ROFL: Routing on Flat Labels

Matthew Caesar Tyson Condie Jayanthkumar Kannan
Karthik Lakshminarayanan Ion Stoica Scott Shenker

University of California at Berkeley

{mccaesar,tcondie,kjk,karthik,istoica,shenker}@cs.berkeley.edu

ABSTRACT

It is accepted wisdom that the current Internet architecture conflates network locations and host identities, but there is no agreement on how a future architecture should distinguish the two. One could sidestep this quandary by routing directly on host identities themselves, and eliminating the need for network-layer protocols to include any mention of network location. The key to achieving this is the ability to route on flat labels. In this paper we take an initial stab at this challenge, proposing and analyzing our ROFL routing algorithm. While its scaling and efficiency properties are far from ideal, our results suggest that the idea of routing on flat labels cannot be immediately dismissed.

Categories Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking; C.2.2 [**Computer-Communication Networks**]: Network Protocols – *Routing Protocols*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Algorithms, Design, Experimentation.

Keywords

Routing, naming, Internet architecture

1. INTRODUCTION

For a variety of reasons, including the NewArch project [47], various commentaries [29], NSF's GENI [45] and FIND [44] programs, and pent-up frustration at the current state of affairs, it has become fashionable to consider clean-slate redesigns of the Internet architecture. These discussions address a wide range of issues, and would take the architecture in many different (and sometimes opposing) directions. However, the one point of consensus (among those who comment on the matter) is that any new architecture should cleanly separate location from identity.[1] The current use of IP addresses to signify both the location and the identity of an endpoint is seen as the source of many ills, including the inability to properly incorporate mobility, multihoming, and a more comprehensive notion of identity into the Internet architecture. As long ago as Saltzer's commentary [31] and the GSE proposal [28], and probably even before that, there have been calls for separating the two, either through new addressing schemes (as in GSE), or through more radical architectural changes (e.g., TRIAD [10], IPNL [16], HIP [22], FARA [11], LFN/DOA [3, 38], i3 [33], SNF [23], etc.).

[1] By location we mean a label that enables one to find the object in the network, and by identity we mean a label that uniquely and persistently specifies that object. We will use the terms name and identity interchangeably throughout this paper.

All of these proposals define or assume the existence of a (not necessarily global) endpoint namespace (or namespaces), but they differ greatly in the nature of the namespace, from using FQDNs, to flat names, to allowing any namespace at all (*i.e.,* the architecture is namespace-neutral).

Despite the differences in namespaces and many other factors, there is an underlying similarity in how these proposals *use* endpoint names. Most designs involve *resolution*; that is, at some point in the process, the name gets turned into a location (be it an address or a more general forwarding directive [11]), and the network uses this location information to deliver the packet to the destination. This location information is considered ephemeral, and only the name serves as a long-term identifier. The resolution could be done through DNS, or by the network (as in [33]), or through some other unspecified process.

This paper takes a very different approach. Rather than split identity from location, we get rid of location altogether. That is, we propose that the network layer not contain location information in the packet header; instead, we propose to route directly on the identities themselves.[2] This approach inherits all the advantages of the location-identity split, such as mobility, multihoming, and stable identities, but also has several practical advantages of its own:

- No new infrastructure: There is no need for a separate name resolution system (which already exists for DNS names, but would have to be created for anything other than DNS names).
- Fate-sharing: Packet delivery does not depend on anything off the data path, because there is no need to contact a resolution infrastructure before sending a packet.
- Simpler allocation: Unlike IP addresses, which need to be carefully allocated to ensure both uniqueness and adherence to the network topology, the allocation of identities need only ensure uniqueness.
- More appropriate access controls: Network-level access controls, which are now largely based on IP addresses, can now be applied at a more meaningful level, the identifier.

However, this design isn't motivated solely by these advantages. The real driving force is our wanting to question the implicit assumption, which has been around for as long as the Internet, that

[2] We will return to these papers later when we review related work, but for now we note that TRIAD and IPNL both routed on FQDNs; however, they used resolution to reach objects that are outside of their home realm. The design in [18] does not use resolution, but cannot scale if many objects don't follow the DNS hierarchy. Thus, none of these three designs can scalably route on fully general (and movable) identities.

scalable routing requires structured location information in the packet header. So we now ask: how can you route just on names, and how well can it be done?

First we need to settle what these names look like. If they are to be the cornerstone of the architecture, one would like names to serve as persistent identifiers. As argued in [3, 33, 38], though, persistence can only be achieved if the names are free of any mutable semantics. The easiest way to ensure a name has no mutable semantics is to give the name *no* semantics at all. Thus, in what follows we use a *flat* namespace, where names have no semantic (but perhaps have cryptographic) content (see, *e.g.,* [3, 22, 26, 33, 38]). One can argue for or against the desirability of flat namespaces, and we certainly don't have the space to make a persuasive case here, but not only do we believe they have significant advantages, we also believe that if you route on any form of structured names then you are indeed back in the realm of using structure to scale routing.

The technical challenge, then, is to scalably route on flat labels (we use the term label because from a routing perspective it doesn't matter whether these are names or something else; the goal is to route to wherever that label currently resides). To our knowledge, every practical and scalable routing system depends on the structure of addresses to achieve scalability,[3] so this is a daunting challenge indeed. Our goal isn't to prove that ROFL can match the performance of the current Internet, it is merely to see how far we can get in this direction of the design space.

Our quest is related to the work on *compact routing* (which is essentially how to route on flat labels), which for the Internet context has been most usefully explored in [24, 25]. The focus there was on the asymptotic static properties of various compact routing schemes on Internet-like topologies, but there was no attempt to develop or analyze a dynamic routing protocol that implemented these algorithms. It is precisely that problem, the definition and performance of a practical routing protocol on flat labels, that is our focus here. While ROFL falls far short of the static compact routing performance described in [24, 25] and elsewhere, it seems far better suited for a distributed dynamic implementation.

Roadmap: We start by giving a high level overview of our design in Section 2. We then provide a more detailed description in two parts: *Intradomain* routing (routing within a single ISP, Section 3), and *Interdomain* routing (Internet-scale routing across ISPs, Section 4). We touch on extensions to the basic ROFL design to address related concerns in routing (Section 5) and then discuss simulation results (Section 6). We then conclude in Section 7.

2. OVERVIEW

Before we present our design, we should note the three dimensions along which it should be evaluated.

Architecture: These are the broad issues raised in the previous section about what benefits flow from routing on flat names.

Features: We will show, in the detailed design sections, that ROFL can support policy routing (Section 4) and can be extended to support anycast and multicast (Section 5).

Performance: We will address this through simulation, where we study stretch, join-overhead (which captures the effect of host churn), and failure-recovery. The numbers aren't pretty, but they suggest that with a big enough cache, one that is well within reach of current technology, the performance might be acceptable.

[3]While DHTs might appear to be a counterexample, they run on top of a point-to-point routing system and thus don't truly address the problem of building, from scratch, a system that routes without using structured location information.

We now give a very high-level view of our ROFL design, which borrows heavily from insights and techniques in HIP [22], Chord [34], Canon [17], and Virtual Ring Routing (VRR) [7].

2.1 Preliminaries

Identifiers: We use self-certifying identifiers; that is, we assume a host's or router's identity is tied to a public-private key pair, and its identifier (ID) is a hash of its public key. In general, a physical box can have multiple IDs, and an ID can be held by multiple boxes (which is how we will implement anycast and multicast), but for this simple description we will assume each host and router has a single, globally unique ID. We wrap these values to create a circular namespace and, as in Chord, we use the notions of *successor* and *predecessor* and will establish a ring of pointers that ensures routing is correct; some additional pointers cached along the way will lead to shorter routes. Nodes maintain pointers to both *internal* (within the same AS) and *external* (in a different AS) successors, as shown in Figure 1.

Source routes: As done today, hosts are assigned to a first-hop or gateway router through either DHCP or manual configuration (and, in fact, a host can have several gateway routers). We say that a host's ID is *resident* at this gateway router, so each router maintains a set of resident IDs (in addition to its own ID), and it maintains source routes to their successors on their behalf. We call the router at which an ID is resident the ID's *hosting* router. A source route or path from one ID to another is a hop-by-hop series of physically connected router IDs that goes from one hosting router to another.

Classes of nodes: There are three classes of nodes in the system: routers, stable hosts (*e.g.,* servers and stable desktop machines), and ephemeral hosts (hosts that are intermittently connected at a particular location, either because of mobility, *e.g.,* laptops, or because of frequent shut-downs or failures, *e.g.,* home PCs turned off when not in use). The decision about whether a host is stable or ephemeral is made by the authority who administers the router at which it is resident. When we use the term host without a modifier, we will mean a stable host; ephemeral hosts will be treated as a special case and dealt with later in this section.

Source-Route Failure Detection: In order to detect source route failures, ROFL assumes an underlying OSPF-like protocol that provides a network map (and not routes to hosts) and can identify link failures in the physical network. In the intra-domain case, this protocol finds paths to other hosting routers within the same AS. In the inter-domain case, this protocol maintains routes to external border routers whom the internal hosting routers have pointers to. This protocol can also be used to find the egress router by which an adjoining AS can be reached. This protocol is used to detect link and node failures, and notifies the routing layer of such events.

Security: The self-certifying identifiers can also help fend off attacks against ROFL mechanisms itself. When a host is assigned to a hosting router, before its ID can become resident, the host must prove to the router cryptographically that it holds the appropriate private key. Thus, there can be no spoofing of IDs unless, of course, the router misbehaves. However, end-to-end verification (both from routers and from hosts) can prevent such spoofing even with a misbehaving router. A more subtle attack is the Sybil attack [13], where-in a compromised router may concoct identifiers to gain a larger footprint in the system. Damage control against such attacks may be achieved by auditing mechanisms within an AS that limit the number of IDs hosted by a router.

For ease of exposition, we first describe how ROFL does intra-domain routing, and then go on to the more complicated case of

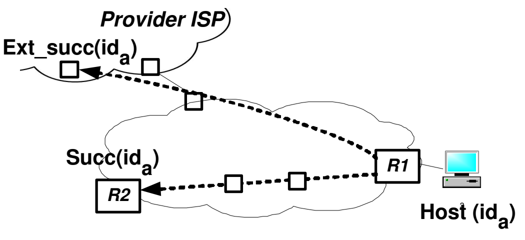


Figure 1: A host with id_a has pointers to an internal successor, Succ(id_a), and an external successor, Ext_succ(id_a).

inter-domain routing. The discussion here is informal (the more detailed and precise explanation is presented in the following sections) and we focus on the steady-state (when no joins or leaves are in progress) for clarity.

2.2 Intradomain

Joining: Whenever a new host a arrives, its hosting router sets up a source route from id_a to its successor ID, and contacts the hosting router for the predecessor ID to have it install a source route from it to id_a. This can be done using Chord-like joining algorithms, which return an ID's predecessor and successor. In steady-state, the set of nodes forms a ring, with each ID having a source-route to its successor and predecessor IDs. The same is true for newly arrived routers, except that they do their own path establishment (routing through one of their physically connected next-hop routers).

Caching: Whenever a source route is established, the routers along the path can cache the route (keeping track of the entire path). Thus, in steady-state each router has a set of *pointers* to various IDs, some emanating from their own resident IDs to successor and predecessor IDs, and others being cached from source routes passing through it. The pointer-cache of routers is limited in size, and precedence is given to pointers in the former class.

Routing: Routing is greedy; a packet destined for an ID is sent in the direction of the pointer that is closest, but not past, the destination ID. This is guaranteed to work in steady state because in the worst case it can always walk along the series of successor pointers.

Recovering: In the case of a router failure, the neighboring routers inspect all their cached pointers and send tear-down messages along any path containing the failed router. In the case of host failure (or, as we will call it, ID failure), the router sends tear-down messages to each of the ID's successors and predecessors. When a tear-down message reaches a hosting router, it rejoins the relevant ID so it can find its current successor/predecessor. To increase resilience to ID failure, nodes can hold multiple successors (*i.e.,* the successor and its successor), and we will call these successor-groups.

Finally, certain sequences of failure events could cause the successor ring to partition into multiple pieces, even if the underlying network is connected. To prevent this, routers continuously distribute routes to a small set of stable identifiers. Routers locally perform a correctness check based on the contents of this set, then execute a partition-repair protocol that ensures network state converges correctly into a single ring. This ensures that if a path exists between hosts a and b, ROFL will ensure a and b can reach each other.

Ephemeral hosts: Ephemeral hosts cannot serve as successor or predecessor to other IDs; they merely establish a path between themselves and their predecessor, which keeps a source-route to the ephemeral hosts; when other nodes route to this ephemeral ID, the packet will travel to the predecessor router, and then be forwarded to the host. Ephemeral hosts (or, rather, their hosting router) can set up these backpointers at other routers for more efficient routing, but state at the predecessor is necessary.

2.3 Interdomain

Our inter-domain design is similar in spirit to our intra-domain design, but it must be modified to abide by AS-level policies. ROFL's interdomain design leverages the fact (see [35, 36]) that most current policies can be modeled as arising out of a simple hierarchical AS graph. For supporting such policies, we extend Canon [17] which, when translated from its original DHT context to our interdomain one, only supports standard customer-provider relationships as they would arise in a tree graph (namely, every AS has a single provider). Our extensions to Canon (for our context) allow ROFL to support most of today's Internet policies — such as customer-provider, multihoming, peering (direct/indirect) — but not all policies implemented today using BGP.

Constructing a global ring: In our design, each AS X runs its own ROFL-ring (RR), RR_X, as specified by our intra-domain design. In order to ensure that hosts within its RR are reachable from other domains, RR_X needs to be merged with the RRs of other domains. This is done in three phases. First, AS X discovers its *up-hierarchy* graph G_X, which consists of all ASes "above" X in the AS hierarchy (X's providers, its providers' providers, and so on). Edges in G_X correspond to X's view of the customer-provider, multihoming, and peering relationships in X's up-hierarchy. G_X does not need to be complete: providers of AS X may choose not to reveal certain links to X, or X may decide to prune G_X to reduce its join and maintenance overhead (which is roughly linear in the number of edges in this graph).

Next, X performs a Canon-style [17] recursive merging protocol (Section 4.1) that constructs additional successors to RRs in other ASes. This is done by merging X's RR with all the RRs in the domains at or below X in the AS graph. This is done in a manner that respects certain interdomain policies. Moreover, the merging process provides a useful *isolation* property: when a host in domain X sends a packet to a host in domain Y, the data path is guaranteed to stay within the subtree rooted at the earliest common ancestor of these two domains. As a corollary, traffic internal to an AS stays internal.

In addition to using successor pointers, our inter-domain design also uses proximity-based routing tables to reduce stretch. These are routing tables that allow fast progress in the ID-space, and are similar to Pastry routing tables: the main difference is that a routing table entry for an ID in AS X points to the node with the appropriate prefix which resides in the *lower-most* level of the hierarchy (relative to X). This ensures that following routing tables does not violate the isolation property.

Joining: Whenever a host with id_a comes up in AS X, and wishes to be globally reachable, its hosting router is responsible for finding a successor and predecessor *at each level* of the G_X sub-hierarchy. This can be done by looking up the predecessor and successor of id_a at each level of the AS hierarchy. The hosting router then associates the successor and predecessor pointers for id_a with an AS-level source-route to the routers hosting the predecessor and successor identifiers for id_a. This can be any source route consistent with the graph G_X, and there can be multiple source routes for resilience to failure. These AS-level routes are used in determining which of these pointers are available for relaying a packet (in a process quite similar to how BGP determines the links to forward a route advertisement). To reduce stretch, the hosting router uses a similar procedure to discover fingers at each level. Border routers in an AS may optionally maintain bloom filters that summarize the set of hosts in the subtree rooted at the AS. These bloom filters are also updated during the join process.

Routing: Our mechanism for routing relies on greedy routing, augmented with in-packet AS-level source-routes. As a packet is routed towards its destination, it is marked with an AS-level source route denoting the path traversed until that point. When a router receives a packet, it uses the source-route in determining the candidate set of outgoing pointers can be used in forwarding the packet; that is, it finds the paths that are consistent with policy. This decision is made by comparing the source-route on the packet to the source-routes on the pointers using BGP-like import and export filtering rules. Then, greedy routing is used to determine the closest candidate pointer, whose source-route is tacked on to the packet. Note that the salubrious properties of greedy routing (such as loop-free forwarding, eventual reachability) apply even when the packet is forwarded in this fashion.

Recovering: In the case of a router failure, routers with pointers to the failed router are notified either pro-actively by neighbors of the failed router, or discover the failure when forwarding a packet. In the case of host failure, the router sends tear-down messages to each of the ID's successors and predecessors. When a host/router failure is noticed by a router which has pointers to the ID, it rejoins the relevant ID by finding successors/predecessors at the relevant level.

In the case of AS-level link failures that lead to a partition in G, the isolation property ensures that hosts in ASes X and Y can route to one another provided there is a subtree in $G_X \cup G_Y$ such that all AS-level links in the subtree are functional. Hence in the common case where one access link of a multi-homed AS goes down, incoming and outgoing traffic will be automatically shifted to the other access links. Note however that in some failure patterns, there is a *path* in the Internet graph between ASes X, Y, but no fully functional subtree in $G_X \cup G_Y$. In this case, AS X can either prune the graph G_X to only working links, and redetermine the successors of its IDs over this graph; or, it can add working links to G_X to ensure that such a working subtree exists, and re-join its IDs over those links.

Handling Policies: Our design also handles peering and multi-homing relationships between ASes. We treat multi-homing links as backup links; an AS joins the global ROFL ring through one of its providers, and uses the other providers as backup, in case the primary provider fails.

Peering relationships can be handled in our design in two different ways. One design option is to transform the graph G so that doing greedy routing over the links established via joins in G suffices to handle peering. In this case, the property we provide is that, if a customer of provider X routes to a customer of a peer AS Y of X, it is guaranteed to use the peering link for that purpose. However, the limitation here is that the peering link may also be used in routing packets destined to customers not belonging to Y; such packets will be simply returned via the peering link, and will be routed via X's provider. This is necessary since it is not possible to determine whether the destination is a customer of Y without doing a complete search of the customers of Y. Our second design option is to use bloom filters. In this method, AS X uses the bloom filters of its peers to determine if the destination is possibly a customer of any of its peers. If so, it uses the peering link to forward the packet to Y, which uses its pointer to route to the destination. Note that in order to handle false positives in the bloom filter, this method may require back-tracking, in case the destination is discovered to not be in Y.

We note that our design requires ISPs to reveal customer-provider, multi-homing, and peering relationships to their downstream customers (since a downstream-customer X uses them to compute G_X). This may not be a serious concern, since as shown in [35], such relationships are mostly inferable in BGP today. Finally, our design allows multi-homed ASes some degree of control over incoming traffic on their access links, though we are yet to fully understand how this degree of freedom compares to that permitted (or forbidden) by BGP. This control in ROFL is achieved by investing the join process and identifiers with some traffic-engineering semantics (described in Section 5).

3. INTRADOMAIN

3.1 Host Join

Algorithm 1 The join_internal(**id**) function is executed by a router upon receipt of a host request for joining the network. The function bootstraps a virtual node on behalf of the host.

```
1: authenticate(id) # exception on error
2: $vn$ = new VirtualNode(id)
3: register_virtual_node($vn$)
4: pred = find_predecessor(id)
5: # Setup state with local participants
6: $vn.successor_{internal} = pred.successor_{internal}$
7: $pred.successor_{internal} = vn$
8: S = select_providers()
9: for all $s \in S$ do
10:     br = locate_border_router(s)
11:     p = get_path_to_root(s)
12:     br.join_external($vn$, p)
13: end for
```

The joining host with id_a first selects an upstream gateway router to join the ring on its behalf. It opens a session to the router and calls *join_internal* (Algorithm 1), which performs the bootstrap process. The router authenticates the host and spawns a virtual node $vn(id_a)$ that will hold the routing state with respect to this host's identifier. The router then joins the internal ring by using the host's identifier to locate the predecessor in the internal AS. The predecessor is used to initialize the internal successor state in $vn(id_a)$. The router then discovers the external successor state by first determining the set of paths along the up-hierarchy on which to join. This set of paths is selected in a manner obeying the policies of the joining host and its internal AS. For each of these paths, the router then selects a border router connected to the next AS-hop along the path. The router forwards the join request to this router, which in turns performs an external join using the *join_external* function (described in Section 4.1).

However, this procedure does not work if id_a is the router R's first resident ID, since R does not have any pointers and hence cannot make progress in the ring. To deal with this, when R first starts up it creates a *default virtual node*. The default virtual node's ID is the router-id, and its successors act as default routes if it has no other successors that it can use to make progress. The default virtual node joins by flooding a message containing the router-ID. The router-ID's predecessors add a pointer to the router-ID, and its successors respond back via the path contained in the message. This ensures that all resident IDs find a predecessor in the internal AS when joining.

When forwarding a control message, intermediate routers may cache destination IDs contained in the message if they have spare memory. The control messages also build up a list of routers along the way, and this list is stored by the router hosting the destination ID. This list is used to maintain consistency in the presence of host failure, as described below.

3.2 Failure

We aim to maintain routing state so as to preserve two invariants: (a) if there is a working network-path between a pair of nodes (A, B), then ROFL ensures that A and B are reachable from each other (b) if A has a pointer to B, and if either B or the path to B fails, then A will delete its pointer. We describe how this is achieved below.

Router failure: If a router R hosting several IDs goes down, there are two things that need to happen. (1) Each host connected to the router R discovers the outage (via a session timeout) and needs to rejoin via an alternate router. Alternatively it can do this proactively by joining via multiple routers during its initial join. (2) There are a set of virtual nodes residing at other routers with pointers to IDs at R that need to be updated. Although we could simply rejoin each virtual node affected by the failure, we instead improve performance by having routers in advance agree on a sorted list of routers that will be failed over to in event of failure. Upon node failure, the end host and remote routers deterministically fail over to the next alive router on the list.

Host failure: When host with ID id_a fails, the gateway router R will detect the failure through a session timeout. R needs to inform all other routers with pointers to id_a that it has failed. One simple way to do this would be to flood all routers with an invalidation message. However, flooding the entire system on host failure would not be efficient. Instead, we address this by constraining the set of routers in the system that are allowed to maintain cached state for id_a. For simplicity we constrain this set to be routers holding predecessors of id_a and routers that lie on the shortest path to those routers. When there is a host failure, the router sends a *directed flood*, i.e. a source-routed flood that traverse only this subset of routers. When shortest paths change, or links fail, routers can optionally update this set via additional directed floods, however this is an optimization that is not necessary for correctness. As a fallback to handle router failure, routers also monitor link-state advertisements and delete pointers to IDs residing at unreachable routers.

Link failure, no partition: If the set of links that fail do not create a partition, then the router need not make any changes on behalf of its resident IDs since the network map will find alternate paths to their successors. However, the contents of pointer caches that traverse the link should be temporarily invalidated while the link is failed (to avoid sending packets over the failed link).

Link failure, partition: In the event of a network-layer partition, the successor pointers maintained by routers need to remerge into two separate, consistent namespaces. First, invalid pointers (pointers that terminate at routers that are no longer reachable) are torn down. Next, the router attempts to repair these pointers locally by shifting the successors down to fill the empty space left by each failed successor (since it knows no closer IDs may exist in the network), then it tries asking each of its successors S_i starting at the one furthest away to fill the gap at the end of its successor list. Unfortunately this process could cause the ring to partition into multiple pieces, even if the underlying network is connected. To recover from this, we require routers to distribute the smallest ID they know about (the *zero-ID*, i.e. the ID closest to zero) to all its neighbors. The zero-ID a router propagates is set equal to the minimum of the smallest ID it is hosting and the smallest ID it receives from its neighbors (the path is also distributed to avoid circular dependencies and allow all nodes to reach the zero node). The end result is that all routers become aware of the smallest ID in the network. This ensures multiple partitions will heal if the network layer is connected: if the zero-ID is on one ring, its predecessor on the other ring will learn about it and add it, triggering a merging process. The zero-ID will repair its successor and predecessor, who in turn repair their successors, and so on until the rings are merged. In practice, the zero node advertisements are piggybacked on link-state advertisements, and we use the router-IDs of routers instead of the zero-ID to reduce sensitivity to churn and balance load over several routers during the recovery phase.

3.3 Packet forwarding

When a router forwards a packet, it selects the closest ID it knows about to the destination ID. This is done using the link-state database to return the next hop towards the router containing that ID. This approach requires routers to return the closest entry in the namespace as opposed to the shortest-prefix match lookups commonly done today. Finding the closest entry can be implemented with minor modifications to routers that support longest-prefix match. The key observation is that, given a list of IDs in sorted order, the closest namespace distance match is either the shortest prefix match or the one right before it in the sorted list.

Algorithm 2 The route (**pkt**) function is executed by a internal router upon receipt of a packet destined for a particular virtual node.

```
1: $next\_hop_{vn}$ = VN.best_match(pkt.destination.id)
2: if $pkt.destination.id == next\_hop_{vn}.id$ then
3:     deliver_to_host($next\_hop_{vn}$, pkt)
4: else
5:     $next\_hop_c$ = PC.best_match(pkt.destination.id)
6:     if $next\_hop_{vn}.id < next\_hop_c.id$ then
7:         sendto($next\_hop_c$.path_to_router, pkt)
8:     else
9:         sendto($next\_hop_{vn}$.path_to_router, pkt)
10:    end if
11: end if
```

The forwarding algorithm is shown in Algorithm 2. The router maintains a list of resident virtual nodes (VN), which exports a best_match function that determines the next_hop by choosing the closest ID among all resident IDs and their successors that does not overshoot the destination. If the destination is an attached host, *VN.best_match* returns the interface for the host, which the router uses to deliver the packet. Otherwise, $next\_hop_{vn}$ is set to the successor state of some resident virtual node. Before forwarding the packet, the router first checks its pointer cache (PC) for an entry that is closer to the destination than the value stored in $next\_hop_{vn}$. If such a cached entry exists, the router uses its value, stored in $next\_hop_c$, instead.

4. INTERDOMAIN

In this section we describe our design for interdomain ROFL (which borrows heavily from Canon [17]). First, we give an overview of how the basic protocol works. Next we provide more details regarding how hosts join, how packets are routed, and how failures are handled. Then we describe how customer-provider, peering, and multihoming policies are supported by our augmented greedy routing protocol over a suitably defined Directed Acyclic Graph (DAG).

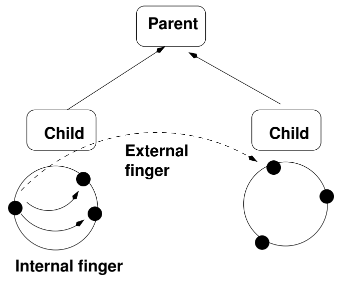


Figure 2: Merging rings

4.1 Basic design

Interdomain ROFL constructs a DHT over a hierarchical graph, where nodes correspond to ASes and links correspond to inter-AS adjacencies. Within each AS, the identifiers form an internal ring as described in Section 3. These rings are then merged with one another in a bottom-up fashion (traversing up towards the root of the AS-hierarchy) by having virtual nodes maintain routes to *external* successors that reside in other ASes, as shown in Figure 2. For a identifier id_a in ring 1, these external pointers are established to identifiers id_b in ring 2 that satisfy two conditions: (a) id_b would be id_a's successor if the two rings were merged into a single ring, and (b) there are no identifiers in either AS within the interval $[id_a, id_b]$. This approach is repeated for each level in the hierarchy. Condition (b) thus limits the number of external pointers that are formed. Prior work has shown that the expected total number of pointers (both internal and external) is $O(\log(n))$ (where n is the total number of identifiers across all stub domains) [17].

Routing occurs as in Chord. Note that on a single customer-provider hierarchy, a packet sent between a pair of ASes will traverse no higher than their least-common ancestor in that hierarchy. Moreover, if a host within an AS sends a packet to another host in that same AS, no external pointers will be used. We refer to this as the *isolation property*.

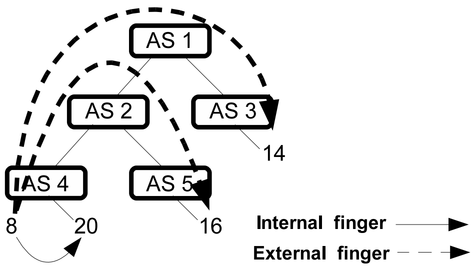


Figure 3: Routing state for virtual node with identifier 8.

For example, Figure 3 shows the internal and external routing state for a router hosting an identifier 8 residing in AS 4. The hosting router has an internal successor pointer to the router hosting identifier 20 and external successors to hosting routers residing in ASes 5 and 3. The join protocol discovers the external successor at each level of the joining node's up-hierarchy. For instance, the hosting router for 8 maintains a external successor to 16 at the level of AS 2, and an external successor to 14 at the level of AS 1.

Joining: When a hosting router R performs a join for an end-host with ID id_a, R joins both the internal ring (as described in Section 3) and also the ROFL ring on behalf of id_a. id_a joins the ROFL ring by, for each AS X in its up-hierarchy, routing towards its successor using links that traverse no higher than X. In this fashion, it builds a list of candidate successors, one corresponding to each AS in its up-hierarchy. It then removes unnecessary successors. For example in Figure 3, if the identifier in AS 5 were 12 instead of 16, 8 would not maintain 14 as a successor (as doing so could violate isolation). Finally, if id_a is the first host in the ISP, it needs a way to bootstrap itself into the ROFL ring. This is done by having host identifiers register with their providers (and their provider's providers, and so on) when they join. Their providers need only maintain a short list of such identifiers (a few at each level of the hierarchy for resiliency purposes). When a new host joins that does not have a predecessor in its internal ring, the ISP will forward the join request to one of its providers to lookup a bootstrap node. The registration process also allows operators to control which set of ASes id_a can join through, and to constrain connectivity to follow policy or traffic engineering goals.

Algorithm 3 The join_external (**vn, p**) function is executed by a border router upon receipt of a request for a joining virtual node **vn** along the path **p**.

```
1: pred = find_predecessor(vn.id)
2: $RS_{pred} = pred.successor_{external} \cup pred.successor_{internal}$
3: $RS_{vn} = vn.successor_{external} \cup vn.successor_{internal}$
4: prune_route_entries($RS_{pred}$, p)
5: prune_route_entries($RS_{vn}$, p)
6: if $min\_id(RS_{pred}) < min\_id(RS_{vn})$ then
7:     $vn.successor_{external}.add(min\_id(RS_{pred}))$
8: end if
9: if $vn.id < min\_id(RS_{pred})$ then
10:    $pred.successor_{external}.add(vn)$
11: end if
12: br = next_border_router(p)
13: if $br! = NULL$ then
14:    br.join_external(vn, p)
15: end if
```

The join_external function (Algorithm 3) shows this process in more detail. First, the external successor at a level is discovered by routing towards the external predecessor at that level and then pruning away any references to virtual nodes outside the current hierarchy in both the predecessor's and the virtual node's routing state. After pruning, the virtual node with the minimum identifier in the predecessor's routing state is kept if it is a better external successor than the virtual node's current set of successors. The next step of the algorithm tests if the virtual node itself is a better external successor to the predecessor, and if so adds it to the predecessor's routing state. The final step uses the path vector passed in as the argument to recursively call this same function at the border router of the next provider. This recursive call terminates at the root of the hierarchy.

Exploiting network proximity: ROFL exploits network proximity to reduce routing stretch by maintaining *proximity-based fingers* in addition to successor pointers. That is, when selecting fingers at each level of the hierarchy, ROFL tries to select fingers that are nearby in the physical network. This reduces the number of network level hops required to make a given amount of progress in the namespace.

We store these fingers in a prefix-based finger table (along the lines of Bamboo/Pastry/Tapestry), where each row corresponds to a given prefix-length and each column corresponds to a digit at that prefix. Each entry contains an ID that is reachable via the smallest number of up-links. In other words, an entry K may be inserted in the element (i, j) in J's finger table iff (a) K matches i bits of J's ID and K's $[i, i + b]$ bits are equal to digit j (b) of all joined IDs L matching the position (i, j), it is not the case that the path from J to L contains fewer up-links than the path J to K. If this table is correctly maintained, the isolation property is preserved. To exploit proximity, entries that are reachable via fewer AS-level hops are preferred. For correctness purposes, each ID also maintains a list of IDs that are pointing to it.

Our joining and maintenance protocols for these fingers are adapted from the proximity extensions in [9] to support the policies and properties described in Section 4.2. The join consists of three phases. First, the joining host sends a *join request* towards its own ID. At each network-level hop n, n attempts to insert entries from its own finger table into the message. The message is then returned back to J after it reaches J's predecessor. At this point, J's entries are correct. Next, J may need to be inserted into the finger tables of other IDs. This is done by having virtual nodes maintain copies of their finger's finger tables. In particular, we modify the join to also record a list of IDs that need to insert J. J then sends a multicast message containing its ID to every virtual node in this list. Upon receipt of this message, virtual nodes check to see if any of their fingers need to insert J, and if so update their neighbors, and so on. Nodes also piggyback probes on data packets to ensure this state is maintained correctly (note if this state becomes inconsistent, isolation may be violated, but we will still reach the correct final destination).

We now describe several other detailed issues:

Failure recovery: The isolation property ensures that failures and instability outside of a particular hierarchy will not influence routing within the hierarchy. Because of this, link failures that cause partitions (the inability to reach successors via a certain level of the hierarchy) are not reacted to immediately, as ROFL ensures that alternate paths are available. Also, an ISP may host virtual servers on behalf of a customer ISP, which it can maintain during that customer's outages. Finally, in the event of long-term failures, we need to ensure that the ring converges consistently at each level of the hierarchy. We do this using a similar approach to that given in Section 3.2. In particular, each AS maintains a route to the zero-ID (the ID closest to zero) in their down-hierarchy. Hosts then merge changes to the zero-ID to ensure partitions and other anomalous conditions (e.g. loopy cycles) heal properly.

Integrating EGP and IGP routing: Today's Internet uses iBGP to redistribute externally learned routes internally. In our architecture, we have a similar need for a protocol to do this redistribution. As mentioned in previous sections, packets contain a list of ISPs that can be used to reach the final destination. Hence a router containing a packet needs to know how to reach the next-hop AS in the list. To solve this problem, we have border routers flood their existence internally. We believe doing this does not significantly impact performance since even the largest ISPs typically only have a few hundred border routers. Moreover, these advertisements can be aggregated if ISPs wish to treat two routes to the same next-hop ISP through different border routers as being equal.

Exploiting reference locality: ROFL exploits locality by using *pointer-caches* [7]. Routers maintain caches in fast memory which contain frequently accessed routes. When routing a packet, the router checks its pointer cache, and shortcuts if it observes a cached pointer is numerically closer to the final destination. However, naive pointer-caching violates the isolation property, as an AS may select a pointer from its cache that traverses its provider. Hence ASes that cache pointers maintain bloom filters containing the set of hosts joined below that AS. When receiving a packet destined to identifier id_b, the border router consults the bloom filter to see if identifier id_b is below it in the hierarchy. If not, the router is free to use its pointer-cache to find a closer next-hop ID. The source-route on the packet is used to determine which pointer-cache entry to use based on policy. Note that the use of bloom filters guarantees the isolation property in the presence of caching. Further, the size of bloom filters can be traded off against the false positive rate. Finally, the decision of whether to use pointer caches can be made by each ISP in isolation. Unless otherwise mentioned, in our simulations we assume no ISPs use interdomain pointer caches or their associated bloom filters.

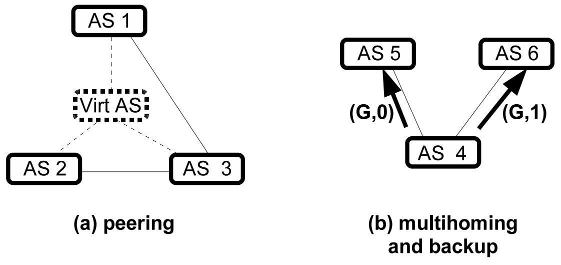


Figure 4: Conversion rules for (a) peering (b) multihoming and backup.

4.2 Handling policies

We aim to support four common types of inter-ISP relationships arising from the Internet's hierarchical structure: *provider-customer* links where a customer ISP pays a provider to forward its traffic, *peering* links where two ISPs forward each other's traffic typically without exchanging payment, *backup* links where an ISP forwards to its neighbor only if there is a failure along its primary link, and *multihomed* connections, where an ISP may have several outgoing links. We extend Canon [17] to support policies using two *conversion rules* (Figure 4) that conceptually convert the AS hierarchy into a Canon-style hierarchy (these rules do not actually modify ISP relationships, but rather are implemented as modifications to the Canon join).

Handling peering: As previously mentioned, we can handle a peering relationship in two ways. In the first option, we modify the AS relationship graph to include *virtual ASes*. A virtual AS is a construct that allows ROFL to discover successors reachable via peering links (it is not explicitly maintained as additional state, but is implemented as an additional set of join rules). An example is shown in Figure 4a. For each peering link, a virtual AS is constructed that acts as a provider for the ASes on either side of the link, and as a customer of each AS's provider. When virtual nodes join, they treat links to virtual ASes as multihomed links, and join them as they would a provider. In this fashion, a host in AS 2 will discover its successors in AS 3, however Canon will ensure that its join will not traverse AS 1 (because relaying between providers is prevented as described below). Note that if several ASes are all peered together in a clique (e.g. the Tier 1 ISPs), we only need a single virtual AS rather than a separate virtual AS for each link.

In the second option, we use bloom filters to deduce when a packet should be allowed to traverse a peering link. When the packet is being routed via an AS (using successor pointers or routing table entries), the AS can check the bloom filters corresponding to its peers to determine if the destination is a customer of any of them. If so, the packet is routed over the peering link, and a bit set to indicate that it has traversed a peering link. In this mode, the packet is not allowed to go up the hierarchy (this ensures that an AS would not use its provider to route packets for its peer). If the destination is not found in the down hierarchy, then it is returned over the peering link, at which point, the packet continues on its original path.

These two options have complementary advantages and disadvantages. The virtual AS option has the disadvantage of increasing join overhead (due to joins corresponding to the peering links), but it makes the data plane protocol simpler. The bloom filter option has the disadvantage of requiring a complicated backtracking protocol, but requires no joins over peering links. For this reason, we describe simulation results comparing both of these design options.

Handling multihoming: A multihomed ISP purchases connectivity from more than one provider and typically has policies indicating how each access link is to be used. There are three kinds of multihomed connections: *single-address* multihoming, where an ISP has a single block of addresses but is connected to multiple providers, *multi-address* multihoming, where an ISP has a separate block of addresses corresponding to each multihomed connection, and *single-neighbor* multihoming, where an ISP is connected with a neighboring ISP via multiple links. Multi-address multihoming is handled by joining each ID via a different provider, and single-neighbor multihoming is handled by applying policy to select which link to use to reach the neighbor. Single-address multihoming is done by repeating the Canon join for each member of the AS's *up-hierarchy*. The up-hierarchy for an AS consists of its providers, their providers, and so on up to the Tier-1 ISPs, plus ASes reachable across peering links (although repeating the join increases overhead, the up-hierarchy above a node is typically small [41], and we can eliminate redundant lookups that terminate at the same successor at multiple levels). Finally, backup relationships are supported by directing join requests only over non-backup links.

5. ADDITIONAL ROUTING ISSUES

We now describe preliminary extensions to the ROFL design to (a) support more flexible routing policies and traffic engineering (b) provide improved delivery models such as anycast, multicast (c) deal with security concerns, specifically, denial-of-service attacks. The last two concerns are meant to be illustrative examples that suggest how the clean-slate design of ROFL may provide better-than-IP routing and security properties.

5.1 Routing Control

BGP and OSPF, two commonly used routing protocols today, allow the operator extremely flexible policy and traffic engineering knobs. We discuss the flexibility of ROFL on these metrics.

Inter-domain routing control: The extensions developed for Canon support customer-provider, backup, and peering relationships. Although these paths may suffice for most traffic, custom paths that satisfy high-level policy goals, stronger QoS constraints, or multipath connectivity may be desired. We propose to handle other policies and route selection mechanisms via two complementary approaches.

We propose the use of *endpoint-based negotiation* where we allow the source and destination nodes to negotiate the path (or set of paths) to be used. Here, we leverage a particular observation about the Internet hierarchy: all paths that can be used to reach AS X from AS Y traverse ASes in the intersection of X's and Y's up-hierarchies. Moreover, up-hierarchies are typically fairly small and can be represented in just a few hundred bytes. Hence when sending the first packet in a session, we allow the source and destination to negotiate a subset of ASes in this set that can be used to forward packets between the two. This is done by having the destination select a subset of ASes above it in the hierarchy and appending this set to the response.

Next, when a hosting router in a multihomed AS performs a join, it sends a join out on each of its AS's p providers with IDs with variable suffixes (G, x_k) $(1 \leq k \leq p)$. Hosts then route packets to (G, r) where r is a randomly chosen suffix. Hosts or intermediate routers may vary r and the suffixes x_k to control the path selected for forwarding packets.

Intra-domain routing control: We can leverage our interdomain design to deal with certain intradomain policies. For example, a transit AS that is spread over multiple countries can create sub-rings corresponding to each of those regions. The isolation property ensures that internal traffic will not transit costly inter-country links. Further, our inter-domain traffic engineering mechanisms may also be used in this context to perform traffic-engineering between these regions.

5.2 Enhanced Delivery Services

There are a vast number of both overlay and network-level proposals for multicast and anycast, many of which can run directly on top of the ROFL design. A few representative (but grossly incomplete) list of examples includes IP Multicast [12], Overcast [21], PIAS [5], and $i3$ [33]. However, traditional overlay-based approaches don't exploit the network layer to improve efficiency, and current network-level designs don't directly scale to or exploit the properties of flat-ID based routing. In this section, we describe some simple extensions to previous approaches that enable anycast and multicast.

Anycast: Anycast is an extension of ROFL's multihoming design. Servers belonging to group G join with ID (G, x). A host may then route to (G, y), where y is set arbitrarily. Intermediate routers forward the packet towards G, treating all suffixes equally. This results in the packet reaching the first server in G for which the packet encounters a route. This style of anycast can be extended to perform more advanced functions (e.g. load balancing) by modifying X, Y and the size of G in a manner similar to the approach taken in $i3$ [33]. This approach to anycast requires no additional state or control message overhead beyond that of joining the network.

Multicast: A host wishing to join the multicast group G sends an anycast request towards a nearby member of G. At each hop, the message adds a pointer corresponding to the group pointing back along the reverse path, in a manner similar to path-painting [20]. If the message intersects a router that is already part of the group, the packet does not traverse any further. The end result is a tree composed of bidirectional links. A host wishing to multicast a packet P forwards the packet along this tree. Routers forward a copy of P out all outgoing links for which there are pointers, excluding the link on which P was received. In the case of single-source multicast, a more efficient tree can be constructed by having nodes route towards the source.

5.3 Security

ROFL identifiers allow us to leverage existing filtering and capability mechanisms and provide stronger guarantees than possible in the Internet today.

Default off: It has been proposed (for example, in [4, 19]) that in the face of mounting security concerns, hosts should not by default be reachable from other hosts. Our architecture eases this by ensuring hosts are only reachable from their fingers. The host (or its upstream router on its behalf) can control pointer construction to limit which other hosts are allowed to reach it. In addition, we require that hosts explicitly register with their providers and traffic to a host not registered with its provider be dropped. In the worst case this traffic can be dropped at the provider of the destination AS, however the use of flat names can potentially allow this traffic to be dropped even earlier. Filtering mechanisms can also be implemented more securely by verifying that the request for installing a filter dropping traffic to an identifier comes from the host owning that identifier.

Capabilities: The use of flat identifiers allows more fine-grained access control through the use of *capabilities* (similar to TVA [42]). When a destination receives a route setup request, it grants access according to its own policies. If permission is granted, the path information and capability are returned to the source, which it uses to communicate further with the destination. This permission is cryptographically secured by the self-certifying identifier of the receiver. A capability [42] is a cryptographic token designating that a particular source (with its own unique object identifier) is allowed to contact the destination. Only with a proper capability will the data plane forward the data packets. Capabilities are associated with a lifetime to defend against sources that attempt to abuse the capability and commit a DoS attack against the destination. We also support the use of path capabilities to further restrict communication along the AS-level path(s) to a destination. Path restriction allows for fine grain pushback mechanisms and hinders the ability to conduct DDoS attacks.

6. EVALUATION

6.1 Methodology

Realistically simulating the Internet is itself a highly challenging problem, both due to scaling issues and because certain aspects of the Internet (e.g. ISP policies) are difficult to infer. We conducted some highly simplified simulations to make the evaluation tractable, yet as much as possible attempted to use real-word measurements for topologies and parameter settings.

Intradomain: The topologies we used were collected from Rocketfuel [32], over 4 large ISPs: AS 1221 (318 routers, 2.6 million hosts), AS 1239 (604 routers, 10 million hosts), AS 3257 (240 routers, 0.5 million hosts), AS 3967 (201 routers, 2.1 million hosts). The number of hosts in each of these ISPs were estimated using CAIDA *skitter* [43] traces. We do this by correlating the IP addresses found in the traces with Routeviews [48] routing tables to map IP addresses onto ASes. We then normalize by the number of estimated hosts in the Internet, which we assume to be 600 million hosts (one study [46] estimates 354 million as of July 2005) to estimate the number of hosts per AS. Each host is assigned a 128-bit ID. Transit routers are presumed to have 9Mbits of fast memory (e.g. TCAM) that can be devoted to intradomain forwarding state. In these experiments we fill pointer caches only with contents available from control packets (we do not snoop on data packet headers for filling caches). We occasionally point out the overheads associated with CMU-ETHERNET [27], an alternate approach to a similar problem. We acknowledge the authors of [27] were attempting to provide a simplified first-cut solution to this problem rather than to achieve this level of scalability, so we reference their work only as a baseline comparison point.

Interdomain: We use the complete inter-AS topology graph sampled from Routeviews. The AS hierarchy inference tool developed by Subramanian et al [35] was used to infer customer/provider relationships and skitter traces were used to estimate the number of hosts per ISP. Due to the limitations of our simulation approach and our goals for preserving certain aspects of realism, our simulations were not able to scale up to 600 million hosts. Instead, we ran simulations for smaller numbers of hosts (up to thirty thousand) and present scaling trends from our evaluation. For simplicity and lack of sufficiently fine-grained measurements, we model each AS as a single node, and start nodes up one at a time (in random order). Unless otherwise mentioned, the results shown do not use the bloom filter or finger caching optimizations.

Metrics: We evaluate the *join overhead*, which corresponds to the number of network-level messages required to add a host to the network, the *stretch*, or the ratio between the traversed path and the shortest path. For Interdomain, we consider stretch to be the ratio of the traversed path to the path BGP would select.

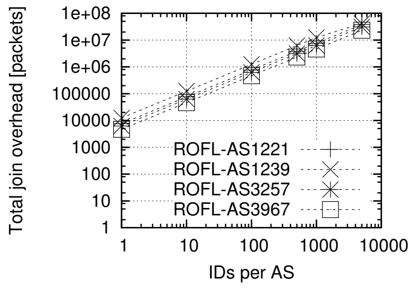
6.2 Intradomain

Host joins: Figure 5a shows the number of messages required to join a given number of hosts, while Figure 5b shows a CDF of the per-host join overhead. Like CMU-ETHERNET (not shown due to lack of space), ROFL scales linearly in the number of hosts. However, CMU-ETHERNET requires between 37 and 181 times more messages to build the network. ROFL's join overhead is roughly four messages times the diameter of the network since only successors need to be notified on join of a new host. Moreover, ROFL gives the operator control over the number of messages generated for host joins. For example, ephemeral hosts can join with a smaller number of successor pointers, and routers can keep successor groups active while host-sessions fluctuate. Figure 5c shows a CDF of the amount of time required to complete a join. This amount of time is typically on the order of the network diameter, because several messages in the join are sent in parallel. In practice, join overhead may be reduced further by ephemeral joins and having the router maintain the virtual node when the host fails or moves temporarily to another AS. Finally, we note this join overhead is a one-time cost in the absence of churn.

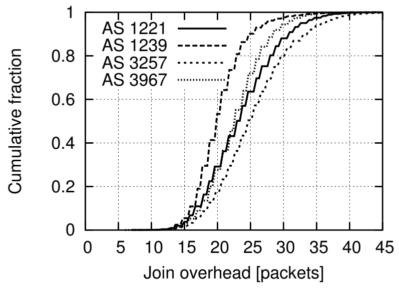
Stretch: Figure 6a plots stretch, measured by routing packets between random sources and destinations, as a function of the size of the pointer cache. Although stretch with small pointer caches can be high, with roughly 70,000 entries (corresponding to a 9Mbit cache of 128-bit IDs) the stretch drops to roughly 2. By comparison a DNS lookup suffers a round trip to the DNS server before sending which could incur a stretch of up to 3. Figure 6b shows the fraction of packets that traverse a particular router. The x-axis corresponds to the rank of the router in a list sorted by the y-value for OSPF. That is, for a particular x value, we plot the load at the ith most congested router in an OSPF network, and the load under ROFL for that same router. We can see that although load varies across routers, the difference from OSPF is fairly slight, indicating that ROFL does not introduce a significant increase in the number of "hot-spots."

Memory requirements: The intradomain pointer-cache memory requirements of ROFL is shown in Figure 6c. By comparison CMU-ETHERNET requires from 34 to 1200 times more memory than ROFL. ROFL's memory requirements were reduced further for routers near the network edge, potentially allowing non-core routers (e.g. customer routers in access networks) to have smaller TCAMs or to cache popular destinations and additional successors. In addition, hosting routers must store state for resident IDs, which requires between 1.3 Mbits for AS 3257 to 10.5 Mbits for AS 1239 assuming IDs are hosted at the Rocketfuel-visible transit routers.

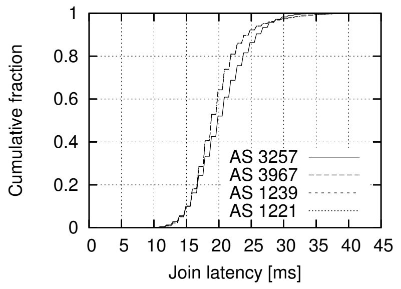
Failure: Here we discuss the overhead and time to reconverge in the presence of network level events. We found the overhead triggered by host failure and mobility to be comparable to join overhead, and link/router failures that do not trigger partitions to be comparable to OSPF recovery times. However if a network-layer partition occurs the ring needs to reconverge into two separate, consistent namespaces. We believe partition events in ISPs are rare in comparison to host failures given the high degree of engineering and redundancy in these networks. Nevertheless, we investigate this overhead to show performance under such extreme scenarios. Figure 7 shows the overhead to recover from a partition. We create



(a)

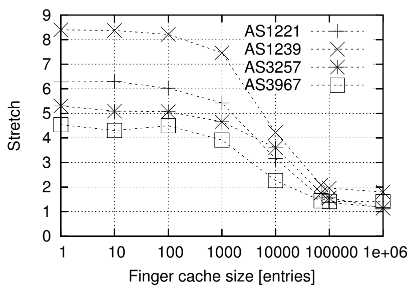


(b)

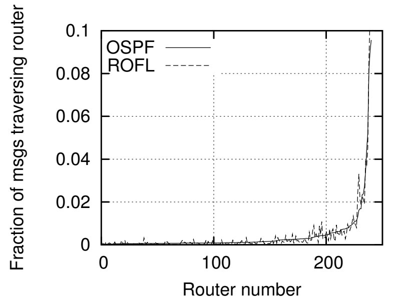


(c)

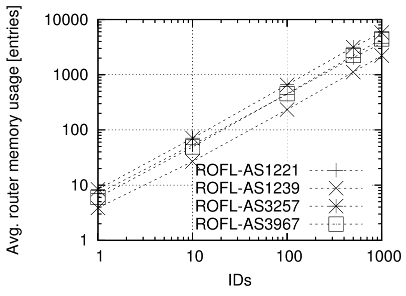
Figure 5: ***Intradomain routing, joining:*** **(a) Cumulative overhead to construct the network (b) CDF of overhead per node join (c) Join latency**



(a)



(b)



(c)

Figure 6: ***Intradomain routing, data traffic performance:*** **(a) Effect of pointer cache size on stretch (b) Load balance, compared with shortest-path routing (OSPF) (c) Memory used per router**

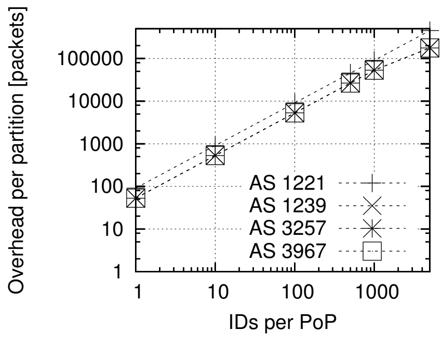


Figure 7: Convergence overhead from Point of Presence (PoP) failures

partitions by varying the number of IDs per PoP between 1 and 10000 (we collect PoP information from Rocketfuel [32] traces), randomly selecting a PoP, and measuring the overhead to disconnect and reconnect it to the graph. We found that repair did not trigger any massive spikes in overhead, which was roughly on the same order of magnitude of rejoining all the hosts in the PoP. Finally, we repeated this experiment for 10 million partitions and our approach converged correctly in every case; we perform consistency checks for misconverged rings in the simulator.

6.3 Interdomain

Join overhead: Figure 8a shows the overhead to join a single host. On the x-axis we vary the number of IDs in the AS, and on the y-axis we plot a moving average of the join overhead over the last 200 joins, averaged over 3 runs. We compare four joining strategies: *ephemeral*, where the host joins only at its global successor, *single-homed*, where the host joins only via a single path towards the core, *recursively multihomed*, where the host joins via all ASes above it in the topology, and *recursively multihomed+peering* (which we call Peering), where the host also joins across all adjacent peering links. The last approach provides the strongest guarantees on isolation, but comes at an increased join overhead. The join overhead for peering can be reduced to that of multihoming with the bloom filter optimization discussed in Section 4.2, at the expense of larger per-router state requirements. Surprisingly however, the cost of a multi-homed join is not significantly larger than that of a single-homed join. This happens because although there are typically 75-100 ASes in an AS's up-hierarchy, and the multi-homed join must discover a successor through each, there are typically a much smaller number of *unique* successors. We leveraged this observation to optimize the multi-homed join, by eliminating redundant lookups that resolve to the same successor. Next, we roughly extrapolated these results to an Internet-scale system with 600 million IDs, and estimate that the ephemeral join requires around 14 messages, the single-homed join requires around 80 messages, and the multi-homed join requires around 100 messages. Moreover, it should be noted that these control messages are more lightweight than traditional routing protocols, since intermediate routers do not need to process these messages in their slow-paths. However, we found that using the bloom filter optimization reduced the overhead of the peering join to be equal to the overhead of the recursively multihomed join. Finally, the state at hosting routers increases with the number of hosts and the number of fingers hosts maintain. We found that with 600 million IDs each maintaining 256 fingers, we required on average 184 Mbits per AS to store hosting state.

Stretch: Figure 8b shows a CDF of data packet stretch for single-homed joins. Stretch decreases with the number of proximity-based fingers: with 60 fingers, ROFL's average stretch is 2.8, while stretch is 2.3 for 160 fingers. If hosts perform a join across peering links as well, the stretch increases to 2.8 for 160 fingers. We found that

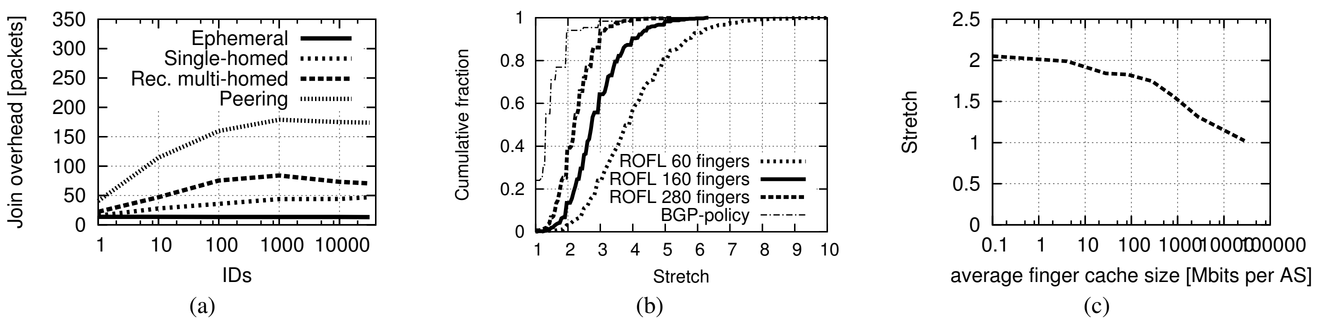


Figure 8: ***Interdomain routing:*** **(a) Comparison of joining strategies (b) Stretch (c) Effect of pointer caching**

stretch decreased slightly (not shown here) as the number of IDs in the system increased. This decrease happens because there is a highly uneven distribution of hosts across ASes in the Internet, and hence as we scale up the number of IDs the chances that the source and destination are in the same AS increases. We roughly extrapolated these results to an Internet-scale graph with 600 million IDs, and estimated 128 fingers (peering join overhead of 200) gives a stretch of around 2.9, and 340 fingers (peering join overhead of 445) gives a stretch of around 2.5. However, increasing the number of fingers also increases the size of the join messages that carry proximity-fingers. For example, with 256 fingers the message size increases to 1638 bytes. If we assume an MTU of 1500 bytes, a 256-finger single-homed join requires 258 IP packets.

Although a stretch of 2-3 seems high, it need only be suffered by the first packet: stretch for remaining packets can be reduced to one by exchanging the list of ASes above the destination in the hierarchy (Section 5.1), or by caching the destination's AS. As a comparison point we plot the stretch incurred today by BGP policies, measured using Routeviews traces (shown as BGP-policy in Figure 8b). In addition, we found that the isolation property contributes significantly to reducing stretch (through consistency checks in our simulator, we verified there were no cases in any of our experiments when the isolation property was broken). Next, Figure 8c shows pointer caching (Section 4.1) reduces stretch further. In these experiments, we model each AS with a pointer cache as a single node, and make the size of this cache proportional to the number of hosts in that AS. The x-axis shows the average amount of pointer caching state per AS, extrapolated to an Internet-scale topology with 600 million hosts. An average pointer cache size of 20M entries per AS reduces stretch from 2 to 1.33 (note that routers today can support millions of entries). Finally, we found that using bloom filters for peering as described in Section 4.2 results in a stretch of 3.29 with size 18 Mbits/AS, though this stretch can be reduced to 2.5 with more fingers or larger 74 Mbit bloom filters.

Failures: *Stub* ASes (ASes near the network edge) are believed to be significantly more unstable than ISPs near the core [14]. In this experiment we fail randomly selected stub ASes and measure two metrics. First, we measure the number of paths affected by the failure. We found on average 99.998% of Internet paths were unaffected by the failure, indicating that the effects of failures were well contained. Next, we found that ROFL required on average 4950 messages to repair successors after a stub AS failure, which roughly corresponds to the number of identifiers hosted in the failed stub AS.

6.4 Summary of results

Intradomain: Based on Rocketfuel traces, we simulated ROFL over four ISPs, ranging in size from 201 to 604 internal routers. ROFL is able to provide a routing stretch of 1.2 to 2 with 9Mbits of pointer cache, with reasonable load balance across routers. Hosts typically complete joining in less than 40ms, with less than 45 control messages generated per host. ROFL correctly heals from partitions, host failures, and host mobility with control overhead roughly that of rejoining the affected hosts.

Interdomain: We extrapolated our simulation results over the AS graph to the Internet scale system with 600 million hosts, and estimated that a ROFL host can join across all providers and peers and acquire 340 fingers with ~445 control messages. This overhead can be reduced for unstable hosts by performing a single-homed join (~75 messages), or an ephemeral join (~14 messages). The host can route packets in a manner that respects several inter-AS policies, with an average stretch of 2.5. This stretch may be reduced to 2.1 by roughly doubling the number of fingers. By maintaining pointer caches at border routers, this stretch may be reduced further (to 1.33 with on average 20 million entries of caching space per AS). Finally, ASes may reduce join overhead by leveraging bloom filters to eliminate joins across peering links. This reduces join overhead to ~100 messages, but requires 74 Mbits of bloom filter state per AS.

7. RELATED WORK AND DISCUSSION

While we've drawn general insights from many sources, our detailed mechanism owes much to two particular sources: VRR [7], which was the basis for our intradomain design, and Canon [17], which was the basis for our interdomain design. Given that VRR was designed for a very different context, ad-hoc routing, we build upon VRR by introducing a simplified path construction/maintenance protocol, a protocol to ensure correctness in the presence of network partitions, and several approaches to improve scalability and resilience to churn. We similarly extend Canon, by modifying the design to support several Internet policies, and leveraging proximity-based fingers to reduce stretch.

The project that seems to have the most in common with our design objectives is TRIAD [10], and its content routing design in [18]. TRIAD routes on URLs by mapping URLs to next-hops. In theory, every network router could do this but, because of load concerns, TRIAD only performs content routing at gateways (firewalls/NATs) between realms and BGP-level routers between ASes. Forwarding state is built up in intermediate content routers as packets are routed, and name suffix reachability is distributed among address realms just like BGP distributes address prefixes among ASes. It thus relies on aggregation to scale, and will fail if object locations do not follow the DNS hierarchy closely. If, to counteract this, name-level redirection mechanisms are used to handle hosts whose names do not match network topology, then this becomes essentially a resolution mechanism. This last comment also applies to IPNL, which also does some routing on FQDNs.

These previous forays into the name-routing arena suggest not only its difficulty but also its worth. Routing on names brings with it several architectural benefits, as we alluded to in the Introduction, but most of all it breaks out of a long-standing architectural mindset. The art of architecture is gracefully maneuvering within the boundaries of the possible. Our goal here is to investigate whether those boundaries can be expanded, not to seek grace.

Our design has a reasonable set of features; multiple delivery models, a fair amount of policy control, and some, but not much, traffic control. The remaining question, then, is about performance. On that score, we view this work as the beginning, not the end. The results are close enough to tempt, but not enough to satisfy.

8. REFERENCES

[1] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Maloo, S. Ron, “Practical locality-awareness for large scale information sharing,” IPTPS, February 2005.

[2] T. Anderson, T. Roscoe, D. Wetherall, “Preventing Internet denial-of-service with capabilities,” *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[3] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, M. Walfish, “A layered naming architecture for the Internet,” *ACM SIGCOMM*, August 2004.

[4] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker, “‘Off by Default!,” HotNets, 2005.

[5] H. Ballani, P. Francis. ”Towards a Global IP Anycast Service,” ACM SIGCOMM, Aug 2005

[6] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422-426, July 1970.

[7] M. Caesar, M. Castro, E. Nightingale, G. O’Shea, A. Rowstron, “Virtual ring routing: network routing inspired by DHTs,” *ACM SIGCOMM*, September 2006.

[8] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach, “Secure routing for structured peer-to-peer overlay networks” *OSDI*, December 2002.

[9] M. Castro, P. Druschel, Y. Charlie Hu, A. Rowstron, “Exploiting network proximity in peer-to-peer overlay networks,” Microsoft Research technical report MSR-TR-2002-82, 2002.

[10] D. Cheriton, M. Gritter, “TRIAD: a scalable deployable NAT-based Internet architecture,” Technical report, January 2000.

[11] D. Clark, R. Braden, A. Falk, V. Pingali, “FARA: reorganizing the addressing architecture,” *SIGCOMM FDNA Workshop*, August 2003.

[12] S. Deering, D. Cheriton. ”Multicast Routing in Datagram Internetworks and Extended LANs,” ACM TOCS, 1990.

[13] J. Douceur, “The Sybil Attack” *IPTPS*, March 2002.

[14] A. Feldmann, O. Maennel, Z. Mao, A. Berger, B. Maggs, “Locating Internet routing instabilities,” *ACM SIGCOMM*, August 2004.

[15] B. Ford, “Unmanaged internet protocol: taming the edge network management crisis,” *HotNets*, Cambridge, MA, Nov. 2003.

[16] P. Francis, R. Gummadi, “IPNL: a NAT-extended Internet architecture,” *ACM SIGCOMM*, August 2002.

[17] P. Ganesan, K. Gummadi, H. Garcia-Molina, “Canon in G major: designing DHTs with hierarchical structure,” ICDCS, March 2004.

[18] M. Gritter and D. Cheriton, “An Architecture for Content Routing Support in the Internet,” In the USENIX Symposium on Internet Technologies and Systems, March 2001.

[19] M. Handley and A. Greenhalgh, “Steps towards a DoS-resistant internet architecture,” FDNA, 2004.

[20] J. Jannotti, “Network layer support for overlay networks,” PhD thesis, MIT, August 2002.

[21] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. W. O’Toole Jr, ”Overcast: Reliable Multicasting with an Overlay Network,” *OSDI*, October 2000.

[22] P. Jokela, P. Nikander, J. Melen, J. Ylitalo, J. Wall, “Host identity protocol - extended abstract,” in *Wireless World Research Forum*, February 2004.

[23] A. Jonsson, M. Folke, B. Ahlgren, “The split naming/forwarding network architecture,” *Proc. Swedish National Computer Networking Workshop (SNCNW)*, September 2003.

[24] D. Krioukov, kc claffy, “Toward compact interdomain routing,” Unpublished draft, `http://www.krioukov.net/~dima/pub/cir.pdf`

[25] D. Krioukov, K. Fall, X. Yang, “Compact routing on Internet-like graphs,” *IEEE Infocom* , March 2004.

[26] D. Mazieres, “Self-certifying file system,” PhD thesis, MIT, May 2000.

[27] A. Myers, E. Ng, H. Zhang, “Rethinking the service model: scaling ethernet to a million nodes,” *HotNets*, November 2004.

[28] M. O’Dell, “GSE - an alternate addressing architecture for IPv6,” ftp://ds.internic.net/internet-drafts/draftietf-ipngwg-gseaddr-00.txt, 1997.

[29] L. Peterson, S. Shenker, J. Turner, “Overcoming the Internet impasse through virtualization,” *HotNets*, November 2004.

[30] A. Rowstron, P. Druschel, “Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems,” IFIP/ACM Middleware, November 2001.

[31] J. Saltzer, “On the naming and binding of network destinations,” RFC 1498, August 1993.

[32] N. Spring, R. Mahajan, D. Wetherall, “Measuring ISP topologies with Rocketfuel,” *ACM SIGCOMM*, August 2002.

[33] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, S. Surana, “Internet indirection infrastructure,” *ACM SIGCOMM*, August 2002.

[34] I. Stoica, R. Morris, D. Lieben-Nowell, D. Karger, M. Kaashoek, F. Dabek, H. Balakrishnan, “Chord: a scalable peer-to-peer lookup protocol for Internet applications,” *IEEE Transactions on Networks*, 11(1) 17-32, 2003.

[35] L. Subramanian, S. Agarwal, J. Rexford, R. Katz,“Characterizing the Internet Hierarchy from Multiple Vantage Points,” in *IEEE Infocom 2002*, June 2002.

[36] L. Subramanian, M. Caesar, C. Ee, M. Handley, M. Mao, S. Shenker, I. Stoica, “HLP: a next-generation interdomain routing protocol,” *ACM SIGCOMM*, August 2005.

[37] M. Walfish, H. Balakrishnan, S. Shenker, “Untangling the web from DNS,” *NSDI* March 2004.

[38] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, S. Shenker, “Middleboxes no longer considered harmful,” *OSDI*, December 2004.

[39] F. Wang, L. Gao, “Inferring and characterizing Internet routing policies,” *Proc. Internet Measurement Conference*, October 2003.

[40] Abraham Yaar, Adrian Perrig, Dawn Song, “Pi: A Path Identification Mechanism to Defend against DDoS Attacks,” IEEE Symposium on Security and Privacy, 2003.

[41] X. Yang, “NIRA: a new Internet routing architecture,” SIGCOMM Workshop on Future Directions in Network Architecture (FDNA), August 2003.

[42] X. Yang, D. Wetherall, and T. Anderson, “A DoS-limiting Network Architecture,” ACM SIGCOMM 2005, August 2005.

[43] CAIDA, “Skitter,” `http://www.caida.org/tools/measurement/skitter`.

[44] “FIND: future Internet network design,” `http://find.isi.edu`, December 2005.

[45] “GENI: global environment for network innovations,” `http://www.geni.net`

[46] Internet Systems Consortium, “Domain survey host count,” `http://www.isc.org/index.pl?/ops/ds/`, July 2005.

[47] ‘NewArch project: future-generation Internet architecture,‘‘ `http://www.isi.edu/newarch/`

[48] “Route Views Project,” `http://www.routeviews.org`.