Internet Indirection Infrastructure

Authors anonymized

Abstract

Attempts to generalize the Internet's point-to-point communication abstraction to provide services like multicast, anycast, and mobility have faced challenging technical problems and deployment barriers. To ease the deployment of such services, this paper proposes an overlay-based Internet Indirection Infrastructure ($i3$) that offers a rendezvous-based communication abstraction. Instead of explicitly sending a packet to a destination, each packet is associated with an identifier; this identifier is then used by the receiver to obtain delivery of the packet. This level of *indirection* decouples the act of sending from the act of receiving, and allows $i3$ to efficiently support a wide variety of fundamental communication services. To demonstrate the feasibility of this approach, we have designed and built a prototype based on the Chord lookup system.

1 Introduction

The original Internet architecture was designed to provide *unicast* point-to-point communication between fixed locations. In this basic service, the sending host knows the IP address of the receiver and the job of IP routing and forwarding is simply to deliver packets to the (fixed) location of the desired IP address. The simplicity of this point-to-point communication abstraction contributed greatly to the scalability and efficiency of the Internet.

However, many applications would benefit from more general communication abstractions, such as multicast, anycast, and host mobility. In these abstractions, the sending host no longer knows the identity of the receiving hosts (multicast and anycast) and the location of the receiving host need not be fixed (mobility). Thus, there is a significant and fundamental mismatch between the original point-to-point abstraction and these more general ones. All attempts to implement these more general abstractions have relied on a layer of *indirection* that decouples the sending hosts from the receiving hosts; for example, senders send to a group address (multicast or anycast) or a home agent (mobility), and the IP layer of the network is responsible for delivering the packet to the appropriate location(s).

Although these more general abstractions would undoubtedly bring significant benefit to end-users, it remains unclear how to achieve them. These abstractions have proven difficult to implement scalably at the IP layer [4, 12, 22]. Moreover, deploying additional functionality at the IP layer requires a level of community-wide consensus and commitment that is hard to achieve. In short, implementing these more general abstractions at the IP layer poses difficult technical problems and formidable deployment barriers.

In response, many researchers have turned to application-layer solutions (either end-host or overlay mechanisms) to support these abstractions [4, 14, 19]. While these proposals achieve the desired functionality, they do so in a very disjointed fashion in that solutions for one service are not solutions for other services; *e.g.*, proposals for application-layer multicast don't address mobility, and vice-versa. As a result, many similar and largely redundant mechanisms are required to achieve these various goals. In addition, if overlay solutions are used, adding a new abstraction requires the deployment of an entirely new overlay infrastructure.

In this paper, we propose a single new overlay network that serves as a general-purpose Internet indirection infrastructure ($i3$). $i3$ offers a powerful and flexible *rendezvous*-based communication abstraction; applications can easily implement a variety of communication services, such as multicast, anycast, and mobility, on top of this communication abstraction. Our approach provides a general overlay service that avoids both the technical and deployment challenges inherent in IP-layer solutions and the redundancy and lack of synergy in more traditional application-layer approaches. We thus hope to combine the generality of IP-layer solutions with the deployability of overlay solutions.

The paper is organized as follows. In Sections 2 and 3 we provide an overview of the $i3$ architecture and then a general discussion of how $i3$ might be used in applications. Section 4 covers additional aspects of the design such as scalability and efficient routing. Section 5 describes some simulation results on $i3$ performance along with a discussion of an initial implementation. Related work is discussed in Section 6, followed by a discussion on future work Section 7. We conclude with a summary in Section 8.

2 $i3$ Overview

In this section we present a brief overview of $i3$. We start with the basic service model and communication abstraction, then briefly describe $i3$'s design.

2.1 Service Model

The purpose of $i3$ is to provide indirection; that is, it decouples the act of sending from the act of receiving. The $i3$ service model is simple: sources send packets to a logical *identifier*, and receivers express interest in packets sent to an identifier. Delivery is best-effort like in today's Internet, with no guarantees about packet delivery.

This service model is similar to that of IP multicast. The crucial difference is that the $i3$ equivalent of an IP multicast join is more flexible. IP multicast offers a receiver a binary decision of whether or not to receive packets sent to that group (this can be indicated on a per-source basis). It is up to the multicast infrastructure to build efficient delivery trees. The $i3$ equivalent of a join is inserting a *trigger*. It allows receivers to *control* the routing of the packet. This provides two advantages. First, it allows them to create, at the application level, services such as mobility, anycast, and service composition out of this basic service model. Thus, this one simple service model can be used to support a wide variety of application-level communication abstractions, alleviating the need for many parallel and redundant overlay infrastructures. Second, the infrastructure can give responsibility for efficient tree construction to the end-hosts. This allows the infrastructure to remain simple, robust, and scalable.

$i3$'s Application Programming Interface (API)	
$sendPacket(p)$	send packet
$insertTrigger(t)$	insert trigger
$removeTrigger(t)$	remove trigger

(a)

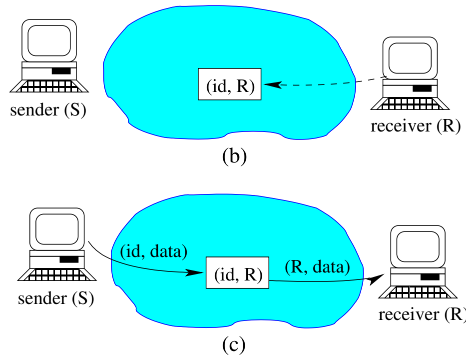


Figure 1: (a) $i3$'s API. Example illustrating communication between two nodes: (b) The receiver R inserts trigger (id, R). (c) The sender sends packet $(id, data)$.

2.2 Rendezvous-Based Communication

The service model is instantiated as a rendezvous-based communication abstraction. In their simplest form, packets are pairs $(id, data)$ where id is an m-bit identifier and $data$ consists of a payload (typically a normal IP packet payload). Receivers use *triggers* to indicate their interest in packets. In the simplest form, triggers are pairs $(id, addr)$, where id represents the trigger identifier, and $addr$ represents a node's address which consists of an IP address and a port number. A trigger $(id, addr)$ indicates that all packets sent to an identifier id should be forwarded (at the IP level) by the $i3$ infrastructure to the node with IP address $addr$. More specifically, the rendezvous-based communication abstraction exports three basic primitives as shown in Figure 1(a).

Figure 1(b) illustrates the communication between two nodes, where receiver R wants to receive packets sent to id. The receiver inserts the trigger (id, R) into the network. When a packet is sent to identifier id, the trigger causes it to be forwarded via IP to R.

Thus, much as in IP multicast, the identifier id represents a logical rendezvous between the sender's packets and the receiver's trigger. This level of indirection decouples the sender from the receiver. The sender need neither be aware of the number of receivers nor their location. Similarly, receivers need not be aware of the number or location of senders.

The above description is the simplest form of the abstraction. We now describe a generalization that allows inexact matching between identifiers. (A second generalization that replaces identifiers with a stack of identifiers is described in Section 2.5.) We assume identifiers are m bits and that there is some *exact-match threshold* k with $k < m$. We then say that an identifier id_t in a trigger *matches* an identifier id in a packet if

(a) id and id_t have a prefix match of at least k bits
(b) There is no trigger with an identifier that has a longer prefix match with id

Thus, a trigger identifier id_t matches a packet identifier id if and only if id_t is a longest prefix match (among all other trigger identifiers) and this prefix match is at least as long as the exact-match threshold k. The value k is chosen to be large enough so that the probability that two randomly chosen identifiers match is negligible. [1] This allows end-hosts to choose the identifiers independently with negligible chance of collision.

2.3 Overview of the Design

We now briefly describe the infrastructure that supports this rendezvous communication abstraction; a more in depth description follows in Section 4. $i3$ is an overlay network which consists of a set of servers that store triggers and forward packets (using IP) between $i3$ nodes and to end-hosts. Identifiers and triggers have meaning only in this $i3$ overlay.

One of the main challenges in implementing $i3$ is to *efficiently* match the identifiers in the packets to those in triggers. This is done by mapping each identifier to a unique $i3$ node; at any given time there is a single $i3$ node responsible for a given id. When a trigger $(id, addr)$ is inserted, it is stored on the $i3$ node responsible for id. When a packet is sent to id it is routed by $i3$ to the node responsible for id; there it is matched against any triggers for that id and forwarded (using IP) to all hosts interested in packets sent to that identifier. To facilitate inexact matching, we require that all id's that agree in the first k bits be stored on the same $i3$ server. The longest prefix match required for inexact matching can be executed at a single node (where it can be done efficiently).

Note that packets are not stored in $i3$; they are only forwarded. $i3$ provides a best-effort service like today's Internet. $i3$ implements neither reliability nor ordered delivery on top of IP. End-hosts use periodic refreshing to maintain their triggers in $i3$. Hosts contact an $i3$ node when sending $i3$ packets or inserting triggers. This $i3$ node then forwards these packets or triggers to the $i3$ node responsible for the associated identifiers. Thus, hosts need only know one $i3$ node in order to use the $i3$ infrastructure.

2.4 Communication Primitives Provided by $i3$

We now describe how $i3$ can be used by applications to achieve the more general communication abstractions of mobility, multicast, and anycast.

[1] In our implementation we choose $m = 256$ and $k = 128$.

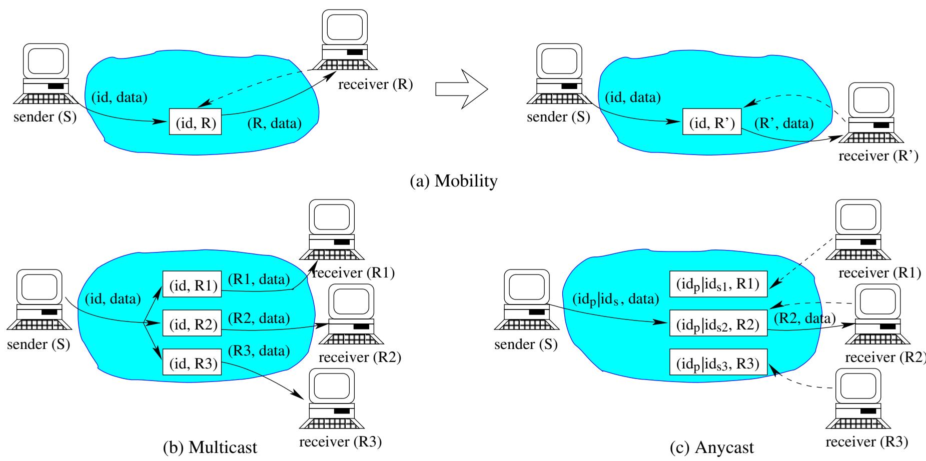


Figure 2: Communication abstractions provided by $i3$. (a) Mobility: The change of the receiver's address from R to R' is transparent to the sender. (b) Multicast: Every packet $(id, data)$ is forwarded to each receiver R_i that inserts the trigger (id, R_i). (c) Anycast: The packet matches the trigger of receiver $R2$. $id_p|id_s$ denotes an identifier of size m, where id_p represents the prefix of the k most significant bits, and id_s represents the suffix of the $m-k$ least significant bits.

2.4.1 Mobility

The form of mobility addressed here is when a host (*e.g.*, a laptop) is assigned a new address when it moves from one location to another.[2] A mobile host that changes its address from R to R' as a result of moving from one subnetwork to another can preserve the end-to-end connectivity by simply updating each of its existing triggers from (id, R) to (id, R'), as shown in Figure 2(a). The sending hosts need not be aware of the mobile host's current location or address.

With any scheme that supports mobility, efficiency is a major concern [20]. $i3$ employs three techniques to achieve efficiency. First, each packet is routed based on its identifier to the server that stores its trigger. This means that no additional operation needs to be invoked when the sender or the receiver move. Thus, $i3$ can maintain end-to-end connectivity even when both end-points move simultaneously. Second, the address of the server storing the trigger is cached at the sender, and subsequent packets are forwarded directly to that server via IP. This way, most packets are forwarded through only one $i3$ server in the overlay network. Third, to alleviate the triangle routing problem that has plagued mobile IP, end-hosts can use off-line heuristics to sample the $i3$ identifier space to place the triggers close to themselves (see Section 4.4 for details).

2.4.2 Multicast

Creating a multicast group is equivalent to having all members of the group register triggers with the same identifier id. As a result, any packet that matches id is forwarded to all members of the group (see Figure 1(b)). We discuss how to make this approach scalable in Section 3.4.

Note that unlike IP multicast, with $i3$ there is no difference between unicast or multicast packets, in either sending or receiving. Such an interface gives maximum flexibility to the application. An application can switch on-the-fly from unicast to multicast by simply having more hosts maintain triggers with the same identifier. For example, in a telephony application this would allow multiple parties to seamlessly join a two-party conversation. In contrast, with IP, an application has to at least change the IP destination address in order to switch from unicast to multicast.

2.4.3 Anycast

Anycast ensures that a packet is delivered to at most one receiver in a group. Anycast enables server selection, a basic building block for many of today's applications. To achieve this with $i3$, all hosts in an anycast group maintain triggers which are identical in the k most significant bits. These k bits play the role of the anycast group identifier. To send a packet to an anycast group, a sender uses an identifier whose k-bit prefix matches the anycast group identifier. The packet is then delivered to the member of the group whose trigger identifier best matches the packet identifier according to the longest prefix matching rule (see Figure 1(c)). Section 3.3 gives two examples of how end-hosts can use the last $m-k$ bits of the identifier to encode their preferences.

2.5 Stack of Identifiers

In this section, we describe a second generalization of $i3$, which replaces identifiers with identifier *stacks*. An identifier stack is a tuple of identifiers that takes the form $(id_1, id_2, id_3, \ldots, id_k)$ where id_i is either an identifier or an address. Packets p and triggers t are thus of the form:

$$\text{Packet } p = (id_{stack}, data)$$
$$\text{Trigger } t = (id, id_{stack})$$

The generalized form of packets allows a source to send a packet to a series of identifiers, much as in source routing. The generalized form of triggers allows a trigger to send a packet to another identifier rather than to an address. As we see below, these extensions to the abstraction allow for a much greater flexibility in their use. To illustrate this point, we show how identifier stacks can be used to

[2] We do not address the issue of a fixed address moving in space.

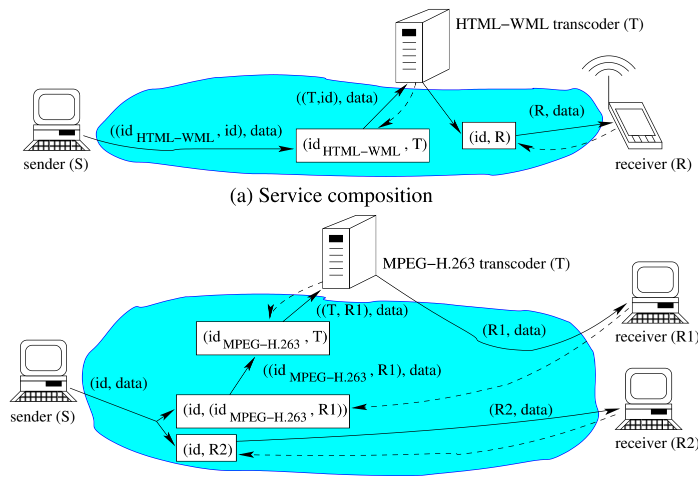
```
i3_forward(p) // send/forward packet p
  id = head(p.id_stack); // get head of p's stack
  if (type(id) = IP_ADDR_TYPE)
    IP_send(id, p); // id is an IP address; send p to id via IP
  else
    forward(p); // forward p via overlay network

i3_recv(p) // upon receiving packet p
  id = head(p.id_stack); // get head of p's stack
  // is local server responsible for id's best match?
  if (isMatchLocal(id) = FALSE)
    i3_forward(p); // matching trigger stored elsewhere
    return;
  pop(p.id_stack); // pop id from p's stack...
  set_t = get_matches(id); // get all triggers matching id
  if (set_t = ∅)
    if (p.id_stack = ∅)
      drop(p) // nowhere else to forward
    else
      i3_forward(p);
  while (set_t ≠ ∅) // forward packet to each matching trigger
    t = get_trigger(set_t);
    p1 = copy(p); // create new packet to send
    // ... add t's stack at head of p1's stack
    prepend(t.id_stack, p1.id_stack);
    i3_forward(p1);
```

Figure 3: Pseudo-code of the receiving and forward operations executed by an $i3$ server.

provide service composition, implement heterogeneous multicast, and increase $i3$'s robustness (see Sections 3.1, 3.2, and 4.2).

A packet p is always forwarded based on the first identifier id in its identifier stack until it reaches the server who is responsible for storing the matching trigger(s) for p. Consider a packet p with an identifier stack (id_1, id_2, id_3). If there is no trigger in $i3$ whose identifier matches id_1, id_1 is popped from the stack. The process is repeated until an identifier in p's identifier stack matches a trigger t. If no such trigger is found, packet p is dropped. If on the other hand, there is a trigger t whose identifier matches id_1, then id_1 is replaced by t's identifier stack. In particular, if t's identifier stack is (x, y), then p's identifier stack becomes (x, y, id_2, id_3). If id_1 is an IP address, p is sent via IP to that address, and the rest of p's identifier stack, i.e., (id_2, id_3) is forwarded to the application. The semantics of id_2 and id_3 are in general application-specific. However, in this paper we consider only examples in which the application is expected to use these identifiers to forward the packet after it has processed it. Thus, an application that receives a packet with identifier stack (id_2, id_3) is expected to send another packet with the same identifier stack (id_2, id_3). As shown in the next section this allows $i3$ to provide natural support for service composition.

Figure 3 shows the pseudo-code of the receiving and forwarding operations executed by an $i3$ node. Upon receiving a packet p, a server first checks whether it is responsible for storing the trigger *matching* packet p. If not, the server forwards the packet at the $i3$ level. If yes, the code returns the set of triggers that match p. For each matching trigger t, the identifier stack of the trigger is prepended to p's identifier stack. The packet p is then forwarded based on the first identifier in its stack.



(a) Service composition

(b) Heterogeneous multicast

Figure 4: (a) Service composition: The sender (S) specifies that packets should be transcoded at server T before being delivered to the destination (R). (b) Heterogeneous multicast: Receiver $R1$ specifies that wants to receive JPEG data, while $R2$ specifies that wants to receive MPEG data. The sender sends MPEG data.

3 Using $i3$

In this section we present a few examples of how $i3$ can be used. We discuss service composition, heterogeneous multicast, server selection, and large scale multicast. In the remainder of the paper, we say that packet p *matches* trigger t if the first identifier of p's identifier stack matches t's identifier.

3.1 Service Composition

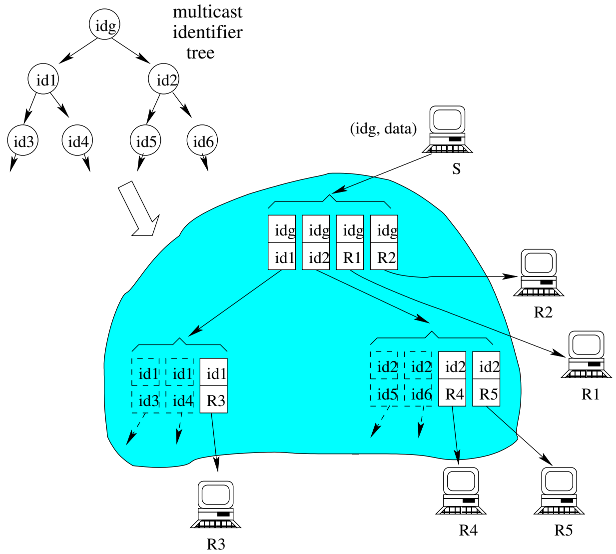
Some applications may require third parties to process the data before it reaches the destination [10]. An example is a wireless application protocol (WAP) gateway translating HTML web pages to WML for wireless devices [25]. WML is a lightweight version of HTML designed to run on wireless devices with small screens and limited capabilities. In this case, the server can forward the web page to a third-party server T that implements the HTML-WML transcoding, which in turn processes the data and send it to the destination via WAP.

In general, data might need to be transformed by a series of third-party servers before it reaches the destination. In today's Internet, the application needs to *know* the set of servers that perform transcoding and then *explicitly* forward data packets via these servers.

With $i3$, this functionality can be easily implemented by using a stack of identifiers. Figure 4(a) shows how data packets containing HTML information can be redirected to the transcoder, and thus arrive at the receiver containing WML information. The sender associates with each data packet the stack $(id_{HTML-WML}, id)$, where id represents the flow identifier. As a result, the data packet is routed first to the server which performs the transcoding. Next, the server inserts packet $(id, data)$ into $i3$, which delivers it to the receiver.

```
// host R joins multicast group id_G
// A is a known i3 server by R
R.join_multicast(id_G)
   t.id = id_G;
   t.stack_id = R;
   while (A.insertTrigger(t) == FAIL)
      // choose a child of t.id in the
      // multicast identifier tree
      t.stack_id = chooseChild(t.id);
      A.insertTrigger(t);
      t.id = t.stack_id;
      t.stack_id = R;
```

(a)



(b)

Figure 5: (a) Pseudo-code to join group id_G. (b) Example of a multicast group. Sender S inserts idg, and receivers $R1, \ldots, R5$ join the group. The multicast identifier tree stores all possible identifiers of the receiver triggers. For each identifier $th_{addrs} = 2$, and the degree of the multicast identifier tree is $h = 2$. As an example, $R1$ maintains trigger $(idg, R1)$, while $R5$ maintains triggers (idg, id_2) and (id_2, R_5). Possible but unused triggers are represented by dash lines.

3.2 Heterogeneous Multicast

Figure 4(b) shows a more complex scenario in which an MPEG video stream is played back by one H.263 receiver and one MPEG receiver.

To provide this functionality, we use the ability of the *receiver*, instead of the sender (see Section 2.5), to control the transformations performed on data packets. In particular, the H.263 receiver $R1$ inserts trigger $(id, (id_{MPEG-H.263}, R1))$, and the sender sends packets $(id, data)$. Each packet matches $R1$'s trigger, and as a result the packet's identifier id is replaced by the trigger's stack $(id_{MPEG-H.263}, T)$. Next, the packet is forwarded to the MPEG-H.263 transcoder, and then directly to receiver $R1$. In contrast, an MPEG receiver $R2$ only needs to maintain a trigger $(id, R1)$ in $i3$. This way, receivers with different display capabilities can subscribe to the same multicast group. Section 4.8 presents more details of our implementation of the heterogeneous multicast example on top of $i3$.

Another useful application can have the receiver insist that all data go through a firewall first before reaching it.

3.3 Server Selection

$i3$ provides good support for basic server selection through the use of the last $m - k$ bits of the identifiers to encode application preferences.[3] To illustrate this point consider two examples.

In the first example, assume that there are several web servers and the goal is to balance the client requests among these servers. This goal can be achieved by setting the $m - k$ least significant bits of both trigger and packet identifiers to random values. If servers have different capacities, then each server can insert a number of triggers proportional to its capacity. Finally, one can devise an adaptive algorithm in which each server varies the number of triggers as a function of its current load.

In the second example, consider the goal of selecting a server that is "close" to the client in terms of latency. To achieve this goal, each server can use the last $m - k$ bits of its trigger identifiers to encode its location. In the simplest case, the location can be the zip code of the place where the server is located. In each of the cases, the client uses the last $m - k$ bits in the packets' identifier to encode its own location; the longest prefix matching procedure used by $i3$ would result in the packet being forwarded to a server close to it.[4]

3.4 Large Scale Multicast

The multicast abstraction presented in Section 2.4.2 assumes that all members of a multicast group insert triggers with identical identifiers. Since triggers with identical identifier are stored at the same $i3$ server, that server is responsible for forwarding each multicast packet to every member of the multicast group. This solution obviously does not scale to large multicast groups. We now present a simple solution to address this problem. This solution assumes that each server maintains two additional thresholds for each trigger identifier id_t: (a) one threshold th_{addrs} for all triggers with identifier id_t that point to end-hosts, and (b) another threshold th_{ids} for triggers with identifier id_t that point to other triggers.

[3] Recall that identifiers are m bits that that k is the exact-matching threshold.

[4] Here we assume that nodes that are geographically close to each other are also close in terms of network distances, which is not always true. One could use instead "latency" based encoding, much as in [16].

The key idea is to build a hierarchy of triggers, where each member R of a multicast group, identified by id_G, replaces its trigger (id_G, R) by a chain of triggers (id_G, x_1), (x_1, x_2), ..., (x_i, R). Note that this substitution is transparent to the sender: a packet $(id_G, data)$ will still reach R via the chain of triggers. The chain maintained by each member represents a path through the tree. Thus, some of the triggers (x_i, x_{i+1}) may be shared by more than one receiver.

Figure 5 shows an iterative algorithm to construct such a hierarchy of triggers. The algorithm is completely decentralized, involving only end-hosts. With each multicast group id_G, we associate a complete and infinite h-ary tree ($h \leq th_{ids}$), called the *multicast identifier tree*. Each node has a unique identifier that can be independently computed by every member of the multicast group. In particular, the root's identifier is id_G, and the identifier of every other node in the tree is computed as a function of (1) node's index in the breadth-first traversal, and (2) group's identifier id_G.

The goal of a receiver R is then to find an identifier id in this tree that allows R to insert a trigger pointing to itself, (id, R). Recall that such an insertion is possible if and only if there are less than th_{addrs} triggers (identified by id) that point to end-hosts. Besides trigger (id, R), receiver R needs to maintain the chain of triggers $(id_G, id_1), \ldots, (id_i, id)$, where $id_G, id_1, \ldots, id_i, id$ represents the path from root to node id in the virtual identifier tree. To find such an id, each receiver R starts from the root and walks down the tree. At each level, R selects a node id_c among the children of the node selected at the previous level, id_p. The selection criteria for id_c can be as simple as random or as complex as the "closest" identifier to R, where the distance is represented by the network latency from R to the server responsible for that identifier. Then, R inserts trigger (id_p, id_c), and then tries to insert trigger (id_c, R). If the second insertion succeeds the algorithm terminates; otherwise, R continues by selecting a random child of id_c.

We make several observations. First, end-hosts need to choose trigger identifiers in a manner that reduces the chance of triggers part of the same multicast tree being stored on the same $i3$ servers. Otherwise, a few servers handle the entire multicast traffic. This can be accomplished by using a "good" random hash function such as SHA-1 to generate these identifiers based on the node index in the multicast identifier tree and the multicast group identifier. Second, as long as $h < th_{ids}$, the insertion of triggers of form (id_p, id_c) is guaranteed to succeed. This is because id_c can take at most h different values. Third, note that each member refreshes all triggers in its chain; this causes the servers storing triggers high in the hierarchy to receive an unacceptably large numbers of refreshes.[5] This problem is addressed in Section 4.5.

4 Additional Design and Performance Issues

In this section we discuss some additional $i3$ design and performance issues. The $i3$ design was intended to be (among other properties) robust, self-organizing, efficient, secure, scalable, incrementally deployable, and compatible with legacy applications. In this section we discuss these issues and some details of the design that are relevant to them.

Before addressing those issues, we first review our basic design, $i3$ is organized as an overlay network in which every node (server) stores a subset of triggers. At any moment of time a trigger is stored at only one server. Each end-host knows about one or more $i3$ servers. When a host wants to send a packet $(id, data)$, it forwards the packet to one of the servers it knows. If the contacted server doesn't store the trigger matching $(id, data)$, the packet is forwarded via IP to another server. This process continues until the packet reaches the server that stores the matching trigger. The packet is then sent to the destination via IP.

[5] In particular, the server storing the root trigger will receive refreshes from all members of the group!

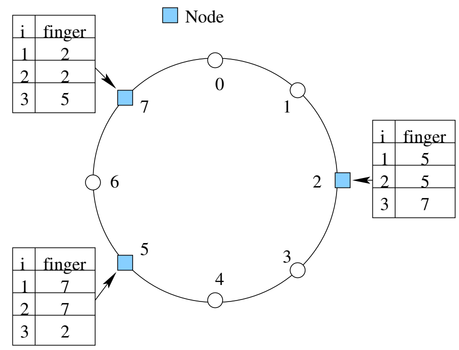


Figure 6: Routing information (finger tables) maintained by the Chord nodes.

4.1 Properties of the Overlay

The performance of $i3$ depends greatly on the nature of the underlying overlay network. In particular, we need an overlay network that exhibits the following desirable properties:

- **Robustness:** With a high probability, the overlay network remains connected even in the face of massive server and communication failures.
- **Scalability:** The overlay network can handle the traffic generated by millions of end-hosts and applications.
- **Efficiency:** Routing a packet to the server that stores the packet's best matching trigger involves a small number of servers, e.g., $O(\log N)$ servers where N is the total number of servers in the overlay network.
- **Stability:** The mapping between triggers and servers is relatively stable over time, that is, it is unlikely to change during the duration of a flow. This property allows end-hosts to optimize their performance by choosing triggers that are stored on nearby servers.

For correctness, we have chosen to implement $i3$ over Chord [21], which is known to satisfy the above properties. Chord uses an m-bit circular identifier space where 0 follows $2^m - 1$. Each server is associated a unique identifier in this space. In the original Chord protocol, each identifier id is mapped on the server with the closest identifier that follows id on the identifier circle. This server is called successor of id and it is denoted by $successor(id)$. Figure 6 shows an example in which there are three nodes, and $m = 3$. Server 2 stores identifiers 0, 1, and 2, server 5 stores 3, 4 and 5, and server 7 stores 6 and 7.

To implement the routing operation, each server maintains a routing table of size m. The i-th entry in the routing table of server n contains the first server that follows $n + 2^{i-1}$, i.e., $successor(n + 2^{i-1})$. This server is called the i-th finger of n and is denoted by $n.finger(i)$. Note that the first finger is the same as the successor server.

Upon receiving a packet with identifier id, server n checks whether id lies between itself and its successor. If yes, the server forwards the packet to its successor, which should store the packet's trigger. If not, n sends the packet to the closest server in its routing table that precedes id. In this way, we are guaranteed that the distance to id (in the identifier space) is halved at each step. As a result, it takes $O(\log N)$ hops to route a packet to the server storing the best matching trigger for the packet, irrespective of the starting point of the packet, where N is the number of $i3$ servers.

In the current implementation, we assume that all identifiers that share the same k-bit prefix are stored on the same server. A simple way to achieve this is to set the last $m - k$ bits of every node identifier to zero. As a result, finding the best matching trigger reduces to performing a longest prefix matching operation locally.

While $i3$ is implemented on top of Chord, in principle $i3$ can use any of the recently proposed P2P lookup systems such as CAN [17], Pastry [18] and Tapestry [27]. In the reminder of this section we discuss $i3$'s properties.

4.2 Robustness

$i3$ inherits much of the robustness properties of the overlay itself in that routing of packets within $i3$ is fairly robust against $i3$ node failures. In addition, end-hosts use periodic refreshing to maintain their triggers into $i3$. This soft-state approach allows for a simple and efficient implementation and frees the $i3$ infrastructure from having to recover lost state when nodes fail. If a trigger is lost – for example, as a result of an $i3$ server failure – the trigger will be reinserted, possibly at another server, the next time the end-host refreshes it.

One potential problem with this approach is that although the triggers are eventually reinserted, the time during which they are unavailable due to server failures may be too large for some applications. There are at least two solutions to address this problem. The first solution does not require $i3$-level changes. The idea is to have each receiver R maintain a backup trigger (id_{backup}, R) in addition to the primary trigger (id, R), and have the sender send packets with the identifier stack (id, id_{backup}). If the server storing the primary trigger fails, the packet will be then forwarded via the backup trigger to R.[6] Note that to acomodate the case when the packet is required to match everey trigger in its identifier stack (see Section 3.2), we use a flag in the packet header, which, if set, causes the packet to be dropped if the identifier at the head of its stack doesn't find a match. The second solution is to have the overlay network itself replicate the triggers and manage the replicas. In the case of Chord, the natural replication policy is to replicate a trigger on the immediate successor of the server responsible for that trigger [5]. Finally, note that when an end-host fails, its triggers are automatically deleted from $i3$ after they time-out.

[6] Here we implicitly assume that the primary and backup triggers are stored on different servers. The receiver can ensure that this is the case with high probability by choosing id_{backup} randomly.

4.3 Self-Organizing

$i3$ is an overlay infrastructure that may grow to large sizes. Thus, it is important that it not require extensive manual configuration or human intervention. The Chord overlay network is self-configuring, in that nodes joining the $i3$ infrastructure use a simple *bootstrapping* mechanism (see [21]) to find out about at least one existing $i3$ node, and then contacts that node to join the $i3$ infrastructure. Similarly, end-hosts wishing to use $i3$ can locate at least one $i3$ node using a similar bootstrapping technique; knowledge of a single $i3$ node is all that's needed to fully utilize the $i3$ infrastructure.

4.4 Routing Efficiency

As with any network system, efficient routing is important to the overall efficiency of $i3$. While $i3$ tries to route each packet efficiently to the server storing the best matching trigger, the routing in an overlay network such as $i3$ is typically far less efficient than routing the packet directly via IP. To alleviate this problem, the sender caches the $i3$ server's IP address. In particular, each data and trigger packet carry in their headers a refreshing flag f. When a packet reaches an $i3$ server, the server checks whether it stores the best matching trigger for the packet. If not, it sets the flag f in the packet header before forwarding it. When a packet reaches the server storing the best matching trigger, the server checks flag f in the packet header, and, if f is set, it returns its IP address back to the original sender. In turn, the sender caches this address and uses it to send the subsequent packets with the same identifier.

However, one problem still remains. If the sender and the receiver are close by, but the server storing the trigger is far away, the routing can still be inefficient. For example, assume that both the sender and the receiver are in the same city, but the server that stores the receiver's trigger is located on the other side of the world!

One solution to this problem is to have the receivers choose their triggers such that they are located on nearby servers. This would ensure that packets won't take a long detour before reaching their destination. To achieve this goal, each end-host can sample the identifier space to find ranges of identifiers that are stored at nearby servers. To find these ranges, a node A can insert random triggers (id, A) into $i3$, and then estimate the RTT to the server that stores the trigger by simply sending packets, $(id, dummy)$, to itself. Note that since we assume that the mapping of triggers onto servers is relatively stable over time, this operation can be done off-line.

4.5 Avoiding Hot-Spots

There are two types of hot-spots in $i3$. The first type is when a large number of clients try to contact a popular trigger such as CNN's trigger. The classical solution to this problem is to use caching. When the rate of the packets matching a trigger t exceeds a certain threshold, the server S storing the trigger pushes a copy of t to another server. This process can continue recursively until the load is spread out. The decision of where to push the trigger is subject to two constraints. First, S should push the trigger to the server most likely to route the packets matching that trigger. Second, S should try to minimize the state it needs to maintain; S at least needs to know the servers to which it has already pushed triggers in order to forward refresh messages for these triggers (otherwise the triggers will expire). With Chord, one simple way to address these problems is to always push the triggers to the predecessor server.

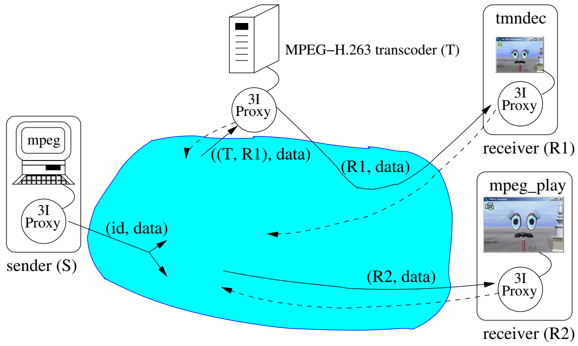


Figure 7: Heterogeneous multicast application. Refer to Figure 4 for data forwarding on $i3$.

If there are more triggers that share the same k-bit prefix with a popular trigger t, all these triggers need to be cached together with t. Otherwise, if the identifier of a packet matches the identifier of a cached trigger t, we cannot be sure that t is indeed the best matching trigger of the packet.

The second type of hot-spot occurs when a large number of clients refresh the same trigger. One example is the multicast algorithm presented in Section 3.4, in which all members of the group refresh the trigger at the root of the triggers' hierarchy (i.e., trigger $(10, 21)$ in Figure 5(b)). The solution is similar to the previous one. The only difference is that in this case we cache a *filter* rather than a trigger. The main function of a filter associated to trigger t is to intercept and suppress all but one refresh messages destined to t that arrive at that server during each refresh period.

4.6 Scalability

Since typically each flow is required to maintain two triggers (one for each end-point), the number of triggers stored in $i3$ is of the order of the number of flows plus the number of end-hosts. At first sight, this would be equivalent to a network in which each router maintains per-flow state. Fortunately, this is not the case. While the state of a flow is maintained by each router along its path, a trigger is stored at only *one* node at a time. Thus, if there are n triggers and N servers, each server will store n/N triggers on the average. This also suggests that $i3$ can be easily upgraded by simply adding more servers to the network. One interesting point is that these nodes do not need to be placed at specific locations in the network.

4.7 Incremental Deployment

Since $i3$ is designed as an overlay network, $i3$ is incrementally deployable. At the limit, $i3$ may consist of only one node that stores all triggers. Adding more servers to the system does not require any system configuration. A new server simply joins the $i3$ system using the Chord protocol, and becomes automatically responsible for an interval in the identifier space. When triggers with identifiers in that interval are refreshed/inserted they will be stored at the new server. In this way, the addition of a new server is also transparent to the end-hosts.

4.8 Legacy Applications

The packet delivery service implemented by $i3$ is best-effort, which allows existing UDP-based applications to work over $i3$ easily. The end-host runs an $i3$ proxy that translates between the applications' UDP packets and $i3$ packets, and inserts/refreshes triggers on behalf of the applications. The applications do *not* need to be modified, and are unaware of the $i3$ proxy. Packets are intercepted and translated by the $i3$ proxy transparently. As a proof of concept, we have implemented the heterogeneous multicast application presented in Section 3.2 over $i3$. The sender sends a MPEG stream, and one receiver plays back with a MPEG player (mpeg_play) and the other with a H.263 player (tmndec), as shown in Figure 7. We are developing solutions for supporting TCP-based applications over $i3$.

4.9 Security

Unlike IP, where an end-host can only send and receive packets, in $i3$ end-hosts are also responsible for maintaining the routing information through triggers. While this allows flexibility for applications, it also (and unfortunately) creates new opportunities for malicious users. We now discuss several security issues and how $i3$ addresses them.

We emphasize that our main goal here is not to design a bullet proof system. Instead, our goal is to design simple and efficient solutions that make $i3$ not worse and in many cases better than today's Internet. The solutions outlined in this section should be viewed as a starting point towards more sophisticated and better security solutions that we will develop in the future.

An important component for using $i3$ securely is for applications to make a distinction between two types of triggers: *public* and *private*. This distinction is made only at the application level: $i3$ itself doesn't differentiate between private and public triggers.

The identifiers of a public trigger is known by all end-hosts in the system. An example is a web server that maintains a public trigger to allow any client to contact it. A public trigger can be defined as a the hash of the host's DNS name, of a web address, or of the public key associated to a web server. Public triggers are long lived, typically days or months. In contrast, private triggers are chosen by a small number of end-hosts and they are short lived. Typically, private triggers exist only during the duration of a flow. To illustrate the difference between public and private triggers, consider a client accessing a web server. First, the client chooses a private trigger identifier id_c, inserts trigger $(id_s, addr_c)$ into $i3$, and sends id_c to the web server via the server's public trigger. Once contacted, the server selects a private identifier id_s, inserts trigger $(id_s, addr_s)$, and sends id_s to the client via the client's trigger identifier id_c. The client and the server then use the private triggers to communicate. Once the communication terminates, the private triggers are destroyed.

Eavesdropping A user that knows a host's trigger can eavesdrop the traffic towards that host by inserting a trigger with the same identifier and its own address. We consider two cases: (a) private and (b) public triggers.

Private triggers are secretly chosen by the application end-points and are not supposed to be revealed to the outside world. The length of the trigger's identifier makes it very difficult for a third party to use a brute force attack.[7] Furthermore, end-points can periodically change the private triggers associated with a flow. Another alternative would be for the receiver to associate multiple private triggers to the same flow, and the sender to send packets randomly to one of these private triggers. The alternative left to a malicious user is to intercept all private triggers. However this is equivalent to eavesdropping at the IP level or taking control of the $i3$ server storing the trigger, which makes $i3$ no worse than IP.

With $i3$, a public trigger is known by all users in the system, and thus anyone can eavesdrop the traffic to such a trigger. To alleviate this problem, end-hosts can use the public triggers to choose a pair of private triggers, and then use these private triggers to exchange the actual data. To keep the private triggers secret, one can use public key cryptography. To initiate a connection, a host A chooses a private trigger id_a, encrypts it under the public key of a receiver B, and then sends it to B via B's public trigger. B decrypts A's private trigger id_a, then chooses its own private trigger id_b, and sends this trigger back to A over A's private trigger id_a. Since the sender's trigger is encrypted, a malicious user cannot impersonate B.[8]

Trigger hijacking A malicious user can isolate a host by removing its public trigger. Similarly, a malicious user in a multicast group can remove other members from the group by deleting their triggers. While removing a trigger also requires to specify the IP address of the trigger, this address is, in general, not hard to obtain.

One possibility to guard against this attack is to add another level of indirection. Consider a server S that wants to advertise a public trigger with identifier id_p. Instead of inserting the trigger (id_p, S), the server can insert two triggers, (id_p, x) and (x, S), where x is an identifier known only by S. Since a malicious user has to know x in order to remove either of the two triggers, this simple technique provides effective protection against this type of attack. To avoid performance penalties, the receiver can choose x such that both (id_p, x) and (x, S) are stored at the same server. With the current implementation this can be easily achieved by having i_p and x share the same k-bit prefix.

DoS Attacks The fact that $i3$ gives end-hosts control on routing opens new possibilities for DoS attacks. We consider two types of attacks: (a) attacks on end-hosts, and (b) attacks on the infrastructure. In the former case, a malicious user can insert a hierarchy of triggers in which all triggers on the last level point to the victim. Sending a single packet to the trigger at the root of the hierarchy will cause the packet to be replicated and all replicas to be sent to the victim. This way an attacker can mount a large scale DoS attack by simply leveraging the $i3$ infrastructure. In the later case, a malicious user can create trigger loops, for instance by connecting the leaves of a trigger hierarchy to its root. In this case, each packet sent to the root will be exponentially replicated!

To alleviate these attacks, $i3$ uses three techniques:

1. **Challenges** $i3$ assumes implicitly that a trigger that points to an end-host R is inserted by the end-host itself. An $i3$ server can easily verify this assumption by sending a challenge to R the first time the trigger is inserted. The challenge consists of a random nonce that is expected to be returned by the receiver. If the receiver fails to answer the challenge the trigger is removed. As a result an attacker cannot use a hierarchy of triggers to mount a DoS attack (as described above), since the leaf triggers will be removed as soon as the server detects that the victim hasn't inserted them.
2. **Resource allocation** Each server uses Fair Queueing [7] to allocate resources amongst the triggers it stores. This way the damage inflicted by an attacker is only proportional to the number of triggers it maintains. An attacker cannot simply use a hierarchy of triggers with loops to exponentially increase its traffic. As soon as each trigger reaches its fair share the excess packets will be dropped. While this technique doesn't solve the problem, it gives $i3$ time to detect and to eventually break the cycles.
   To increase protection, each server can also put a bound on the number of triggers that can be inserted by a particular end-host. This will preclude a malicious end-host from monopolizing a server's resources.
3. **Loop detection** When a trigger that doesn't point to an IP address is inserted, the server runs a procedure to detect whether the new trigger doesn't create a loop. A simple procedure is to send a special packet with a random nonce. If the packet returns back to the server, the trigger is simply removed. To increase the robustness, the server can invoke this procedure periodically after such a trigger is inserted.

4.10 Anonymity

Point-to-point communication networks such as the Internet provide limited support for anonymity. Packets usually carry the destination and the source addresses, which makes it relatively easy for an eavesdropper to learn the sender and the receiver identities. In contrast, with $i3$, eavesdropping the traffic of a sender will not reveal the identity of the receiver, and eavesdropping the traffic of a receiver will not reveal the sender's identity. The level of anonymity can be further enhanced by using chain of triggers or stack of identifiers to route packets.

5 Simulation Results

In this section, we evaluate several aspects of the $i3$ design by simulation. In particular, we analyze the efficient routing of $i3$. One of the main challenges in providing efficient routing is that end-hosts have little control over the location of their triggers. However, we show that simple heuristics can significantly enhance $i3$'s performance. The metric we use to evaluate these heuristics is the ratio of the inter-node latency on the $i3$ network to the inter-node latency on the underlying IP network. This is called the *latency stretch*.

The simulator is based on the Chord protocol and uses iterative style routing [21]. We assume that node identifiers are randomly distributed. This assumption is consistent with the way the iden-

[7] While other application constraints such as storing a trigger at a server nearby can limit the identifier choice, the identifier is long enough (i.e., 256 bits), such that the application can always reserve a reasonable large number of bits (e.g., 64 bits) that are randomly chosen.

[8] Note that an attacker can still count the number of connection requests to B. However, this information is of very limited use, if any, to the attacker. If, in the future, it turns out that this is a problem, then other security mechanisms such as public trigger authentication will need to be used.

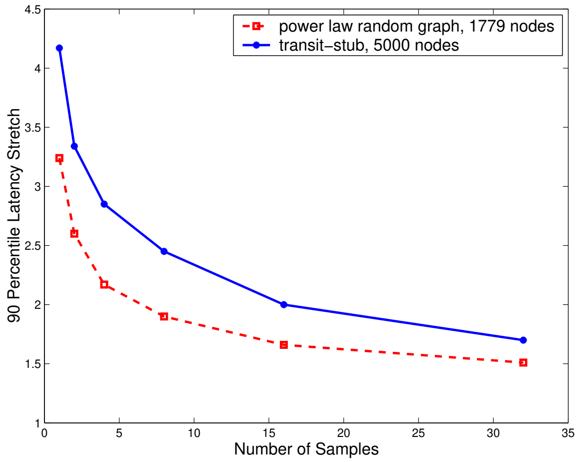


Figure 8: The 90 percentile latency stretch vs. number of samples for PLRG with 1779 nodes and transit-stub with 5000 nodes.

tifiers are chosen in other lookup systems such as CAN [17] and Pastry [18]. As discussed in [21], using random node identifiers increases system robustness and load-balancing. [9] We consider the following network topologies in our simulations:

- A power-law random graph topology with 1779 nodes, where the delay of each link is uniformly distributed in the interval $[5, 100)$ ms. The $i3$ servers are randomly assigned to the network nodes.
- A transit-stub topology generated with the GT-ITM topology generator [11] with 5000 nodes, where link latencies are 100 ms for intra-transit domain links, 10 ms for transit-stub links and 1 ms for intra-stub domain links. $i3$ servers are randomly assigned to stub nodes.

5.1 End-to-End Latency

Consider a source A that sends packets to a receiver R via trigger (id, R). As discussed in Section 4.4, once the first packet reaches the server S storing the trigger (id, R), A caches S and sends all subsequent packets directly to S. As a result, the packets will be routed via IP from A to S and then from S to R. The obvious question is how efficient is routing through S as compared to routing directly from A to R. Section 4.4 presents a simple heuristic in which a receiver R samples the identifier space to find an identifier id_c that is stored at a nearby server. Then R inserts trigger (id_c, R).

Figure 8 plots the 90th percentile latency stretch versus the number of samples k. Each point represents the 90th percentile over 1000 measurements. For each measurement, we randomly choose a sender and a receiver. In each case, the receiver generates k triggers with random identifiers. Among these triggers, the receiver retains the trigger that is stored at the closest server. Then we sum the shortest path latency from the sender to S and from S to the receiver, and divide it by the shortest path latency from the sender to the receiver to obtain the latency stretch. Sampling the space of identifiers greatly lowers the stretch. While increasing the number of samples decreases the stretch further, the improvement appears to saturate rapidly, indicating that in practice, just 16-32 samples should suffice. The receiver does *not* need to search for a close identifier every time a connection is open; in practice, an end-host can sample the space periodically and maintain a pool of identifiers which it can reuse.

[9] We have also experimented with identifiers that have location semantics. In particular, we have used space filling curves, such as the Hilbert curve, to map a d-dimensional geometric space—which was shown to approximate the Internet latency well [16]—onto the one-dimensional Chord identifier space. However, the preliminary results do not show significant gains as compared to the heuristics presented in this section, so we omit their presentation here.

5.2 Proximity Routing in $i3$

Section 5.1 evaluates the end-to-end latency experienced by data packets, after the sender caches the server storing the receiver's trigger t. In this section, we evaluate the latency incurred by the sender's *first* packet that matches trigger t. This packet is routed through the overlay network until it reaches the server storing t. While Chord ensures that the overlay route length is only $O(\log N)$, where N is the number of $i3$ servers, the routing *latency* can be quite large. This is because server identifiers are randomly chosen, and therefore servers close in the identifier space can be very far away in the underlying network. To alleviate this problem, we consider two simple heuristics:

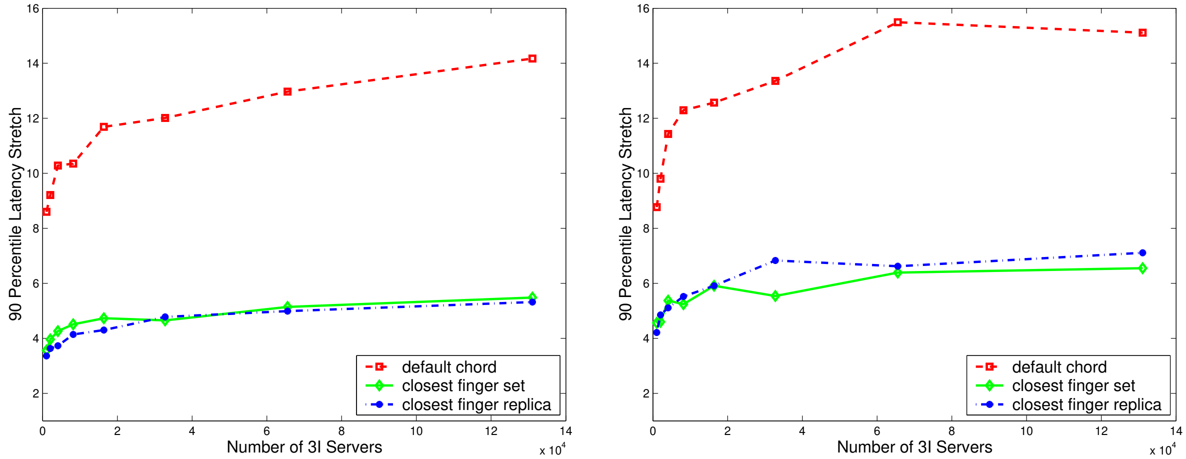
- **Closest finger replica** In addition to each finger, a server maintains $r - 1$ immediate successors of that finger. To route a packet, a server selects the closest node in terms of network distance amongst (1) the finger with the largest identifier preceding the packet's identifier and (2) the $r - 1$ immediate successors of that finger. This heuristic was originally proposed in [5].
- **Closest finger set** Each server s chooses $\log_b N$ fingers as $successor(s + b^i)$, where $(i < \log_b N)$ and $b < 2$. To route a packet, server s considers only the closest $\log_2 N$ fingers in terms of network distances among all fingers.

Figure 9 plots the 90th percentile latency stretch as a function of $i3$'s size for the baseline Chord protocol and the two heuristics. The number of replicas r is 10, and b is chosen such that $\log_b N = r * \log_2 N$. Thus, with both heuristics, a server considers roughly the same number of fingers to perform routing. We vary the number of $i3$ servers from 2^{10} to 2^{16}, and in each case we average routing latencies over 1000 routing queries. In all cases the $i3$ server identifiers are randomly generated.

As shown in Figure 9, both heuristics can reduce the 90th percentile latency stretch up to 2-3 times as compared to the default Chord protocol. In practice, we choose the "closest finger set" heuristic. While this heuristic achieves comparable latency stretch with "closest finger replica", it is easier to implement and does not require to increase the routing table size. The only change in the Chord protocol is to sample the identifier space using base b instead of 2, and store only the closest $\log_2 N$ fingers among the nodes sampled so far.

5.3 Scalable Multicast

In this section, we simulate the scalable multicast algorithm presented in Section 3.4. For simplicity, we consider a single source



(a) (b)

Figure 9: The 90 percentile latency stretch in the case of (a) a power-law random network topology with 1779 nodes, and (b) a transit-stub topology with 5000 nodes. The $i3$ servers are randomly assigned to all nodes in case (a), and only to the stub nodes in case (b).

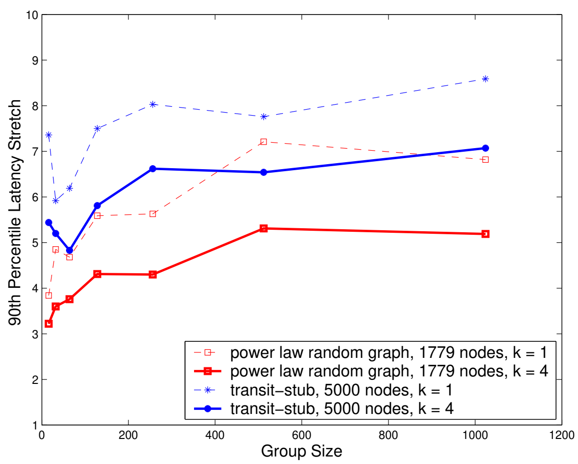


Figure 10: The 90 percentile latency stretch vs. group size for PLRG with 1779 nodes and transit-stub with 5000 nodes.

multicast tree where the source chooses the identifier of the group G such that G maps onto a close server.

Recall that each server puts a limit, th_{addrs}, on the number of triggers with the same identifier that point to end-hosts, and another limit, th_{ids}, on the number of triggers with the same identifier that point to other triggers. In these experiments, we choose $th_{addrs} = 2$, and $th_{ids} = 8$. The multicast identifier tree has degree $h = th_{ids}$. At each level of the identifier tree, a receiver chooses from s random children the one that is stored on the closest server (see Section 3.4).

Figure 10 plots the 90th percentile of the latency stretch for the receivers of a multicast group as the group size varies from 16 to 1024. Each point represents the average of 10 simulation runs. In the case of both topologies, increasing s from 1 to 4 help to reduce the 90th percentile by up to 30%. The cost is that the receiver needs to sample a larger number of triggers when joining the tree.

5.4 Implementation and Experiments

We have implemented a bare-bones version of $i3$ based on the Chord protocol. The control protocol used to maintain the overlay network is fully asynchronous and is implemented on top of UDP. The implementation uses 256 bit ($m = 256$) identifiers and assume that the matching procedure requires exact matching on the 128 most significant bits ($k = 128$). This choice makes it very unlikely that a packet will erroneously match a trigger, and at the same time gives applications up to 128 bits to encode application specific information such as the host location (see Section 2.4.3).

For simplicity, in the current implementation we assume that all triggers that share the first 128 bits are stored on the same server. In theory, this allows us to use any of the proposed lookup algorithms that performs exact matching. Currently, we are developing a version of Chord capable of performing longest prefix matching in a distributed fashion.

Both insert trigger requests and data packets share a common header of 48 bytes. In addition, data packets can carry a stack of up to four triggers (this feature isn't used in the experiments). Triggers need to be updated every 30 seconds or they will expire. The control protocol to maintain the overlay network is minimal. Each server performs stabilization every 30 seconds (see [21]). During every stabilization period all servers generate approximately $N \log N$ control messages. Since in our experiments the number of servers N is in the order of tens, we neglect the overhead due to the control protocol.

The testbed used for all of our experiments was a cluster of Pentium III 700 MHz machines running Linux. We ran tests on systems of up to 32 nodes, with each node running on its own processor. The nodes communicated over a shared 1 Gbps Ethernet. For time measurements, we use the Pentium timestamp counter (TSC). This method gives very accurate wall clock times, but sometime it includes interrupts and context switches as well. For this reason, the high extremes in the data are unreliable.

5.5 Performance

In the section, we present the overhead of the main operations performed by $i3$. Since these results are based on a very preliminary implementation, they should be seen as a proof of feasibility and not as a proof of efficiency. Other Chord related performance metrics such as the route length and system robustness are presented in [5].

Trigger insertion: We consider the overhead of handling an insert trigger request locally, as opposed to forwarding a request to another server. Triggers are maintained in a hash table, so the time is practically independent of the number of triggers. Inserting a trigger involves just a hash table lookup and a memory allocation. The average and the standard deviation over 10,000 of trigger insertion are 12.5 μsec, and 7.12 μsec, respectively. This is mostly the time it takes the operating system to process the packet and to hand it to the application. By comparison, memory allocation time is just 0.25 μsec on the test machine.

Data packet forwarding: Figure 11 plots the overhead of forwarding a data packet to its final destination. This involves looking up the matching trigger and forwarding the packet to its destination addresses. In our experiments, there was never more than one address. In other words, we didn't test multicast. Like trigger insertion, packet forwarding consists of a hash table lookup. In addition, this measurement includes the time to send the data packet. Packet forwarding time, in our experiments, increases roughly linearly with the packet size. This indicates that as packet size increases, memory copy operations and pushing the bits through the network dominate processing time.

$i3$ **routing:** Figure 12 plots the overhead of routing a packet to another $i3$ node. This differs from data packet forwarding in that we route the packet using a node's finger table rather than its trigger table. This occurs when a data packet's trigger is stored on some other node. The most costly operation here is a linear finger table lookup, as evidenced by the graph. There are two reasons for this seemingly poor behavior. First, we augment the finger table with a cache containing the most recent servers that have sent control or data packets. Since in our experiments this cache is large enough to store all servers in the system, the number of nodes used to route a packet (i.e., the fingers plus the cached nodes) increases roughly linearly with the number of nodes in the system. Second, the finger table data structure in our implementation is a list. In a more polished implementation, a more efficient data structure is clearly needed. Performance will benefit greatly as a result.

Throughput: Finally, we ran some experiments to see the maximum rate at which a node can process data packets. Ideally, this should be the inverse of overhead. To test throughput, a single node is bombarded with more packets than it can reasonably handle. We measure the time it takes for $100{,}000$ packets to emerge from the node to determine throughput. Not surprisingly, as packet payload increases, throughput in packets decreases. In addition, we calculate the data throughput from the user perspective. Only the payload data is considered; headers are overhead to users. The user throughput in Mbps increases as the packet payload increases because the overhead for headers and processing is roughly the same for both small and large payloads.

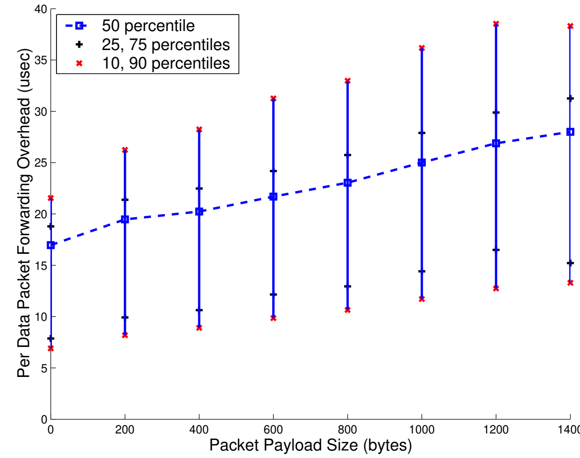


Figure 11: Per packet forwarding overhead as a function of payload packet size. In this case, the $i3$ header size is 48 bytes.

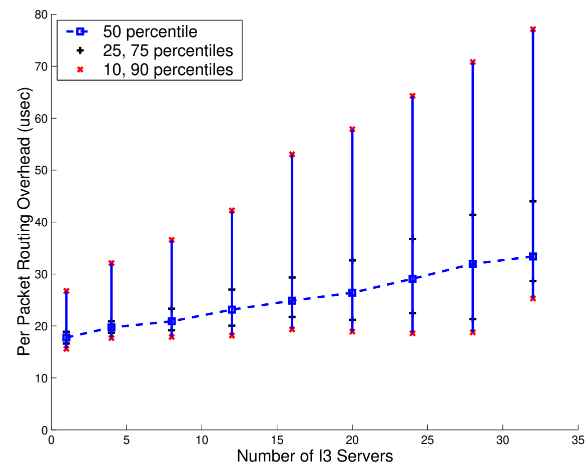


Figure 12: Per packet routing overhead as a function of $i3$ nodes in the system. The packet payload size is zero.

Payload Size (bytes)	Avg. Throughput (std. dev.) (pkts/sec)	Avg. Throughput (payload Mbps)
0	35,753 (2,406)	0
200	33,130 (3,035)	53.00
400	28,511 (1,648)	91.23
600	28,300 (595)	135.84
800	27,842 (1,028)	178.18
1,000	27,060 (1,127)	216.48
1,200	26,164 (1,138)	251.16
1,400	23,339 (1,946)	261.39

Figure 13: The throughput of the data packet forwarding.

6 Related Work

The rendezvous-based communication is similar in spirit to the tuple space work in distributed systems [2, 13, 26]. A tuple space is a shared memory that can be accessed by any node in the system. Nodes communicate by inserting tuples and retrieving them from a tuple space, rather than by point-to-point communication. Tuples are more general than data packets. A tuple consists of arbitrary typed fields and values, while a packet consists of just an identifier and a data payload. In addition, tuples are guranteed to be stored until they are explicitly removed. Unfortunately, the added expressiveness and stronger guarantees of tuple spaces make them very hard to efficiently implement on a large scale. Finally, with tuple spaces, a node has to explicitly ask for each data packet. This interface is not effective for high speed communications.

$i3$ is similar to many naming systems. This should come as no surprise, as identifiers can be viewed as semantic-less names. One future research direction is to use $i3$ as a unifying framework to implement various name systems.

The Domain Name system (DNS) maps hostnames to IP addresses [15]. DNS names are hierarchical while $i3$ identifiers are flat. A DNS name is mapped to an end-host as a result of an explicit request at the beginning of a transfer. In $i3$, the identifier-to-address mapping and the packet forwarding are tightly integrated. DNS resolvers form a static overlay hierarchy, while $i3$ servers form a self-organizing overlay.

Active Names (AN) map a name to a chain of mobile code responsible for locating the remote service, and transporting its response to the destination [23]. The code is executed on names at resolvers. The goals of AN and $i3$ are different. In AN, applications use names to describe what they are looking for, while in $i3$ identifiers are used primary as a way to abstract away the end-host location. Also, while the goal of AN is to support extensibility for wide-area distributed services, the goal of $i3$ is to support basic communication primitives such as multicast and anycast.

The Intentional Naming System (INS) is a resource discovery and service location system for mobile hosts [24]. INS uses an attribute-based language to describe names. Similar to $i3$ identifiers, INS names are inserted and refreshed by the application. INS also implements a late biding mechanism that integrates the name resolution with message forwarding. $i3$ differs from INS in that from the network's point of view, an identifier does not carry any semantics. This simplicity allows for a scalable and efficient implementation. Another difference is that $i3$ allows end-hosts to control the application-level path followed by the packets.

The rendezvous-based abstraction is similar to the IP multicast abstraction [6]. An IP multicast address identifies the receivers of a multicast group in the same way an $i3$ identifier identifies the multicast receivers. However, unlike IP which allocates a special range of addresses (i.e., class D) to multicast, $i3$ does not put any restrictions on the identifier format. This gives $i3$ applications the ability to switch on-the-fly from unicast to multicast. A telephony application can seamlessly change from two to multiple parties if needed. In addition, $i3$ has ability to support multicast groups with heterogeneous receivers.

Estrin *et al.* have proposed an attribute-based data communication mechanism, called direct diffusion, to disseminate data in sensor networks [8]. Data sources and sinks use attributes to identify what information they provide or are interested in. A user that wants to receive data inserts an *interest* into the network in the form of attribute-value pairs. At a high level, attributes are similar to identifiers, and interests are similar to triggers. However, in direct diffusion, the attributes have a much richer semantic and the rules can be much more complex than in $i3$. At the implementation level, in direct diffusion, nodes flood the interests to their neighbors, while $i3$ uses a lookup service to store the triggers determined based on the trigger identifier.

Two new approaches, TRIAD [3] and IPNL [9], have been recently proposed to solve the IPv4 address scarcity problem. Both schemes use DNS names rather than addresses for global identification. However, TRIAD and IPNL make different tradeoffs. While TRIAD is more general by allowing an unlimited number of arbitrarily connected IP network realms, IPNL provides more efficient routing by assuming a hierarchical topology with a single "middle realm". Packet forwarding in both TRIAD and IPNL is similar to packet forwarding based on identifier stacks in $i3$. However, while with TRIAD and IPNL the realm-to-realm path of a packet is determined during the DNS name resolution by network specific protocols, with $i3$ the path is determined by end-hosts.

Multi-Protocol Label Switching (MPLS) was recently proposed to speed-up the IP route lookup and to perform route pinning [1]. Similar to $i3$, each packet carries a stack of labels that specifies the packet route. The first label in the stack specifies the next hop. Before forwarding a packet, a router replaces the label at the head of the stack. There are several key differences between $i3$ and MPLS. While $i3$ identifiers have global meaning, labels have only local meaning. In addition, MPLS requires special protocols to choose and distribute the labels. In contrast, with $i3$ identifier stacks are chosen and maintained by end-hosts.

7 Discussion and Future Work

While we firmly believe in the fundamental purpose of $i3$ – providing a general-purpose indirection service through a single overlay infrastructure –the details of our design are preliminary. Besides exploring the security and efficiency issues mentioned in the paper further, there are areas that deserve significant additional attention.

For inexact matching, we have used longest-prefix match. Inexact matching occurs locally, on a single node, so one could use any reasonably efficient matching procedure. The question is what inexact matching procedure will best allow applications to choose among several candidate choices. This must work for choosing based on feature sets (*e.g.*, selecting printers), location (*e.g.*, selecting servers), and policy considerations (*e.g.*, automatically directing users to facilities that match their credentials). We chose longest-prefix match mostly for convenience and familiarity, and it seems to work in the examples we've investigated, but there may be far superior options.

Our initial design decision was to use semanticless identifiers and routing; that is, identifiers are chosen randomly and routing is done based on those identifiers. Instead, one could embed location semantics into the node identifiers. This may increase the efficiency of routing, by allowing routes to take lower-latency $i3$-level hops, but at the cost of making the overlay harder to deploy, manage, and load-balance.

While these design decisions are important, they may have little to do with whether $i3$ is ever deployed. We don't know what the economic model of $i3$ would be and whether its most likely deployment would be as a single provider for-profit service (like content distribution networks), or a multiprovider for-profit service (like ISPs), or a cooperatively managed nonprofit infrastructure. While deployment is always hard to achieve, $i3$ has the advantage that it can be incrementally deployed (it could even start as a single, centrally located server!). Moreover, it does not require the cooperation of ISPs, so third-parties can more easily provide this service. Nonetheless, $i3$ faces significant hurdles before ever being deployed.

8 Summary

Indirection plays a fundamental role in providing solutions for mobility, anycast and multicast in the Internet. In this paper we propose a new abstraction that unifies these solutions. In particular, we propose to augment the point-to-point communication abstraction with a rendezvous-based communication abstraction.

To demonstrate the feasibility of this approach, we have built an overlay network based on the Chord lookup system. Preliminary experience with $i3$ suggests that the system is highly flexible and can support relatively sophisticated applications that require mobility, multicast, and/or anycast. In particular, we have developed a simple heterogeneous multicast application in which MPEG video traffic is transcoded on the fly to H.263 format. Currently we are developing two other applications: providing transparent mobility to legacy applications, and a large scale reliable multicast.

References

[1] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan. A framework for multiprotocol label switching, Nov. 1997. Internet Draft, draft-ietf-mpls-framework-02.txt.

[2] N. Carriero. *The Implementation of Tuple Space Machines*. PhD thesis, Yale University, 1987.

[3] D. R. Cheriton and M. Gritter. TRIAD: A new next generation Internet architecture, Mar. 2000. http://www-dsg.stanford.edu/triad/ triad.ps.gz.

[4] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS'00*, pages 1–12, Santa Clara, CA, June 2000.

[5] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proc. ACM SOSP'01*, pages 202–215, Banff, Canada, 2001.

[6] S. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, May 1990.

[7] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Journal of Internetworking Research and Experience*, pages 3–26, Oct. 1990. (Also in Proceedings of ACM SIGCOMM'89, pages 3-12).

[8] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of ACM/IEEE MOBICOM'99*, Cambridge, MA, Aug. 1999.

[9] P. Francis and R. Gummadi. IPNL: A nat extended internet architecture. In *Proc. ACM SIGCOMM'01*, pages 69–80, San Diego, 2001.

[10] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The ninja architecture for robust internet-scale systems and services, 2000.

[11] Georgia tech internet topology model. `http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html`.

[12] H. Holbrook and D. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *Proceedings of ACM SIGCOMM'99*, Cambridge, Massachusetts, Aug. 1999.

[13] Java Spaces. http://www.javaspaces.homestead.com/.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

[15] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proc. ACM SIGCOMM*, Stanford, CA, 1988.

[16] T. S. E. Ng and H. Zhang. Towards global network positioning. In *ACM SIGCOMM Internet Measurements Workshop 2001*, San Francisco, CA, Nov. 2001.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, San Diego, 2001.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Nov. 2001.

[19] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of ACM/IEEE MOBICOM'99*, Cambridge, MA, Aug. 1999.

[20] A. C. Snoeren, H. Balakrishnana, and M. F. Kaashoek. Reconsidering internet mobility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM'01*, pages 149–160, San Diego, 2001.

[22] I. Stoica, T. Ng, and H. Zhang. REUNITE: A recursive unicast approach to multicast. In *Proceedings of INFOCOM'00*, Tel-Aviv, Israel, Mar. 2000.

[23] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active names: Flexible location and transport. In *Proceedings of USENIX Symposium on Internet Technologies & Systems*, Oct. 1999.

[24] W. Adjie-Winoto and E. Schwartz and H. Balakrishnan and J. Lilley. The design and implementation of an intentional naming system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, Dec. 1999.

[25] WAP wireless markup language specification (WML). http://www.oasis-open.org/cover/wap-wml.html.

[26] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM System Journal*, 37(3):454–474, 1998.

[27] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.