1 void handle_pkt_user() {
2     ...
3     // Calculate the buffer descriptor location in
4     // SRAM
5     onl_api_get_buf_handle(&buf_handle);
6     bufDescPtr = onl_api_getBufferDescriptorPtr(
7         buf_handle);
8     // Read the buffer Descriptor from SRAM
9     onl_api_readBufferDescriptor(bufDescPtr, &
10        bufDesc);
11    dramBufPtr = onl_api_getBufferPtr(buf_handle);
12    // calculate ip header DRAM ptr and read into a
13    // buffer
14    ipv4HdrPtr = onl_api_getIpv4HdrPtr(dramBufPtr,
15        bufDesc.offset);
16    onl_api_readIpv4Hdr(ipv4HdrPtr, &ipv4_hdr);
17
18    if(ipv4_hdr.ip_v == 4) {
19        ip_id = ipv4_hdr.ip_id;
20        index = ipv4_hdr.ip_ttl % 8;
21        //Get the appropriate routing table number
22        slice = ((ip_id & (0x0003 << (2 * index))) >>
                (2 * index)) & 0x0003;
                helper_set_meta_default(MUX);
                helper_set_meta_mux_tag(slice);
    }
}

```

Figure 10: Plugin code for path splicing on network processors.

that processes the IP ID and TTL fields from the packet headers and extracts the corresponding forwarding table number. The relevant part of the plugin code, which is written in about 30 lines of C++, is shown in Figure 10. Our tech report [19] has more details about this and our other implementations.

7. Building Applications Using Path Bits

We perform experiments to demonstrate the benefits path bits could provide to applications. Using a very simple and naïve monitoring algorithm (as described in Section 5.3), end-systems can still benefit from multiple paths. We show two benefits: recovery from failed or highly lossy paths, and higher throughput by using multiple paths simultaneously. We also explore some trade-offs of our monitoring algorithm. We performed all of these experiments on Emulab; the end system is running the implementation described in Section 5, and the routers are Emulab nodes running the Click software router implementations of Path Splicing, Routing Deflections, and ECMP++, as described in Section 6.1.

7.1 Failure Recovery

Real-time applications must quickly find a working path in the network. Available bandwidth is also a concern, but the overriding criteria is that the network is able to provide a working path to the application and fast recovery in the event of failures. We run our monitoring daemon and fail links from the topology shown in Figure 3. We measure the average number of path switches and average failure recovery time for different parameters of the monitoring algorithm. Table 3 shows the overheads depend-

Paths Monitored	Average Switches	Average Recovery Time (seconds)	Probing Rate (probes/second)
1	1.7	1.4	2
2	3	4.3	3.8
4	2.5	3.8	7.5
8	1.85	2.1	13.7

Table 3: Monitoring overhead.

ing on the number of paths that the monitoring algorithm monitors simultaneously. The probing overhead (in terms of the number of probes) increases as the monitoring daemon monitors a larger set of paths. Owing to the small size of the topology used for our experiment, we find that just monitoring one path and switching to a new random path when the performance of the path degrades is sufficient for fast recovery.

The plots in Figure 11 use the same Emulab setup as shown in Figure 3. In the experiment, we randomly introduce loss on each of links between the nodes $R1$, $R2$, $R3$ and $R4$. We also recover the links after some interval of time. The experiment consists of sending a TCP flow from src and dst nodes and recording the throughput observed at the dst . Figure 11(a) shows the baseline, where there is no monitoring of paths or switching of paths while the TCP flow is in action. We then run the experiment by making the routers in the network aware of multiple paths and running the monitoring daemon to monitor the paths and switch the TCP flow to a different path, via the setting of appropriate path bits, when it detects high losses on the current path. Figures 11(b) and 11(c) shows the throughput plots when the network is using Path Splicing and ECMP++ respectively, as the multi-path mechanism.

We did not modify the monitoring daemon even when we changed the multi-path mechanism being used by the network. The path bits mechanism ensures that the monitoring can be oblivious to the underlying multi-path mechanism. We can also run multiple, multi-path protocols at the same time in the network. In our setup, it is possible to have a few routers running Path Splicing and the others running Routing Deflections or ECMP++.

7.2 Bulk Transfer

Bulk transfer applications can benefit from simultaneous use of multiple paths in the network. Also, critical but short communications like DNS queries can be simultaneously sent on multiple paths to get a quick reply (in case one of the paths or DNS servers has failed), which is critical to establishing communication. In this experiment, we allow a flow to use multiple paths, specified by different path bits simultaneously. The Click module can keep multiple entries corresponding to each flow in its path bitstable. For enabling use of multiple paths, the Click module simply multiplexes packets from a flow on each of the multiple paths in a round-robin fashion. We are aware of more sophisticated techniques like flowlet switching [25] which ensures that packets are sent on paths with similar

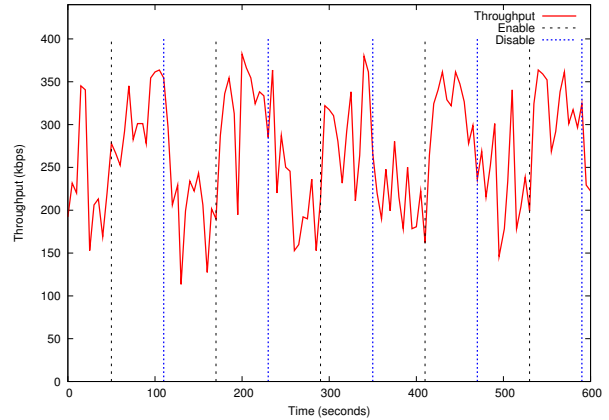


Figure 12: Higher throughput by simultaneously using multiple paths in parallel for a single TCP connection.

delay to avoid issues with reordering of packets in a flow. However, our goal here is to demonstrate the use of multiple paths by a flow. Figure 12 shows the throughput plot for a flow on which the use of multiple paths (two paths in this case) is enabled and disabled periodically.

8. Conclusion

Many networked applications benefit from access to multiple paths for improved performance and rapid failure recovery. Despite its benefits, however, no standard interface has emerged that provides applications access to these paths, nor one that multipath mechanisms can use to provide such access. *Path bits* is a new narrow waist for multipath routing—a standard interface that makes minimal demands of applications and of the network—that we believe will enable evolution of protocols and implementations below the waist, and applications above it. The interface is simple, and thus easy to use by applications, and, while general, admits efficient implementation in both hardware and software, as demonstrated by our implementation of a path bits interface on four hardware/software platforms for three different multipath mechanisms.

We have released the implementations from this paper to the community as the first framework that will allow the community to compare both different multipath algorithms and different real-time monitoring and recovery frameworks in a common context. We hope that both researchers and practitioners will extend and evolve this reference implementation to support new multipath routing protocols implementations, new applications that use the interface, and provide access to multipath routing in new environments, from interdomain routing to data centers.

REFERENCES

- [1] NetFPGA. <http://www.netfpga.org>.
- [2] Path Splicing. <http://www.gtnoise.net/splicing/>, Sept. 2009.

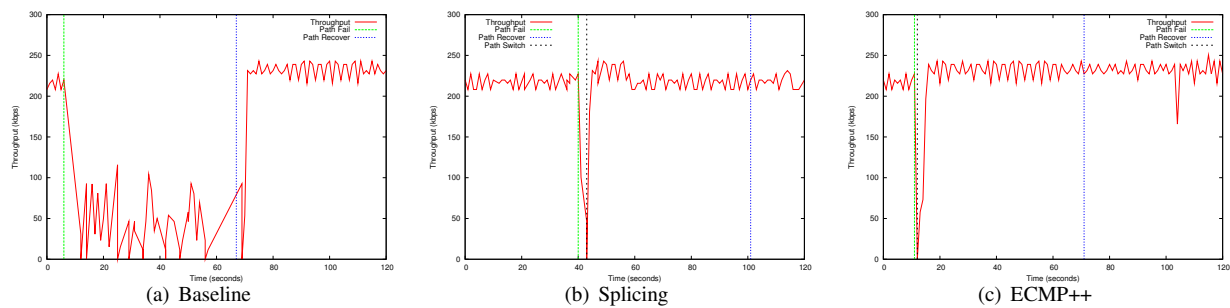


Figure 11: Failure Recovery Experiment on Emulab.

- [3] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.
- [4] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, Oct. 2001.
- [5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving Web availability for clients with MONET. In *Proc. 2nd USENIX NSDI*, Boston, MA, May 2005.
- [6] M. B. Anwer and N. Feamster. Building a Fast, Virtualized Data Plane with Programmable Hardware. In *Proc. ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, Barcelona, Spain, Aug. 2009.
- [7] A. Ayyangar, K. Kompella, J. Vasseur, and A. Farrel. *Label Switched Path Stitching with Generalized Multiprotocol Label Switching Traffic Engineering (GMPLS TE)*. Internet Engineering Task Force, Feb. 2008. RFC 5150.
- [8] MPLS Traffic Engineering Fast Reroute – Link Protection. <http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120st/120st16/frr.htm>.
- [9] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown. A Packet Generator on the NetFPGA platform. In *FCCM '09: IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009.
- [10] A. Ermolinskiy and S. Shenker. Reducing Transient Disconnectivity Using Anomaly-Cognizant Forwarding. In *Proc. 7th ACM Workshop on Hot Topics in Networks (Hotnets-VII)*, Calgary, Alberta, Canada., Oct. 2008.
- [11] B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet routing. In *Proc. ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.
- [12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.
- [13] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.
- [14] C. Hopps. *Analysis of an Equal-cost Multi-Path algorithm*. Internet Engineering Task Force, Nov. 2000. RFC 2992.
- [15] H.-Y. Hsieh and R. Sivakumar. pTCP: An end-to-end transport layer protocol for striped connections. In *IEEE International Conference on Network Protocols (ICNP)*, Paris, France, Nov. 2002.
- [16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.
- [17] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-BGP: Staying connected in a connected world. In *Proc. 4th USENIX NSDI*, Cambridge, MA, Apr. 2007.
- [18] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving Convergence-Free Routing with Failure-Carrying packets. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.
- [19] M. Motiwala, M. bin Tariq, B. Anwer, D. Andersen, and N. Feamster. A Narrow Waist for Multipath Routing. Technical Report, number forthcoming, Georgia Institute of Technology, Oct. 2009.
- [20] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path Splicing. In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.
- [21] OpenFlow Switch Consortium. <http://www.openflowswitch.org/>, 2008.
- [22] Open Network Laboratory (ONL). <http://onl.wustl.edu/>, 2009.
- [23] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proc. 1st USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 135–146, Monterey, CA, Dec. 1997.
- [24] M. Shand and S. Bryant. IP Fast Re-route framework. <http://www3.tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-framework-07>, June 2007.
- [25] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCP’s burstiness with flowlet switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, San Diego, CA, Nov. 2004.
- [26] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.
- [27] J. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, et al. Supercharging PlanetLab: A High Performance, Multi-application, Overlay Network Platform. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.
- [28] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.
- [29] D. Wendlandt, I. Avramopoulos, D. Andersen, and J. Rexford. Don’t Secure Routing Protocols, Secure Data Delivery. In *Proc. 5th ACM Workshop on Hot Topics in Networks (Hotnets-V)*, Irvine, CA, Nov. 2006.
- [30] C. Wiseman et al. A Remotely Accessible Network Processor-Based Router for Network Experimentation. In *ANCS*, 2008.
- [31] W. Xu and J. Rexford. MIRO: Multi-path Interdomain Routing. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.
- [32] X. Yang. NIRA: A New Internet Routing Architecture. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, Karlsruhe, Germany, Aug. 2003.
- [33] X. Yang, D. Wetherall, and T. Anderson. Source selectable path diversity via routing deflections. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.
- [34] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc. USENIX Annual Technical Conference*, pages 99–112, Boston, MA, June 2004.