Scalable Internet Routing on Topology-Independent Node Identities

Bryan Ford
Massachusetts Institute of Technology

October 31, 2003

Abstract

Unmanaged Internet Protocol (UIP) is a fully self-organizing network-layer protocol that implements scalable *identity-based routing*. In contrast with address-based routing protocols, which depend for scalability on centralized hierarchical address management, UIP nodes use a flat namespace of cryptographic node identifiers. Node identities can be created locally on demand and remain stable across network changes. Unlike location-independent name services, the UIP routing protocol can stitch together many conventional address-based networks with disjoint or discontinuous address domains, providing connectivity between any pair of participating nodes even when no underlying network provides direct connectivity. The UIP routing protocol works on networks with arbitrary topologies and global traffic patterns, and requires only $O(log\ N)$ storage per node for routing state, enabling even small, ubiquitous edge devices to act as ad-hoc self-configuring routers. The protocol rapidly recovers from network partitions, bringing every node up-to-date in a multicast-based chain reaction of $O(log\ N)$ depth. Simulation results indicate that UIP finds routes that are on average within $2\times$ the length of the best possible route.

This technical report describes a work in progress and does not contain complete, final, or polished results. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

1 Introduction

Routing protocols for flat node namespaces are traditionally limited in scalability by per-node storage or per-node routing traffic overheads that increase at least linearly with the size of the network. The scalability of today's Internet to millions and soon billions of nodes is currently possible only through *address-based* routing, in which topology information is embedded into structured node addresses. Classless Inter-Domain Routing (CIDR) [28] enables IP routers to store detailed routing information only for nodes and subnets within a local administrative domain, aggregating all routing information about more distant networks into larger address blocks.

The scalability of CIDR depends on careful assignment of node addresses to mirror the structure of the network, however. Manual IP address assignment is tedious and technical, while dynamic assignment [7] makes addresses unstable over time and cripples nodes in edge networks that become temporarily disconnected from assignment services [4]. Organizational resistance plagues IP address renumbering efforts [2], and host mobility and multihoming violate the hierarchical CIDR model, leading to extensions demanding additional care and feeding [27, 11]. Firewalls and network address translators (NATs) create discontinuous address domains [31], making remote access and peer-to-peer communication difficult [13]. Finally, new networking technologies may require fundamentally different and incompatible address architectures [33, 16]. These factors suggest that no single address-based routing protocol, let alone a single centrally-administered routing domain, may ever provide connectivity between every pair of nodes in the world that want to communicate.

UIP is a scalable *identity-based* internetworking protocol, designed to fill the connectivity gaps left by address-based protocols such as IP. UIP stitches together multiple address-based layer 2 and layer 3 networks into one large

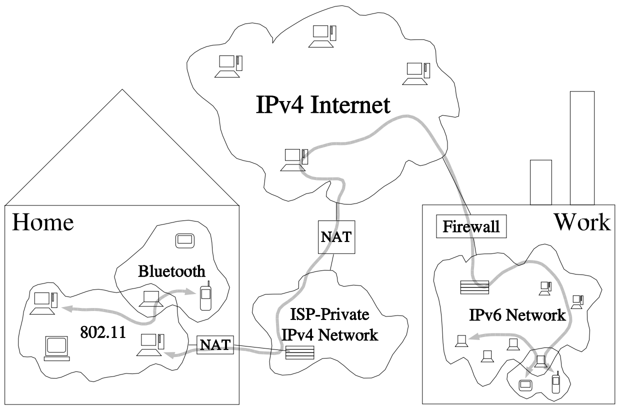


Figure 1: Today's Internetworking Challenges

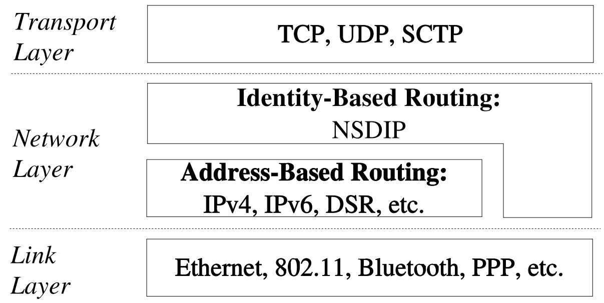


Figure 2: UIP in the Internet Protocol Architecture

"layer 3.5" internetwork, in which nodes use topology-free identifiers in a flat namespace instead of hierarchical addresses. All UIP nodes act as self-configuring routers, enabling directly- or indirectly-connected UIP nodes to communicate via paths that may cross any number of address domains.

1.1 A Motivating Example

Joe Average User has a Bluetooth-enabled phone, a laptop with both Bluetooth and 802.11 support, and several other 802.11-only devices on his home network, as illustrated in Figure 1. He just moved in, however, and does not yet have a working Internet connection. With UIP running on each of these devices, Joe's Bluetooth phone can communicate through his laptop with all of his other 802.11 devices. The laptop acts as a self-configuring router for all of the devices reachable on his home network, without Joe having to assign any addresses manually.

Joe eventually obtains an Internet connection and deploys a home NAT, which turns out to be located behind a larger NAT deployed by his (cheap) ISP. When his Internet connection becomes active, Joe's home devices automatically merge into the global UIP network and he can access them through any other Internet-connected UIP host. While at his friend Jim's home, for example, Joe's Bluetooth phone automatically discovers and connects with Jim's PC, a well-connected Internet node that also runs UIP. Joe can then use his phone to control and remotely access the devices in his home, exactly as he would if he was at home. Again no configuration is required; Joe's home devices and Jim's PC automatically conspire with other UIP nodes on the Internet to build the necessary forwarding paths.

Joe's company runs an IPv6 network behind a firewall with a highly restrictive forwarding policy, but the firewall permits UIP traffic to and from specific internal hosts whose installed software the company's network administrator trusts. Joe is fortunate enough to have such a trusted host at work, which likewise merges into the global UIP network. Joe can now access his home devices from work and his work PC from home, and he can access any of them from his Bluetooth phone while at either location. Joe's network administrator must set up the firewall policy to allow UIP traffic to Joe's work machine, but Joe doesn't have to do anything.

1.2 UIP's Role in the Internet

UIP sits on top of existing address-based network-layer protocols such as IPv4 and IPv6, and can also operate directly over link-layer protocols such as Ethernet, 802.11, and Bluetooth (see Figure 2). Upper-level protocols and applications use UIP in the same way they as they use traditional address-based network-layer protocols. Instead of addresses, however, upper-level protocols and applications name and connect with other UIP nodes using *cryptographic identifiers*, comparable to Moskowitz's proposed host identities [21]. Since UIP node identifiers have no relationship to network topology, nodes can create their own identifiers without reference to central authorities, and node identifiers remain valid as long as desired even as the node moves and the surrounding network topology changes.

This paper focuses purely on UIP's routing and forwarding algorithms, leaving other aspects of UIP to be developed in future work. For this reason, the exposition of the protocol in this paper is high-level and algorithmic in nature. The only properties UIP node identifiers have that are of importance in this paper are that they are relatively uniformly distributed in a flat namespace.

1.3 Key Properties of UIP

In contrast with conventional routing algorithms for flat namespaces, UIP's routing protocol has only $O(log\ N)$ per-node storage and update traffic requirements. UIP achieves this scalability by distributing routing information throughout the network in a self-organizing structure adapted from the Kademlia distributed hash table (DHT) algorithm [18]. Unlike location-independent naming services such as DHTs, UIP does not assume that underlying protocols provide connectivity between any two nodes. When address-based routing protocols fail to provide direct connectivity for any reason, such as intermittent glitches, network address translators, or incompatible address-based routing technologies, UIP routes around these discontinuities by forwarding traffic through other UIP nodes.

The cost of distributing routing information throughout the network for scalability is that individual UIP nodes rarely have enough information to determine the shortest or "best" possible route to another node. In effect, UIP does not implement a distributed "all-pairs shortest paths" algorithm like conventional protocols for flat namespaces do [15]. Instead, UIP attempts the more moderate goal of *efficiently* finding *some* path whenever one exists, and usually finding reasonably short paths. This goal is appropriate for UIP since the purpose of UIP is to find communication paths that address-based protocols such as IP cannot find at all.

In general we cannot expect identity-based routing to be as efficient as routing protocols that take advantage of the locality and aggregation properties of structured addresses. UIP is not intended to replace address-based routing protocols, but to complement them. By using address-based protocols such as IP to move data efficiently across the many "short" hops comprising the core Internet infrastructure and other large managed networks, UIP only needs to route data across across a few "long" hops, resolving the discontinuities between address domains and bridging managed core networks to ad hoc edge networks. For this reason, it is less important for UIP to find the best possible route all the time, and more important for the algorithm to be scalable, robust, and fully self-managing.

We explore two specific UIP forwarding mechanisms based on the same routing protocol. One mechanism guarantees that nodes can operate in $O(log\ N)$ space per node on any network topology. The other forwarding mechanism allows UIP to find somewhat better routes and still uses $O(log\ N)$ space on typical networks, but may require $O(N)$ space on worst-case network topologies. With either forwarding mechanism, simulations indicate that UIP consistently finds paths that are on average within $2\times$ the length of the best possible path. UIP occasionally chooses paths that are much longer than the best possible path, but these bad paths are rare.

1.4 Road Map

The rest of this paper is organized as follows. Section 2 details the routing protocol by which UIP nodes organize and find paths to other nodes, and Section 3 describes the two alternative mechanisms UIP nodes use to forward data between indirectly connected nodes. Section 4 evaluates the routing and forwarding protocol and demonstrates key properties through simulations. Section 5 summarizes related work, and Section 6 concludes.

2 The Routing Protocol

This section describes the distributed lookup and routing structure that enables UIP nodes to locate and communicate with each other by their topology-independent identities.

2.1 Neighbors and Links

Each node in a UIP network maintains a *neighbor table*, in which the node records information about all the other UIP nodes with which it is actively communicating at a given point in time, or with which it has recently communicated. The nodes listed in the neighbor table of a node A are termed A's *neighbors*. A neighbor of A is not necessarily "near" to A in either geographic, topological, or node identifier space; the presence of a neighbor relationship merely reflects ongoing or recent pairwise communication.

Some neighbor relationships are mandated by the design of the UIP protocol itself as described below, while other neighbor relationships are initiated by the actions of upper-level protocols. For example, a request by an upper-level protocol on node A to send a packet to some other node B effectively initiates a new UIP neighbor relationship between A and B. These neighbor relationships may turn out to be either ephemeral or long-term. A UIP node's neighbor table is analogous to the table an IPv4 or IPv6 host must maintain in order to keep track of the current path maximum transmission unit (MTU) and other vital information about other endpoints currently or recently of interest to upper-level protocols.

As a part of each entry in a node's neighbor table, the node's UIP implementation maintains whatever information it needs to send packets to that particular neighbor. This information describes a *link* between the node and its

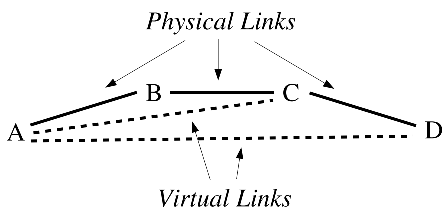


Figure 3: Forwarding via Virtual Links

// build a link from node n to target node n_t,
// using node n_w as a waypoint if necessary
n.**build_link**(n_w, n_t) {

  assert (n and n_w are neighbors)
  assert (n_w and n_t are neighbors)

  try to contact n_t by its IP address, MAC address, etc.
  if direct contact attempt succeeds {
    build physical link from n to n_t
  } else {
    build virtual link from n to n_t via n_w
  }

  assert (n and n_t are neighbors)
}

Figure 4: Pseudocode to Build a Physical or Virtual Link

neighbor. A link between two nodes A and B may be either physical or virtual. A *physical link* is a link for which connectivity is provided directly by some underlying protocol. For example, if A and B are both well-connected nodes on the Internet that can successfully communicate via their public IP addresses, then AB is a physical link from the perspective of the UIP layer, even though this communication path may in reality involve many hops at the IP layer and even more hops at the link layer. If a physical link is available between A and B, then A and B are termed *physical neighbors*, and each node stores the other's IP address or other address information for underlying protocols in the appropriate entry of its neighbor table.

A *virtual link*, in contrast, is a link between two nodes that can only communicate by forwarding packets through one or more intermediaries at the UIP level. We describe such nodes as *virtual neighbors*. The mechanism for UIP-layer packet forwarding and the contents of the neighbor table entries for a node's virtual neighbors will be described later in Section 3. For now, however, we will simply assume that the following general principle holds. Given any two existing physical or virtual links AB and BC with endpoint B in common, nodes A and C can construct a new virtual link AC between them by establishing a UIP-level forwarding path through B. That is, UIP nodes can construct new virtual links recursively from existing physical and virtual links.

In Figure 3, for example, virtual link AC builds on physical links AB and BC, and virtual link AD in turn builds on virtual link AC and physical link CD. Once these virtual links are set up, node A has nodes B, C, and D in its neighbor table, the last two being *virtual neighbors*. Node D only has nodes C and A as its neighbors; D does not necessarily need to know about B in order to use virtual link AC.

2.2 Constructing Virtual Links

UIP nodes construct new virtual links with a single basic mechanism, represented by the **build_link** procedure shown in Figure 4. A node n can only build a virtual link to some other node n_t if n already has some "waypoint" node n_w in its neighbor table, and n_w already has n_t in its neighbor table respectively. Node n can then use the **build_link** procedure to construct a link from n to n_t.

In the **build_link** procedure, n first attempts to initiate a direct connection to n_t via underlying protocols, using any network- or link-layer address(es) for n_t that n may have learned from n_w. For example, if n_t is a node with several network interfaces each in different address domains, then n_t might publish both the IP addresses and the IEEE MAC addresses of all of its network interfaces, so that other UIP nodes in any of these domains can initiate direct connections with n_t even if they don't know exactly which domain they are in. If at least one of these direct connection attempts succeeds, then n now has n_t as a physical neighbor, and a virtual link is not necessary.

If all direct connection attempts fail (or do not succeed quickly enough), however, then n constructs a virtual link to n_t using n_w as a forwarding waypoint. In this way, the **build-link** procedure takes advantage of underlying connectivity for efficiency whenever possible, but succeeds even when only indirect connectivity is available.

2.3 UIP Network Structure

While virtual links provide a basic forwarding mechanism, UIP nodes must have an algorithm to determine *which* virtual links to create in order to form a communication path between any two nodes. For this purpose, all UIP connected nodes in a network self-organize into a distributed structure that allows any node to locate and build a communication path to any other by resolving the

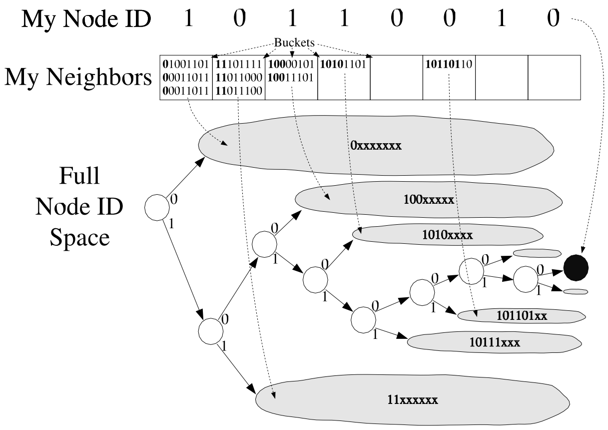


Figure 5: Neighbor Tables, Buckets, and Node ID Space

// build a communication path from node n
// to target node n_t
n.**build_path**(n_t) {
  $i = 1$
  $b_1 = prox(n, n_t)$
  $n_1 = n$.neighbor_table$[b_1]$
  while ($n_i \neq n_t$) {
    $b_{i+1} = prox(n_i, n_t)$
    assert ($b_{i+1} > b_i$)

    $n_{i+1} = n_i \rightarrow$ find_neighbor_in_bucket (b_{i+1})
    if find_neighbor_in_bucket request failed {
      return *failure*: node n_t does not exist or is not reachable.
    }

    n.build_link(n_i, n_{i+1})
    assert (n_{i+1} is now n's neighbor)

    $i = i + 1$
  }
  return *success:* we now have a working link to n_t.
}

Figure 6: Pseudocode to Build a Path to Any Node

target node's identifier one bit at a time from left to right. The UIP network structuring algorithm is closely related to peer-to-peer distributed hash table (DHT) algorithms such as Pastry [30] and Kademlia [18]. Unlike DHTs, however, UIP uses this self-organizing structure not only to look up information such as the IP or MAC address(es) of a node from its UIP identifier, but also as a basis for *constructing* UIP-level forwarding paths between nodes for which underlying protocols provide no direct connectivity.

For simplicity of exposition we will assume that each node has only one identifier, each node's identifier is unique, and all identifiers are generated by the same l-bit hash function. We will treat UIP node identifiers as opaque l-bit binary bit strings. The *longest common prefix* (LCP) of two nodes n_1 and n_2, written $lcp(n_1, n_2)$, is the longest bit string prefix common to their respective UIP identifiers. The *proximity* of two nodes $prox(n_1, n_2)$ is the length of $lcp(n_1, n_2)$: the number of contiguous bits their identifiers have in common starting from the left. For example, nodes 1011 and 1001 have an LCP of 10 and a proximity of two, while nodes 1011 and 0011 have an empty LCP and hence a proximity of zero. Nodes that are "closer" in identifier space have a higher proximity. Since node identifiers are unique, $0 \leq prox(n_1, n_2) < l$ if $n_1 \neq n_2$, and $prox(n, n) = l$.

Each node n divides its neighbor table into l buckets, as illustrated in Figure 5, and places each of its neighbors n_i into bucket $b_i = prox(n, n_i)$ corresponding to that neighbor's proximity to n. This distance metric, also known as the XOR metric [18], has the important symmetry property that if node n_2 falls into bucket b of node n_1's neighbor table, then n_1 falls into bucket b of n_2's neighbor table. This symmetry facilitates the establishment of pairwise relationships between nodes, and allows both nodes in such a relationship to benefit from requests flowing between them in either direction.

In order for a UIP network to be fully functional, the network must satisfy the following *connectivity invariant*. Each node n perpetually maintains an active connection with at least one neighbor in every bucket b, as long a reachable node exists anywhere in the network that could fit into bucket b. In practice each node attempts to maintain at least k active neighbors in each bucket at all times, for some redundancy factor k.

2.4 Building Communication Paths

If the connectivity invariant is maintained throughout a UIP network, then any node n can communicate with any target node n_t by the following procedure, outlined in pseudocode in Figure 6.

Node n first looks in bucket $b_1 = prox(n, n_t)$ of its own neighbor table. If this bucket is empty, then n_t does not exist or is not reachable, and the search fails. If the bucket contains n_t itself, then the target node is already an active neighbor and the search succeeds. Otherwise, n picks any neighbor n_1 from bucket b_1. Since n_1's and n_t's proximity to n are both b_1, the first b_1 bits of n_1 and n_t match those of n's identifier, while their immediately following bits are both opposite that of n. The proximity of n_1 to n_t is therefore at least $b_1 + 1$.

Node n now sends a message to n_1 requesting n_1's nearest neighbor to n_t. Node n_1 looks in bucket $b_2 = p(n_1, n_t)$ in *its* neighbor table, and returns information about at least one such node, n_2, if any are found. The information returned includes the UIP identifier of the nodes found along with any known IP addresses, IEEE MAC addresses, or other underlying protocol addresses for those nodes. Node n then uses the **build_link** procedure in Figure 4 to establish a connection to n_2, via a direct physical link if possible, or a virtual link through n_1 otherwise.

Now n_2 is also an active neighbor of n, falling into the same bucket of n's neighbor table as n_1 but closer in proximity to n_t. The original node n continues the search iteratively from n_2, resolving at least one bit per step and building additional recursive virtual links as needed, until it finds the desired node or the search fails. If the search eventually succeeds, then n will have n_t as an active (physical or virtual) neighbor and communication can proceed.

In practice, nodes can improve the robustness and responsiveness of the **build_path** procedure by selecting a set of up to k neighbor nodes at each iteration and making **find_neighbor** requests to all of them in parallel, in much the same way that Kademlia parallelizes its DHT lookups. Parallelizing the construction of UIP communication paths has the added benefit that the originating node is likely to end up having discovered several alternate paths to the same node. The originating node can evaluate these alternative paths using some suitable criteria and choose the best of them for subsequent communication, and keep information about the others stored away for use if the primary path fails. The two endpoint nodes can even balance their traffic load across these paths if they can find reason to believe that the paths are sufficiently independent for load-balancing to be effective in improving overall performance.

2.5 The Merge Procedure

The above **build_path** procedure is much like the lookup procedure used in the Kademlia DHT, modified to support construction of indirect forwarding paths between nodes that cannot communicate directly via underlying protocols. For network construction and maintenance, however, UIP requires a much more robust algorithm than those used in Kademlia and other DHTs. DHTs generally assume not only that underlying protocols provide full any-to-any connectivity between nodes, but also that nodes join or leave the network at a limited rate and relatively independently of each other. In the discontinuous network topologies on which UIP is intended to run, however, a single broken link can split the network at arbitrary points, causing the nodes in either partition to perceive that all the nodes in the other partition have disappeared *en masse*. If the network split persists for some time, the nodes on either side will re-form into two separate networks, which must somehow be merged again once the networks are re-connected.

```
// merge node n into the portion of a network
// reachable from neighbor n1
n.merge(n1) {
  i = 1
  b1 = prox(n, n1)
  while (bi < l) {

    for j = 0 thru (bi − 1) {
      if n.neighbor_table[j] not already full {
        nj = ni → find_neighbor_in_bucket (j)
        if find_neighbor_in_bucket request succeeded {
          n.build_link(ni, nj)
        }
      }
    }

    ni+1 = ni → find_neighbor_in_bucket (bi)
    if find_neighbor_in_bucket request failed
      break
    bi+1 = prox(n, ni+1)
    assert (bi+1 > bi)

    n.build_link(ni, ni+1)
    i = i + 1
  }
}
```

Figure 7: Pseudocode to Merge a Node Into a Network

Our algorithm assumes that underlying protocols provide some means by which topologically near UIP nodes can discover each other and establish physical neighbor relationships. For example, UIP nodes might use Ethernet broadcasts IPv4 subnet broadcasts, or IPv6 neighbor discovery to detect nearby neighbors automatically. Nodes might also contain "hard-coded" IP addresses of some well-known UIP nodes on the Internet, so that nodes with working Internet connections can quickly merge into the public Internet-wide UIP network. Finally, the user might in some cases explicitly provide the address information necessary to establish contact with other relevant UIP nodes. Whenever a new physical link is established by any of the above means, the node on each end of the link performs the **merge** procedure outlined in Figure 7, to merge itself into the network reachable from the other node.

The merge process works as follows. Suppose that node n has node n_1 as a neighbor, falling in bucket $b_1 = p(n, n_1)$ in its neighbor table. If $b_1 > 0$, then n and n_1 have one or more initial identifier bits in common, and any neighbors of n_1 in buckets 0 through $b_1 - 1$ are also suitable for the corresponding buckets in n's neighbor table. Node n therefore requests information from n_1 about about at least one of n_1's neighbors in each of these buckets, and builds a physical or virtual (via n_1) link to that node. Assuming n_1's neighbor table satisfied the connectivity invariant, n's neighbor table now does as well for buckets 0 through $b_1 - 1$.

Node n now asks n_1 for any neighbor from n_1's bucket b_1 other than n itself, as if n was searching for its own identifier in n_1's network. If such a node n_2 is found, then its proximity $b_2 = p(n, n_2)$ must be at least $b_1 + 1$. Node n builds a link to n_2 via n_1, fills any empty buckets $0 < b_i < b_2$ from n_2's neighbor table as above, and then continues the process from n_2 for neighbors with proximity greater than b_2. Eventually n reaches some node n_i with proximity b_i, whose bucket b_i contains no neighbors other than n itself. This means that there are no other nodes in n_1's network with greater proximity to n than p_i, and so n has satisfied the connectivity invariant in its own neighbor table, at least with respect to the portion of the network reachable from n_1.

2.6 Merge Notifications

After a node n merges into another node n_1's network via the **merge** procedure above, however, there may be other nodes in n_1's network besides the ones that n contacted directly that also need to learn about n before *their* neighbor tables will satisfy the connectivity invariant for the new, larger network. In addition, n may not be just a "lone" node joining n_1's network, but may instead be a member of a larger existing network (reachable from n's neighbor table) that previously split from or evolved independently from n_1's network. In this case, many nodes in n's network may need to learn about nodes in n_1's network, and vice versa, before the connectivity invariant will be re-established globally.

To cause other nodes to update their neighbor tables appropriately, UIP uses a simple notification mechanism. Whenever a node n makes contact for any reason with a new physical or virtual neighbor n_n, and bucket $b_n = prox(n, n_n)$ of n's neighbor table *was not full* before the addition of n_n, n sends a message to all of its existing neighbors notifying them of the new node n_n. In response to this notification message, each of n's existing neighbors n_i contacts n_n via n_i.build_link(n, n_n), and then likewise merges into n_n's network via n_i.merge(n_n). If this process helps n_i to fill any of its previously underfull neighbor table buckets, then n_i subsequently sends notifications to *its* neighbors, and so on. The chain reaction stops when all of the affected nodes cease finding new nodes that fit into underfull buckets in their neighbor tables.

To understand this process, consider two initially separate UIP networks: a "red" network consisting of i nodes $r_1 \dots r_i$, and a "green" network consisting of j nodes $g_1 \dots g_j$. We say that any given node n satisfies the *red connectivity invariant* if each bucket in n's neighbor table contains at least one red node if any red node exists that could fit into that bucket. Similarly, we say that a node n satisfies the *green connectivity invariant* if each of n's buckets contains at least one green node if any green node exists that could fit into that bucket. We assume that all green nodes initially satisfy the green connectivity invariant, but no green nodes satisfy the red connectivity invariant because there are initially no connections between the red and green networks. Similarly, all red nodes satisfy the red connectivity invariant but no red nodes satisfy the green connectivity invariant.

Now suppose that a physical link is somehow established between nodes r_1 and g_1, connecting the two networks. In response, r_1 performs a **merge**(g_1), filling any underfull buckets in its neighbor table that can be filled from green nodes reachable from g_1, and g_1 likewise performs a **merge**(r_1) to fill its buckets from nodes in the red network. Node r_1 effectively locates and builds links with its nearest (highest-proximity) neighbors in the green network, and g_1 likewise locates and builds links with its nearest neighbors in the red network. As a result, after the merge process r_1 satisfies the green connectivity invariant and g_1 satisfies the red connectivity invariant. Since r_1 and g_1 already satisfied the red and green invariants, respectively, and adding new neighbors to a node's neighbor table cannot "un-satisfy" a previously satisfied connectivity invariant, both r_1 and g_1 now satisfy the *global* connectivity invariant covering both red and green nodes.

Assuming node identifiers are reasonably uniformly distributed, with high probability one or both of r_1 and g_1 will find one or more new nodes in the opposite network that fit into previously underfull buckets. Before the merge, bucket $b = prox(r_1, g_1)$ in both r_1 and g_1 may already have been full, which is likely if r_1 and g_1 are far apart in identifier space. There may even be *no* nodes in the green network that fall into underfull buckets in r_1, but this event is unlikely unless the green network is much smaller than the red network. Similarly, there may be no nodes in the red network that fall into underfull buckets in g_1, but only if the red network is much smaller than the green network. If the two networks are similar in size, then *both* r_1 and g_1 will almost certainly find new neighbors that fit into underfull buckets.

At any rate, the discovery of new neighbors falling in these underfull buckets causes r_1 and/or g_1 to send merge notifications to their existing neighbors in the red and green networks, respectively, supplying a link to the opposite node as a "hint" from which other nodes in each network can start their merge processes. Each node in either network that is notified in this way initiates its own **merge** process to fill its neighbor table from nodes in the other network, in the process triggering the merge process in its other neighbors, eventually leaving all nodes satisfying the global connectivity invariant.

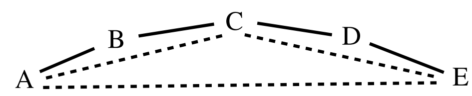
In practice it is important to ensure that the inevitable flurry of merge notifications does not swamp the whole network, especially when two relatively large networks merge. Standard protocol engineering solutions apply to this problem, however, such as rate-limiting the acceptance or spread of notifications, propagating merge notifications periodically in batches, and keeping a cache in each node of recently-seen merge notifications to avoid performing the same merge many times in response to equivalent merge notifications received from different neighbors.

3 Packet Forwarding

The previous section described how UIP nodes form a self-organizing structure in which any node can build a communication path to any other node by recursively constructing virtual links on top of other links, but did not specify exactly how virtual links operate. In this section we explore the construction and maintenance of virtual links in more detail. We will explore in particular two alternative methods for implementing virtual links: one based on source routing, the other based on recursive tunneling. Source routing potentially enables nodes to find more efficient routes and keeps the basic forwarding mechanism as simple as possible, while the recursive tunneling approach minimizes the amount of state each node must maintain in its neighbor table.

3.1 Source Routing

With source routing, each entry in a node's neighbor table that represents a virtual neighbor contains a complete *source route* to the target node. The source route lists the UIP identifiers of a sequence of nodes, starting with the origin node and ending with the target node, such that each adjacent pair in the sequence has (or recently had)



Source Routing

A's Neighbor Table:

B	Phys	123.45.67.89
C	Virt	A→B→C
E	Virt	A→B→C→D→E

"→" Refers to *physical links only*

Tunneling

A's Neighbor Table:

B	Phys	123.45.67.89
C	Virt	A⇨B⇨C
E	Virt	A⇨C⇨E

"⇨" Refers to *physical or virtual links*

Figure 8: Source Routing versus Recursive Tunneling

a working *physical* link between them. Of course, since these links need only be "physical" from the perspective of the UIP layer, each link in a UIP source route may represent many hops at the IP routing or link layers.

Consider for example Figure 8, in which the five nodes A, B, C, D, E are connected by a chain of physical links. Nodes A and C have established a virtual link AC by building a two-hop source route via their mutual neighbor B, and nodes C and E have similarly established a virtual link CE via D. Suppose node A subsequently learns about E from C and desires to create a virtual link AE via C. Node A contacts C requesting C's source route to E, and then appends C's source route for CE (A, B, C) to A's existing source route for AC (C, D, E), yielding the complete physical route A, B, C, D, E.

To send a packet to E, node A includes in the packet's UIP header the complete source route for the virtual link AE stored in its neighbor table entry for E. Each UIP node along the path examines the header to find the packet's current position along its path, and bumps this position indicator to the next position before forwarding the packet to the next UIP node in the path. Forwarding by source routing in UIP is thus essentially equivalent to source routing in IP [6].

In theory each node may have to store up to $l \times k$ entries in its neighbor table, where l is the node identifier size and hence the number of buckets in the neighbor table, and k is the redundancy factor within each bucket. In practice only the top $log_2 N$ buckets will be non-empty, where N is the total number of nodes in the network. With source route forwarding, neighbor table entries may have to hold source routes for paths up to $N - 1$ hops in length, in the worst-case network topology of N nodes connected together in one long chain. In this case each node may require $O(N\ log\ N)$ storage. In practical networks these source routes will of course be much shorter, so this large worst-case storage requirement may not be a problem.

3.2 Recursive Tunneling

In contrast with source routing, where each entry in a node's neighbor table for a virtual neighbor contains a complete, explicit route that depends only on physical links, recursive tunneling preserves the abstraction properties of neighbor relationships by allowing the forwarding path describing a virtual link to refer to both physical and (other) virtual links. As a result, each neighbor table entry representing a virtual link only needs to hold two UIP identifiers: the identifier of the target node, and the identifier of the "waypoint" through which the virtual link was constructed. Recursive tunneling therefore guarantees that each node requires at most $O(log\ N)$ storage, since neighbor table entries have constant size.

In the example in Figure 8, node A has constructed virtual link AC via B, and node C has constructed virtual link CE via D, and as before, A learns about E from C and wants to construct a virtual link AE via C. With recursive tunneling, A does not need to duplicate its route C or ask C for information about its route to E in order to construct its new virtual link to E. Instead, A merely depends on the knowledge that it already knows how to get to C, and that C knows how to get to E, and constructs a neighbor table entry for E describing the "high-level" two-hop forwarding path A, C, E.

Recursive tunneling has several beneficial properties. First, since each neighbor table entry for a virtual neighbor needs to store only two UIP identifiers, the size of each neighbor table entry can be limited to a constant, and the size of a node's entire neighbor table depends only on the size of UIP identifiers (and hence the number of buckets), and the number of entries in each bucket. Second, if "low-level routes" in the network change, all "higher-level routes" that are built on them will immediately use the correct, updated information with no information propagation delays. For example, if node D above goes down making the path C, D, E unavailable, but C finds an alternate route to E, then the virtual link AE will automatically use this new route without A even having to be aware that something in C's neighbor table changed.

The actual packet forwarding mechanism for recursive tunneling is of course slightly more involved than for source routing. As illustrated in Figure 9, to send a packet to E, node A wraps the packet data in three successive headers. First, it prepends a UIP tunneling header describing the "second-level" virtual path from A to E via C. Only nodes C and E will examine this header. Second,

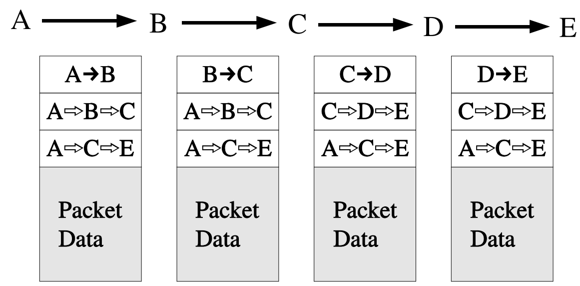


Figure 9: Forwarding by Recursive Tunneling

A prepends a second UIP tunneling header describing the "first-level" virtual path from A to C via B. Finally, A prepends the appropriate lower-layer protocol's header, such as an IP or Ethernet header, necessary to transmit the packet via the physical link from A to B.

When the packet reaches node B, B strips off the lower-layer protocol header, and looks in the first-level (outer) UIP tunneling header to find the UIP identifier of the next hop. B then looks up this identifier in its neighbor table, prepends the appropriate (new) lower-layer protocol header, and transmits the packet to C.

When the packet reaches node C, C strips off *both* the lower-layer protocol header and the first-level UIP tunneling header (since C was the destination according to that header), and examines the second-level tunneling header to find the final destination, E. C now looks up E in its neighbor table and, finding that E is a first-level virtual neighbor, C prepends a new first-level tunneling header describing the route from C to E via D. Finally, C prepends the lower-layer protocol header for the physical link from C to D and forwards the message to D. D subsequently forwards the message to E, which finally strips off the lower-layer protocol header and both of the tunneling headers before interpreting the packet data.

3.3 Path Optimization

When an upper-layer protocol on one node attempts to contact some other node via UIP, the **build_path** procedure described in Section 2.4 searches the network structure for the requested node identifier, and in the process may build one or more virtual links using the **build_link** procedure of Section 2.1. The search process through which these virtual links are constructed is essentially driven by the distance relationships in UIP identifier space, which have nothing to do with distance relationships in the underlying physical topology.

Each UIP node has complete flexibility, however, in the way it chooses the k nodes to fill a particular bucket in its neighbor table whenever there are more than k nodes in the network that could fit into that bucket. If the network contains N nodes with uniformly distributed identifiers, then we expect nodes to have some flexibility in their choice of neighbors throughout the first $log_2 N - log_2 k$ buckets. Further, we naturally expect nodes to select the "best" k nodes they find for each such bucket: either the closest in terms of physical topology (UIP hop count), or the best according to some other pragmatic measure involving latency, bandwidth, and/or reliability for example.

In general, therefore, we expect the first few iterations of the **build_path** process to stay within the node's immediate topological vicinity, with subsequent hops covering larger topological distances as the remaining distance in identifier space is progressively narrowed. While the first few **build_path** hops will depend only on physical or inexpensive "low-order" virtual links, the last few hops might each depend on an expensive "high-order" virtual link, eventually resulting in a communication path that criscrosses throughout the network in a highly non-optimal fashion. It is therefore important that we find a way to optimize the routes produced using this process.

The most basic path optimization is inherent in the **build_link** procedure. If a node A locates target node B via the **build_path** process, but A subsequently finds that it can contact B directly using underlying protocols such as IP using address information it discovers during the process, then **build_link** will "short-circuit" the path from A to B with a physical link requiring no UIP-level forwarding.

A second important path optimization is for nodes to check for obvious redundancies in the routes produced as higher-order virtual links are built from lower-order virtual links. Source routing makes this type of path optimization easier, since each node has information about the complete physical route to each neighbor in its neighbor table, but we will explore a more limited form of path optimization as well that works with recursive tunneling. Other path more sophisticated forms of path optimization are certainly possible and desirable, such as optimizations relying on a deeper analysis of the relationships between known neighbors, or based on additional information exchanged between neighbors beyond the minimal information requred to maintain the network and build virtual links. We leave more advanced path optimizations for future work, however, and focus for now on the effects of simple optimizations that rely on strictly local information.

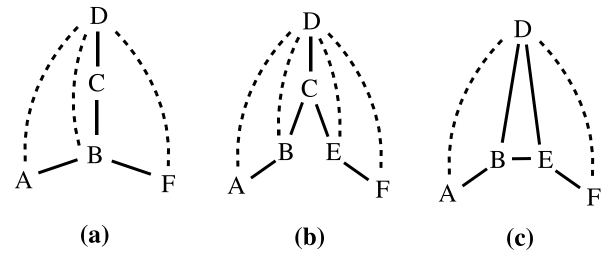


Figure 10: Path optimization opportunities on different topologies, when A builds a virtual link to F via D.

3.3.1 Source Route Optimization

In UIP forwarding by source routing, we optimize source routes when combining two shorter paths into a longer one simply by checking for nodes that appear in both shorter paths. For example, in Figure 10(a), suppose node A has established a virtual link AD via B with path A, B, C, D, by building on virtual link BD with path B, C, D. A virtual link also exists between D and F. A now learns about F through D and attempts to create a virtual link AF via D. Without path optimization, the resulting path will be A, B, C, D, C, B, F. The path can be trivially shortened to the optimal A, B, F, however, simply by noting that B appears twice and eliminating the redundant hops between them.

The same optimization shortens the path from A to F in Figure 10(b) from A, B, C, D, C, E, F to the optimal A, B, C, E, F. This path optimization does not help in the case of Figure 10(c), however, since A does not necessarily know that B and E are direct neighbors.

3.3.2 Recursive Tunnel Optimization

Path optimization is not as easy in forwarding by recursive tunnels, because the information needed to perform the optimization is more spread out through the network. For example, in Figure 10(a), node A knows that the first hop along virtual link AD is the physical link AB, but A does not necessarily know what type of link BD is and may not even know that node C exists.

In general, for any virtual link from n to n_1 via n_2, node n also contains in its neighbor table a virtual or physical link representing the first hop from n to n_2. If the lower-order link from n to n_2 is a virtual link via some node n_3, then n also contains in its neighbor table a physical or virtual link from n to n_3, and so on. We call this chain of intermediate nodes along the path from n to n_1 that n inherently knows about n's *first hop chain* for n_1. For example, A's first hop chain for D in Figure 10(a) is A, B, D, whereas D's first hop chain for A is D, C, B, A.

To implement path optimization for recursive tunnels, we extend the **build_link** procedure of Section 2.1 so that when a node n attempts to build a new virtual link to n_t via waypoint node n_w, n contacts its existing neighbor n_w requesting n_w's first hop chain for n_t. Node n then compares the information returned against its own first hop chain for n_w, and short-circuits any redundant path elements.

For example, in Figure 10(a), node A is building a virtual link to F via D, so A requests D's first hop chain to F, which is D, C, B, F. A compares this chain with its first hop chain for D, which is A, B, D, discovering redundant node B and shortening the path to A, B, F.

This form of path optimization does not help in Figure 10(b), however, where the redundant path component between C and D is hidden from A because C is not in A's first hop chain. Similarly, this optimization does not handle Figure 10(c) for the same reason that the source routing optimization above fails.

4 Protocol Evaluation

In this section we use simulation results to evaluate the behavior of UIP's routing and forwarding protocol. A "real-world" implementation of the protocol is under development, but until an implementation has been deployed and a substantial critical mass of users has developed, simulations provide the only realistic option for tuning the protocol and predicting how it will behave on the large networks it is intended to support.

4.1 Performance Metrics

In order to asses the basic viability of the UIP routing protocol, we focus here on measuring the efficiency of the network paths the protocol finds through random network topologies. Many other important factors that will affect the performance of real-world UIP networks remain for future study. In particular, while our simulations confirm that the protocol recovers from node failures and network partitions, we do not yet have a full characterization of the dynamic behavior of a UIP network under continuous change.

In order to measure the efficiency of routing paths chosen by UIP nodes, we define the *UIP path length* between two nodes n_1 and n_2 to be the total number of physical hops in the path that n_1 constructs to n_2 using the **build_path** procedure in Figure 6. We define the *stretch* between n_1 and n_2 to be the ratio of the UIP path length to the length of the best possible path through the underlying topology.

We measure the stretch for a given pair of nodes by using **build_path** to construct a path from one node to the other, measuring the total number of physical hops in the path, and then eliminating all the virtual links that **build_path** constructed so that the measurement of one path does not affect the measurement of subsequent paths. On networks of 100 nodes or less we measure all possible paths between any two nodes; on larger networks we take a sample of 10,000 randomly chosen node pairs.

4.2 Test Network Topology

Selecting appropriate network topologies for simulations of UIP is difficult, because we have no way to predict the topologies of the networks on which a protocol like UIP will actually be deployed. Using topological maps of the existing IPv4 Internet would not make sense: the existing well-connected Internet is precisely the portion of today's global network infrastructure across which UIP will *not* have to find paths, because IP already does that well enough, and UIP simply treats these paths as direct physical links. For the function UIP is designed provide, finding paths between nodes on the Internet and nodes on the many private and ad hoc networks attached to it, no reliable topological data is available precisely because most of these adjoining networks are private.

Nevertheless, we can construct artificial topologies that approximate the most important characteristics we believe this global network infrastructure to have. First, we expect the topology on which UIP is deployed to consist of many clusters, in which each node in a given cluster can reliably address and connect with any other node in the same cluster, but nodes in one cluster have very limited connectivity to nodes in other clusters. Second, because of the diversity of existing networking technologies and deployment scenarios, we expect the size of these clusters to follow a power law distribution, with larger clusters having better connectivity to neighboring clusters. Finally, we expect all of these clusters to be within at most a few hops from a single huge, central cluster, namely the public IP-based Internet.

To construct an artificial topology having these characteristics, we start with a single distinguished cluster we will call the *root cluster*, initially containing a single node. We then randomly "grow" the network one node at a time as follows. For each new node, we choose the number of attachment points the node will have based on a geometric random variable with a *multihoming probability* parameter p_m. Approximately $p_m N$ of the network's N

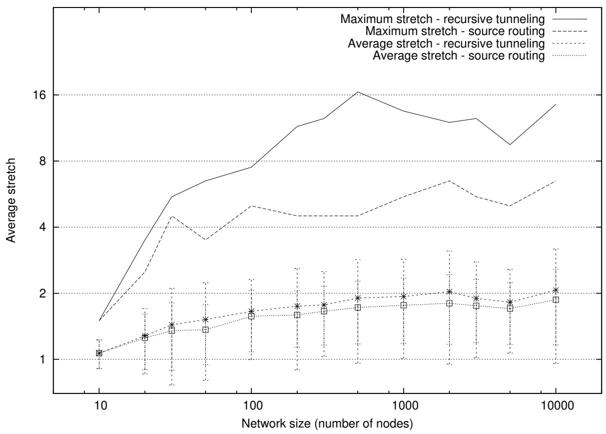


Figure 11: Network path stretch for source routing versus recursive tunneling

nodes will have at least two attachment points, $p_m^2 N$ have at least three attachment points, and so on.

We choose each attachment point for a new node via a random walk from the root cluster using a *downstream probability* parameter p_d and a *new cluster probability* parameter p_n. At each step, with probability p_d we move the attachment point downstream, and with probability $1 - p_d$ we terminate the process. To move the attachment point downstream, we choose a node at random from the current cluster, then we either create a new cluster "private" to that node with probability p_n, or else we with probability $1 - p_n$ we pick at random any cluster that node is attached to (which could be the cluster we just came from). Once the random walk terminates, we add the new node to the cluster at which the walk ended.

We call the resulting random network topology a *rooted* topology, since it consists of many small clusters centered around the single large root cluster, approximating the well-connected IP-based Internet surrounded by many smaller private networks.

We choose somewhat arbitrarily the following "baseline" parameters for our experiments. We use network topologies of varying sizes constructed with a multihoming probability $p_m = 1/10$, a downstream probability $p_d = 3/4$, and new link probility $p_n = 1/2$. On these topologies we build UIP networks with a redundancy factor $k = 3$, by adding nodes to the network one at a time in random order. We will vary these parameters to explore their impact on the efficiency of the routing protocol.

4.3 Source Routing versus Recursive Tunneling

In Figure 11 we measure the average and maximum path stretch observed (vertical axis) between any two nodes on networks of a given size (horizontal axis), for both source routing and recursive tunneling. The error bars indicate standard deviation of the measured stretch. In the random 10,000-node rooted topology, the root cluster contains 3233 nodes (32% of the network), the average distance between any two nodes is 2.5 hops, and the maximum distance between any two nodes (total network diameter) is 8 hops.

With both source routing and recursive tunneling, we see that the UIP routing protocol consistently finds paths that are on average no more than twice as long as the best possible path. The average-case efficiency of recursive tunneling is slightly worse than for source routing, due to the more limited amount of information nodes have to optimize paths they find through the network. The routing protocol occasionally chooses very bad paths—up to $6\times$ stretch for source routing and up to $16\times$ for recursive tunneling—but the low standard deviation indicates that these bad paths occur very rarely.

4.4 Rooted versus Unrooted Networks

We would next like to determine how much the UIP routing protocol benefits from the tree-like structure of rooted network topologies. Is the UIP routing protocol only viable when some underlying protocol such as IP is doing most of the work of routing within the large central cluster, or could UIP routing also be used to internetwork a number of small link-layer networks joined in ad-hoc fashion?

To explore this question, we modify the random network creation procedure of Section 4.2 so that the random walk to find each new attachment point for a given starts at a cluster chosen uniformly at random from all existing clusters, rather than at a well-known root cluster. The resulting *unrooted* topologies have a much more uniform and unpolarized distribution in their cluster sizes and in the connections between clusters. In the random 10,000-node unrooted topology, for example, the largest cluster contains only 11 nodes, the average distance between any two nodes is 7.7 hops, and the network diameter is 19 hops. We expect efficient routing on such a diffuse network to be more difficult than on a rooted network.

Figure 12 compares the path efficiency of UIP source route-based forwarding on rooted and unrooted networks of varying sizes. We find that unrooted networks indeed yield greater stretch, but not by a particularly wide mar-

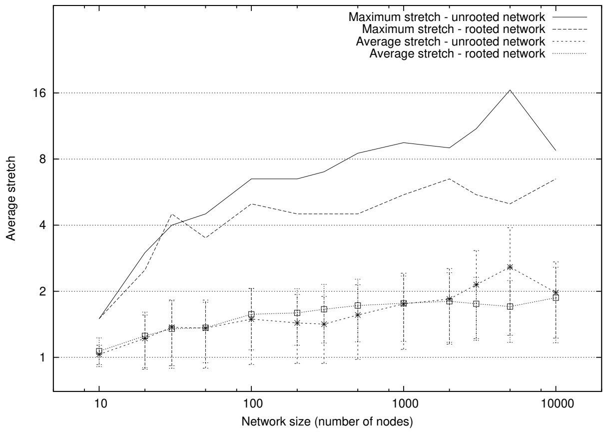


Figure 12: Network path stretch for rooted versus unrooted networks

gin. This result suggests that UIP routing is not highly dependent on rooted topologies and may be useable as well on more diffuse topologies.

5 Related Work

In this section we first compare the UIP routing and forwarding protocol with existing routing algorithms for both wired and ad hoc networks, then we relate UIP to location-independent name services and other systems with similar features.

5.1 Routing Algorithms

In classic distance-vector algorithms [15] such as RIP [12], as well as variants such as MS [19], WRP [23], and DSDV [26], each router continuously maintains routing information about every other addressable node or subnet. With these protocols, each router requires at least $O(N)$ storage for a network of size N, and must regularly exchange connectivity information of size $O(N)$ with each of its neighbors.

In link-state algorithms such as OSPF [22] and FSR [24], routers maintain complete network connectivity maps. This approach can achieve faster routing table convergence and avoid the looping problems of basic distance-vector algorithms, at the cost of even greater storage requirements and maintenance overhead.

Reactive or "on demand" routing algorithms designed for ad hoc networks, such as DSR [14] and AODV [25], require routers to store information only about currently active routes, limiting maintenance traffic and storage overheads on networks with localized traffic patterns. Routing queries for distant nodes may have to be broadcast through most of the network before the desired route is found, however, limiting the scalability of these protocols on networks with global traffic patterns.

Landmark [33], and related hierarchical protocols such as LANMAR [9], L+ [3], and PeerNet [8], dynamically arrange mobile nodes into a tree. The routing protocol assigns each node a hierarchical address corresponding to its current location in this tree, and implements a location-indepdendent identity-based lookup service by which the current address of any node can be found. Each non-leaf node serves as a *landmark* for all of its children, and is responsible for routing traffic to them from nodes outside its local subtree. Landmark routes local traffic purely within the lowest levels of the tree, providing scalability when traffic patterns are predominantly local. Since global traffic must pass through the landmark nodes at the upper levels of the hierarchy, however, these upper-level nodes are easily overloaded in a network with global traffic patterns.

5.2 Location-Independent Name Services

Naming services such as the Internet's domain name system (DNS) [20] can translate location-independent node names on demand to location-specific addresses. Name services inherently assume, however, that each node has *some* globally unique address at which it can be reached from all other nodes. If a desired node is on a private IP network behind a network address translator, for example, then there is generally *no* IP address by which it can be reached from outside the network, and name services do not help. Name-based routing [10] can bridge multiple IP address domains using DNS names, but its dependence on the centrally-administered DNS namespace makes it unsuitable for ad hoc networks.

Recent distributed hash table (DHT) algorithms such as Pastry [30], Chord [5], and Kademlia [18], implement fully decentralized, self-organizing name services that do not depend on top-down, hierarchical administration as DNS and other traditional name services do. The UIP routing protocol uses a self-organizing network structure closely related to the Kademlia DHT. Like conventional name services, however, DHT algorithms do not provide network-layer routing functionality. Although the process of locating up an item in a DHT is sometimes called "routing" because it involves iteratively contacting a sequence of nodes that are progressively closer to the desired item in identifier space, this process still assumes that the node initiating the lookup can directly contact each of the nodes in the sequence using underlying protocols. UIP's virtual link abstraction described in Section 2, and the forwarding mechanisms described in Section 3, provide the fundamental new functionality required to turn the Kademlia DHT into a network-layer routing protocol.

The Internet Indirection Infrastructure ($i3$) [32] uses a similar peer-to-peer search structure to implement location-independent naming and communication with multicast and anycast support. $i3$ provides special-case support for forwarding traffic to hosts behind firewalls, but it depends on all participating hosts being connected to the Internet at all times and does not implement general network-layer routing functionality.

5.3 Other Systems

A resilient overlay networks (RON) [1] serves a function similar in spirit to UIP, increasing the reliability of an IP network by detecting connectivity failures in the underlying network and forwarding traffic around them. RON makes no attempt at scalability beyond a few dozen nodes, however, and assumes that all participating nodes have unique IP addresses.

Several protocols have been developed to provide connectivity through firewalls and NATs, such as SOCKS [17], STUN [29], and UPnP [34]. These special-purpose protocols are tied to the characteristics and network topologies of commonly deployed NATs and firewalls, however, and do not solve the more general problem of routing between different address domains connected in arbitrary fashion.

6 Conclusion

Today's global network infrastructure has grown in size and diversity beyond the reach of any single address-based internetworking protocol. IPv4 and IPv6 have the scalability necessary to route between millions or billions of nodes, but their centrally-administered hierarchical address domains make edge networks dependent on either tedious manual address assignment or continual connectivity to address services such as DHCP. Existing ad hoc networking protocols are fully self-configuring, but they do not have the scalability of IP.

UIP, a scalable *identity-based* routing protocol, stitches together multiple address-based routing domains into a single flat namespace, enabling any-to-any communication via location-independent node identifiers. With identity-based routing, participating nodes in private IP address domains and ad hoc edge networks become uniformly accessible from anywhere while connected to the global Internet. Even while disconnected from the global Internet, however, with UIP these edge networks remain functional and maximally interconnected.

Simulation-based experiments with UIP indicate that its routing and forwarding protocol is practical and scalable. Although UIP nodes do not have enough information to choose the best possible routes to other nodes, the routes chosen by UIP nodes are on average no more than twice as long as the optimal route. These preliminary results suggest that identity-based routing on Internet-scale networks may indeed be viable.

References

[1] David G. Andersen et al. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[2] B. Carpenter, J. Crowcroft, and Y. Rekhter. IPv4 address behaviour today, February 1997. RFC 2101.

[3] Benjie Chen and Robert Morris. L+: Scalable landmark routing and address lookup for multi-hop wireless networks. Technical Report 837, Massachusetts Institute of Technology Laboratory for Computer Science, March 2002.

[4] Microsoft Corporation. Plug and play networking with Microsoft automatic private IP addressing, March 1998.

[5] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[6] DARPA. Internet protocol, September 1981. RFC 791.

[7] R. Droms. Dynamic host configuration protocol, March 1997. RFC 2131.

[8] Jakob Eriksson, Michalis Faloutsos, and Srikanth Krishnamurthy. Peernet: Pushing peer-to-peer down the stack. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[9] Mario Gerla, Xiaoyan Hong, and Guangyu Pei. Landmark routing for large ad hoc wireless networks. In *In Proceedings of IEEE GLOBECOM*, San Francisco, CA, November 2000.

[10] M. Gritter and D. R. Cheriton. An architecture for content routing support in the Internet. In *Usenix Symposium on Internet Technologies and Systems*, March 2001.

[11] J. Hagino and H. Snyder. IPv6 multihoming support at site exit routers, October 2001. RFC 3178.

[12] C. Hedrick. Routing information protocol, June 1988. RFC 1058.

[13] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator, January 2001. RFC 3027.

[14] David B. Johnson. Routing in ad hoc networks of mobile hosts. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 158–163. IEEE Computer Society, December 1994.

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

[16] Young-Bae Ko and Nitin H. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 66–75, 1998.

[17] M. Leech et al. SOCKS protocol version 5, March 1996. RFC 1928.

[18] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[19] P. M. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Transactions on Communications*, COM-27(9):1280–1287, September 1979.

[20] P. Mockapetris. Domain names - implementation and specification, November 1987. RFC 1035.

[21] R. Moskowitz and P. Nikander. Host identity protocol architecture, April 2003. Internet-Draft (Work in Progress).

[22] J. Moy. OSPF version 2, July 1991. RFC 1247.

[23] Shree Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2):183–197, 1996.

[24] Guangyu Pei, Mario Gerla, and Tsu-Wei Chen. Fisheye state routing: A routing scheme for ad hoc wireless networks. In *Proceedings of the IEEE International Conference on Communications*, pages 70–74, New Orleans, LA, June 2000.

[25] Charles E. Perkins and Elizabeth M. Belding-Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, February 1999.

[26] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[27] C. Perkins, Editor. IP mobility support for IPv4, August 2002. RFC 3344.

[28] Y. Rekhter and T. Li (editors). An architecture for IP address allocation with CIDR, September 1993. RFC 1518.

[29] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs), March 2003. RFC 3489.

[30] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[31] P. Srisuresh and K. Egevang. Traditional IP network address translator (Traditional NAT), January 2001. RFC 3022.

[32] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, 2002.

[33] Paul Francis Tsuchiya. The Landmark hierarchy: A new hierarchy for routing in very large networks. In *Proceedings of the ACM SIGCOMM*, pages 35–42, Stanford, CA, August 1988.

[34] UPnP Forum. Internet gateway device (IGD) standardized device control protocol V 1.0, November 2001. `http://www.upnp.org/`.