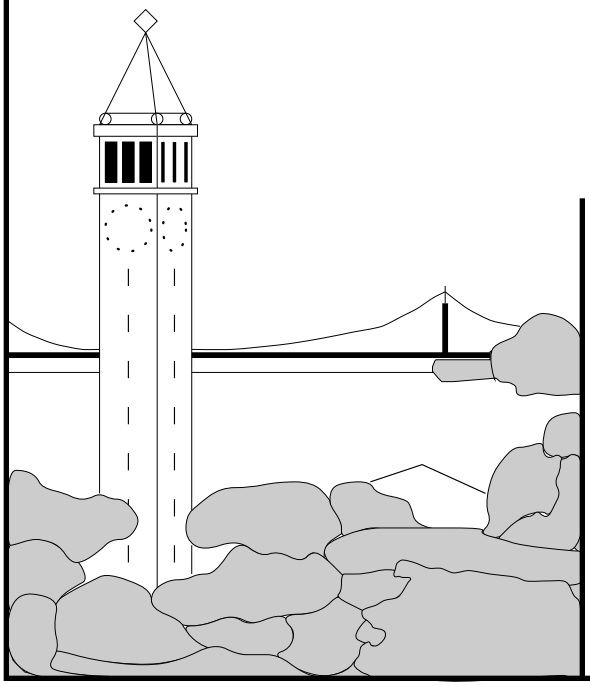
Flexible and Robust Large Scale Multicast using $i3$

Karthik Lakshminarayanan *Ananth Rao* *Ion Stoica* *Scott Shenker*
UC Berkeley *ICSI*



Report No. UCB/CSD-02-1187

June 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Flexible and Robust Large Scale Multicast using $i3$

Karthik Lakshminarayanan Ananth Rao Ion Stoica Scott Shenker
UC Berkeley ICSI

June 2002

Abstract

There have been many proposals to support multicast in the Internet. These proposals can be roughly categorized as being either infrastructure-based, with the multicast functionality provided by designated network nodes, or host-based, with the multicast functionality provided by the members of the multicast group itself. These two classes have very different performance characteristics; typically infrastructure-based solutions are far more scalable, while the host-based solutions are far more deployable and flexible.

This paper proposes a multicast architecture, that is a hybrid of the two approaches, based on the Internet Indirection Infrastructure ($i3$). $i3$ provides a general-purpose rendezvous primitive that end hosts can use to implement multicast in a scalable, flexible, and deployable manner. To demonstrate the feasibility of this approach, we have designed and implemented a scalable solution for multicast, and then extended it to provide reliable data delivery. To evaluate our design we perform extensive simulations, and experiments on two test-beds: a PC cluster, and a small size Internet-wide test-bed consisting of 13 end-hosts. From simulations, we found that the 90th percentile latency stretch for 65,536 receivers is less than 5. Also, for 4,096 receivers, no more than 1.5 duplicates were generated for each lost packet.

1 Introduction

The original Internet architecture was designed to provide *unicast* point-to-point communication, and it has proved tremendously successful at doing so. However, efficiently reaching many users simultaneously with high-bandwidth streams (such as required for real-time video and audio) requires *multicast* functionality. The last decade has seen a myriad of multicast proposals, none of which are in widespread use today.[1] These proposed multicast designs can be roughly classified into infrastructure-based and host-based solutions. Infrastructure-based solutions implement multicast functionality in a set of designated network nodes that are responsible for constructing the multicast tree and replicating multicast packets at the branch points in that tree. Examples of infrastructure-based solutions are IP multicast [10] which implements the multicast functionality at the IP layer (and so the designated multicast nodes are all network routers), and application-level solutions such as Overcast [18] and Fastforward [2], which use a set of special servers for the designated multicast nodes.

In contrast, the host-based multicast designs, such as Narada [8] and NICE application multicast protocol [4], require no support from any designated network nodes. Instead, the multicast functionality is implemented entirely by the collection of end-hosts participating in the multicast group.

These approaches have very different performance characteristics. The infrastructure-based approaches, through their use of designated nodes spread throughout the network, can typically build more efficient multicast trees and so are more scalable in terms of numbers of users. On the other hand, host-based solutions do not involve any additional infrastructure support and are hence much easier to deploy and can more easily add new functionality; in short, they are more deployable and flexible.

In this paper, we propose a hybrid solution that combines features of both approaches and in doing so, provides high degree of scalability, flexilibity and deployability. The key idea in our approach is to rely on the Internet Indirection Infrastucture ($i3$). As described in Section 2, $i3$ provides a very basic and general-purpose rendezvous primitive that has a wide range of applications. We then propose that the end-hosts implement the rest of the multicast functionality, using the $i3$ rendezvous primitive as the building block. This $i3$ primitive enables the end-hosts to create efficient multicast trees, and at the same time gives the end-hosts flexibility to implement additional features such as reliability. Thus, the resulting multicast solution is more scalable than current end-host based solutions and more flexible and deployable than current infrastructure-based approaches.

Following [17], the general design philosophy advocated here is that the infrastructure should not be augmented to implement additional particular services, such as multicast, but rather the infrastructure should support minimal, and more general-purpose primitives that end hosts can then use to construct a variety of services. The challenge is to find primitives that overcome the main impediments to scalability, but yet are sufficiently general to provide flexibility. We believe that the $i3$ primitives are one such example, and this paper is partially a proof-by-example of how these primitives can be

[1] IP multicast is offered in several commercial routers, but multicast service is not made available to users by most ISPs.

$i3$'s Application Programming Interface (API)	
send_packet(p)	send packet
insert_trigger(t)	insert trigger
remove_trigger(t)	remove trigger

(a)

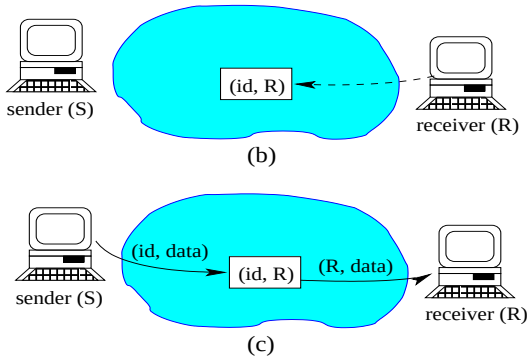


Figure 1: (a) $i3$'s API. Example illustrating communication between two nodes: (b) The receiver R inserts trigger (id, R). (c) The sender sends packet $(id, data)$.

flexibly, yet scalably, used.

The remainder of the paper is organized as follows. Section 2 gives an overview of $i3$. Section 3 presents our scalable multicast solution, and Section 4 presents a simple and efficient solution to provide reliability. Sections 5 and 6 evaluate our proposed solution by simulations an experiments. Finally, Section 7 presents the related work, and Section 8 concludes the paper.

2 Background

In this section we present a brief overview of an Internet Indirection Infrastructure, $i3$ [26], which forms the foundation for our multicast solution. 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. Packet delivery is best-effort like in today's Internet.

2.1 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[2] and $data$ is the payload (typically a normal IP packet payload). Receivers use *triggers* to indicate their interest in packets. In their simplest form, triggers are pairs $(id, addr)$, where id is the trigger identifier, and $addr$ is a node's address, consisting of an IP address and UDP port number. A trigger $(id, addr)$ indicates that all packets sent to identifier id should be forwarded (at the IP layer) by the $i3$ infrastructure to the node with address $addr$. More specifically, the rendezvous-based communication abstraction exports the three primitives 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, 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 and enables them to be oblivious to the other's location. However, unlike IP multicast, hosts in $i3$ are free to place their triggers. As we will see, this allows end-hosts to construct a scalable multicast service.

2.2 Implementation Overview

$i3$ is implemented as 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.

To maintain this overlay network and to route packets in $i3$, we use the Chord lookup protocol [9]. Chord assumes a circular identifier space of integers $[0, 2^m)$, where 0 follows $2^m - 1$. Every $i3$ server has an identifier in this space, and all trigger identifiers belong to the same identifier space. The $i3$ server with identifier n is responsible for all identifiers in the interval $(n_p, n]$, where n_p is the identifier of the node preceding n on the identifier circle. Figure 3(a) shows an identifier circle for $m = 6$. There are five $i3$ servers in the system with identifiers 5, 16, 24, 36, and 50, respectively. All identifiers in the range (5, 16] are mapped on server 16, identifiers in (17, 24] are mapped on server 24, and so on.

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. At that $i3$ node, it is matched against (any) triggers for that id and forwarded (using IP) to all hosts interested in packets sent to that identifier. Chord ensures that the server responsible for a identifier is found after visiting at most $O(\log n)$ other $i3$ servers irrespective of the starting server (n represents the total number of servers in the system). To achieve this, Chord requires each node to maintain only $O(\log n)$ routing state. Chord allows servers to leave and join dynamically, and it is highly robust against failures. For more details refer to [9]. Figure 3(b) shows an example in which trigger $(30, R)$ is inserted at node 36 (i.e., the node that maps $(24, 36]$, and is thus responsible for identifier 30). Packet $(30, data)$ is forwarded to server 30, matched against trigger $(30, R)$, and then forwarded via IP to R.

[2] In the implementation presented in this paper, we use $m = 256$. Such a large value of m allows end hosts to choose trigger identifiers independently since the chance of collision is extremely small. In addition, a large m makes it very hard for an attacker to guess a particular trigger identifier.

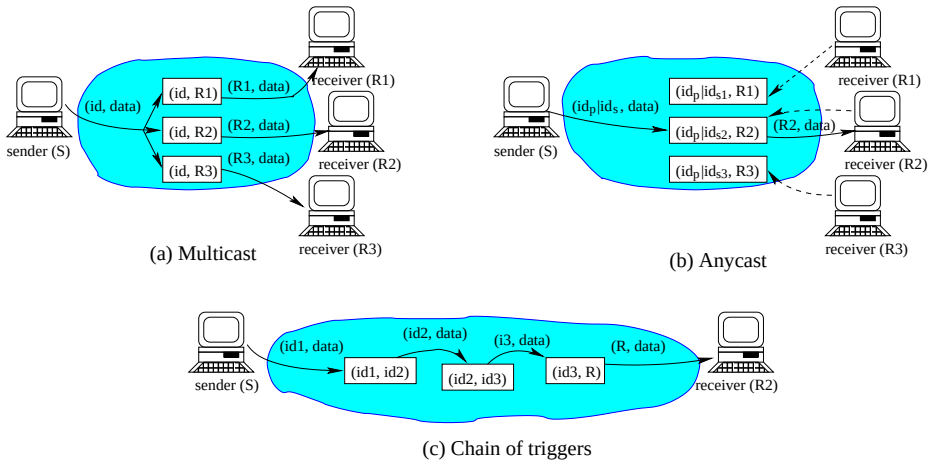


Figure 2: Basic communication primitives provided by $i3$. (a) Multicast: Every packet $(id, data)$ is forwarded to each receiver R_i that inserts the trigger (id, R_i). (b) Anycast: The packet matches the trigger of receiver $R2$. $id_p|id_s$ denotes an identifier of size m, where id_p represents the prefix of the k most significant bits, and id_s represents the suffix of the $m-k$ least significant bits. (c) Trigger chains: Replace the receiver address in the second field of a trigger with another trigger identifier.

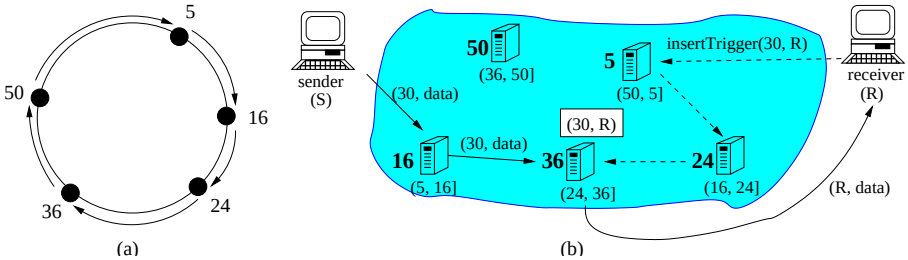


Figure 3: (a) A Chord identifier circle for $m = 6$, with 5 servers identified by 5, 16, 24, 36, and 50, respectively. Each server is responsible for all identifiers between its identifier and the identifier of the node that precedes it on the circle. (b) Receiver R inserts trigger $(30, R)$, and the trigger is forwarded via $i3$ to server 36 which is responsible for identifier 30. The trigger is stored there (shown in the white box) until explicitly removed or timed out. When sender S sends packet $(30, data)$, it is also forwarded via $i3$ to server 36. Servers identifiers are in bold. The interval of identifiers for which each server is responsible are also shown.

Note that packets are not stored in $i3$; they are only forwarded. End hosts use periodic refreshing to maintain their triggers in $i3$. Hosts need only know one $i3$ node to use the $i3$ infrastructure. This can be done through a static configuration file, or by a DNS lookup assuming $i3$ is associated with a DNS domain name. In Figure 3(b), the sender knows only server 16, and the receiver knows only server 5.

One important observation is that once the end host finds the server that is responsible for a specific identifier, it can cache that server's address. The end-host can then refresh the triggers and send packets with the same identifiers directly to that server. Thus only the first few packets of each flow incur the Chord routing overhead.

2.3 Basic Communication Primitives

Small Scale Multicast (Packet Replication): 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 2(a)). Since all triggers with the same 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 present our solution to make the multicast scalable in Section 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 2(b)).

Trigger Chains: $i3$ allows end-hosts to chain multiple triggers by replacing the receiver address of the second field of a trigger with the identifier of another trigger. Figure 2(c) shows an example of a chain of triggers of length three. Note that replacing a trigger (id_1, R) with a a chain of triggers (id_1, id_2), (id_2, id_3), and (id_3, R) is transparent to the end-hosts.

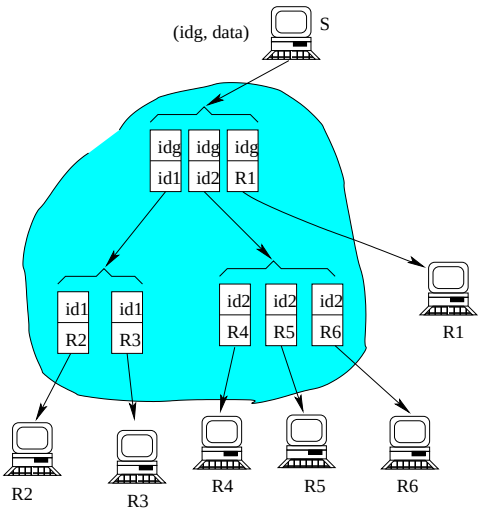


Figure 4: Example of a scalable multicast tree with bounded degree by using chains of triggers.

3 Scalable Multicast Protocol

In this section, we present an algorithm to build a *scalable* multicast tree, and in the process demonstrate the flexibility of the basic multicast and anycast primitives provided by $i3$. Section 4 presents an extension of the algorithm to achieve reliable data delivery. For simplicity, in our discussion we assume a single source tree. Section 3.2.2 extends this model to multiple sources.

As discussed in Section 2.3, storing an arbitrary number of triggers with the same identifier causes scalability problems since the $i3$ server that stores these triggers must send a replica for every incoming packet that matches these triggers. To get around this problem, we assume that each $i3$ server puts a limit D on the number of triggers with the same identifier it stores. In other words, D represents the maximum number of receivers in a multicast group that are allowed by the basic $i3$ multicast primitive. This is the reason why the native multicast support that $i3$ provides does not scale beyond D receivers.

The key idea to achieve scalability 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 4 shows an example of multicast tree with 7 receivers where $D = 3$.

```
// node n_new joins the multicast tree with sender S
join(S, n_new)
  best_dist = ∞
  n_curr = S
  do
    // return the closest node to n_new from jset(n_curr)
    n = select_node(n_new, jset(n_curr))
    if (best_dist > dist(S, n, n_new))
      best_dist = dist(S, n, n_new)
      n_join = n
    if (fset(n_curr) = ∅)
      break
    n_curr = select_node(n_new, fset(n_curr))
  while (dist(S, n_curr, n_new) < best_dist)
  join_at(n_new, n_join) // n_new joins at n_join
```

Figure 5: Joining algorithm.

Next, we present our algorithm to build such a multicast tree. For the clarity of exposition, we divide our presentation into two parts: Section 3.1 presents a simple tree building algorithm that abstracts away the interaction of the end-hosts with the infrastructure, while Section 3.2 presents an efficient implementation of this algorithm in $i3$.

3.1 Basic Tree Building Algorithm

In this section, we present a basic tree building algorithm that abstracts away the interactions between end-hosts and $i3$. In particular, our goal is to build a multicast tree of bounded degree D which exhibits low latency from source to each receiver. Figure 6(a) shows a multicast tree with $D = 3$ consisting of six receivers and one source S, and one new receiver, R_7, that wants to join the multicast group.

We say that a node in the multicast tree is *joinable* if its out-degree is less than D. Otherwise, if the node's out-degree is D, we say that the node is *full*. Let S denote the source of the multicast tree, let n denote a joinable node already in the tree, and let n_{new} denote a new node that wants to join the multicast tree. Assume n_{new} joins the multicast tree at n. Then, let $dist(S, n, n_{new})$ be the latency experienced by a multicast packet from source S to n_{new} via n. In particular, $dist(S, n, n_{new})$ represents the latency from source S to n (via the multicast tree) plus the IP latency from n to n_{new}.

The goal of the joining procedure is to find a joinable node n that provides a short distance path from S to n_{new}. Let $dist(S, n, n_{new})$ denote this distance. To achieve this we use a "branch-and-bound" algorithm that starts from source

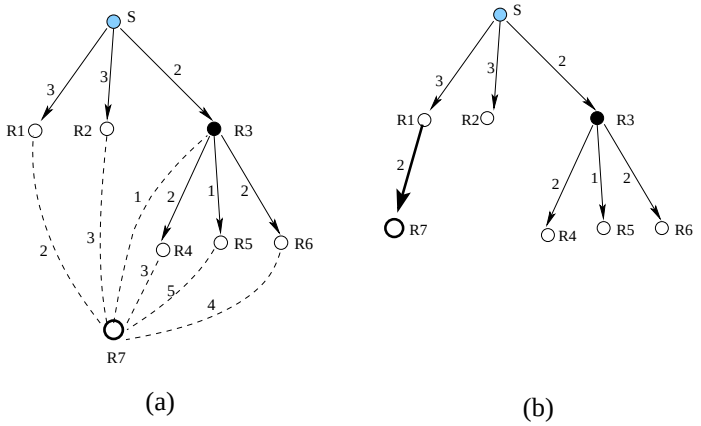


Figure 6: (a) R_7 wants to join an existing multicast tree. Joinable nodes are represented by empty circles; full nodes are represented by black circles. (b) The resulting tree obtained by running the algorithm in Figure 5.

and goes down the tree until it can no longer improve the distance from S to n_{new} or until it reaches the leaves of the tree. Figure 5 shows the pseudocode of the joining procedure. Let $jset(n)$ denote the set of joinable children of n, and let $fset(n)$ denote the set of full children of n. The algorithm maintains a variable n_{curr} that is initialized to source S, and a variable $best\_dist$ that represents the best known latency from S to n_{new} via one of the nodes visited so far. At each iteration, the algorithm selects node n from $jset(n_{curr})$ that minimizes the distance from S to n_{new}, $dist(S, n, n_{new})$. If this distance improves $best\_dist$, the algorithm updates $best\_dist$. Similarly, the algorithm selects the node n' from $fset(n_{curr})$ that minimizes $dist(S, n', n_{new})$. If this distance doesn't improve $best\_dist$ the algorithm terminates, and n_{new} joins the tree at the node corresponding to $best\_dist$. Otherwise, the algorithm sets n_{curr} to n', and iterates.

Figure 6 shows a simple example in which a new receiver, R_7, joins an existing multicast tree with six receivers. The number along each edge represents the latency associated to that edge. At the first level, there are two joinable nodes $jset(S) = \{R_1, R_2\}$, and one full node $jfull(S) = \{R_3\}$. Among the joinable nodes, the algorithm selects R_1 since $dist(S, R_1, R_7) = 5$ is larger than $dist(S, R_2, R_7) = 6$. Next, the algorithm selects the full node R_3 and iterates. Among the children of R_3, the algorithm selects node R_4, as $dist(S, R_4, R_7) < dist(S, R_5, R_7) = dist(S, R_6, R_7)$. Finally, since $dist(S, R_1, R_7) < dist(S, R_4, R_7)$ the algorithm terminates and R_7 joins at R_1.

3.2 Scalable Multicast Protocol

In this section we show how the end-host multicast protocols described in the previous section is implemented in $i3$.

Figure 7(a) shows a possible hierarchy of triggers that corresponds to the multicast tree in Figure 6(a). Indeed, if we collapse all triggers with the same identifiers, and each receiver R_i with its trigger (id_i, R_i), then the tree in Figure 7(a) degenerates into the end-host multicast tree in Figure 5(a). A new node R_k joins the multicast group at an identifier id by inserting two triggers (id, id_k) and (id_k, R_k), where id_k is an identifier located at an $i3$ server close to R_k. For example, in Figure 7(b), R_7 joins at identifier id_1 by inserting triggers (id_1, id_7) and (id_7, R_7), respectively. Since each trigger (id_i, R_i) is assumed to be located on a server close to receiver R_i, in the remainder of this section we neglect the latency between R_i and id_i. In particular we assume that the latency from S to R_7 is equal to the latency from S to R_1 plus the latency from R_1 to R_7. To remain consistent with the notation used in the previous section we will use $dist(S, R_1, R_7)$ and $dist(S, id_1, R_7)$ interchangeably to denote the latency from S to R_7.

The main challenge to implement the pseudocode shown in Figure 5 in $i3$ is to efficiently implement $dist()$ and $select\_node()$ functions. To address these challenges, we use two techniques, (1) $i3$ multicast to implement $select\_node()$, and (2) a simple scheme based on local time-stamps to compute the relative value of $dist()$. Since the pseudocode in Figure 5 uses the results returned by $dist()$ only for comparison proposes, computing a relative instead of an absolute value for $dist()$ is good enough.

Figure 8 shows the pseudocode of the joining procedure in $i3$. To implement $select\_node()$, we use the $i3$ multicast primitive. In particular all receivers in a set (e.g., *jset* or *fset*) will subscribe to a unique $i3$ multicast group. Consider a receiver R that is connected through triggers (id', id) and (id, R) to the multicast tree. To maintain the one-to-one mapping between the multicast trees in Figures 5(a) and 7(a), we slightly change the definition of *joinable* and *full* nodes to take into account the fact that a receiver that joins at an internal node in the multicast tree requires an additional trigger. In particular, we say that receiver R is *joinable* if there are less

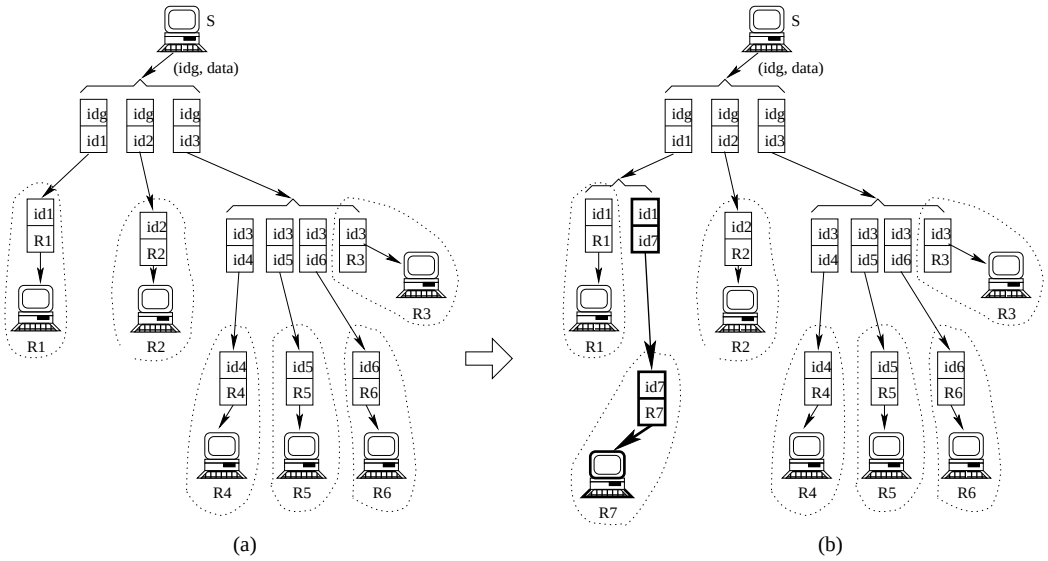


Figure 7: The hierarchy of triggers corresponding to the join operation shown in Figure 6.

than $D+1$ triggers with identifier id, and *full* if there are exactly $D+1$ triggers with identifier id in the system. Node R_3 in Figure 7(a) is a full node since there are four triggers with identifier i_3 in the system. In contrast, all other receivers are joinable. Let $jHash$ and $fHash$ be two well-known hash functions. If R is joinable, R will simply insert the trigger $(jHash(id'), R)$ in $i3$. Thus, $jHash(id')$ identifies the *jset* to which R belongs. Otherwise, if R is full, it will join the corresponding *fset* by inserting the trigger $(fHash(id'), R)$. For example, receivers R_1 and R_2 will join the *jset* identified by $jHash(id_g)$, while R_3, which is a full node, will join the *fset* identified by $fHash(id_g)$.

When node R_{new} wants to find the end-host R in a set (identified by) id_a that minimizes the distance from S to R_{new}, it simply sends a request message with identifier id_a. In turn, each receiver R that receives this message (i.e., each receiver in the set id_a) will send a reply to R_{new} along with the local time-stamp, plus the difference between the time-stamps at S and R associated to a previous message sent from S to R.[3] Upon receiving these reply messages, R uses the local time-stamp in addition to the timing information carried by the reply messages to find node R that minimizes $dist(S, R, R_{new})$ (see the $on\_receiving\_query\_distance()$ function).

3.2.1 Comparing Source-Receiver Distances

In this section, we show that it is possible to compare the distances from S to R_{new} via different intermediate nodes using local time-stamps alone, and without assuming global clock synchronization. In particular, we compute $dist(S, R, R_{new})$ as a function of local time-stamps and the clock skews, and then show that the clock skews cancel out when performing distance comparisons.

Let $d(A, B)$ denote the latency between nodes A and B. Then, we have

$$dist(S, R, R_{new}) = d(S, R) + d(R, R_{new}). \quad (1)$$

Let $p.depart(A)$ denote the *local* time at A when packet p departs from A, and let $p.arrive(B)$ denote the local time at B when packet p arrives at B. Then, we have

$$d(S, R) = p.arrive(R) - p.depart(S) + skew(S, R), \quad (2)$$

where p is a packet that travels from S to R and $skew(S, R)$ denotes the clock skew between S and R. Similarly,

$$d(R, R_{new}) = p'.arrive(R_{new}) - p'.depart(R) + skew(R, R_{new}), \quad (3)$$

where p' is a packet traveling from R to R_{new}. By combining Eqs. (1)-(3), we obtain

$$dist(S, R, R_{new}) = (p.arrive(R) - p.depart(S)) + (p'.arrive(R_{new}) - p'.depart(R)) + skew(S, R_{new}), \quad (4)$$

where we replaced $skew(S, R) + skew(R, R_{new})$ by $skew(S, R_{new})$. Note that the value $(p.arrive(R) - p.depart(S))$ is computed by R upon the arrival of packet p from S, and sent to R_{new} in the reply message as $r.src\_dist$ (see $on\_receiving\_packet\_from\_src()$ and $on\_receiving\_query\_dist()$ functions).

[3] To decrease the variance, in practice we send an average of the differences between R's and S's local time-stamps.

```
// join the multicast tree idg
join(idg)
  best_dist = ∞
  id_curr = idg
  do
    // return reply form selected node in jset(id_curr)
    (id, dist) = select_node(jHash(id_curr))
    if (best_dist > dist)
      best_dist = dist
      id_join = id
    (id, dist) = select_node(fHash(id_curr))
    if (id = NULL) break
    id_curr = id
  while (dist < best_dist)
  join_at(id_join) // node joins at id_join

join_at(id_join)
  close_id = pick_close_id()
  insert_trigger(close_id, my_address)
  insert_trigger(id_join, close_id)

select_node(set_id)
  dist = ∞
  p_set = query_distance(set_id)
  foreach (p ∈ p_set)
    d = curr_time − p.time_stamp + p.src_dist
    if (d < dist)
      dist = d
      id = p.id
  return (id, dist)

on_receiving_query_distance(p)
  r.id = my_trigger_id
  r.time_stamp = curr_time
  r.src_dist = src_dist
  send_pkt(p.requester, r)

on_receiving_pkt_from_src(p)
  src_dist = curr_time − p.time_stamp
```

Figure 8: Joining multicast protocol in $i3$.

The important point to note in Eq. (4) is that $skew(S, R_{new})$ does *not* depend on R. This allows us to use the sum $(p.arrive(R) - p.depart(S)) + (p'.arrive(R_{new}) - p'.depart(R))$ instead of the absolute value of $dist(S, R, R_{new})$ to *compare* the distances between S and R_{new} via any intermediate node R.

3.2.2 Other Issues

So far, we have discussed the case of a single source sending data to the multicast tree. However, this is not a fundamental restriction. As long as a member of the multicast group knows the group identifier id_g, it can send data to the group. The only cost incurred by this simple scheme is that packets originated at sources farther away from the id_g location, will experience higher latencies; the scalability and the robustness of our protocol are not affected.

Our join procedure assumes that at each level there is at least a non empty *jset* or *fset*. However, in time all receivers at one level may leave which will cause these sets to become empty. To alleviate this problem, each receiver periodically probes the parent level and checks if it can join at the parent level. We can also make sure that this process causes negligible increase in control traffic by limiting the periodicity of probes, say once every 10 seconds in practice.

Finally, each receiver needs to decide whether it is joinable or full in order to decide what set to join. We describe a simple scheme to address this problem next. Consider a receiver R connected to the hierarchy through trigger (id, R). Then, R can periodically try to insert a dummy trigger (id, x). If it is successful, this means that id is joinable and R joins *jset*. R also immediately removes the trigger (id, x). If not, this means that there are already D triggers with the identifier id in the system, and as a result R joins *fset*.

3.3 Trigger Refreshing

Recall that $i3$ uses a soft state approach to maintain triggers in the system. If a trigger is not refreshed for a pre-defined period of time T (which in our implementation is 30 sec), the trigger is removed from the system. The question we address in this section is how to keep the triggers in the hierarchy alive without compromising the scalability of our protocol.

A simple solution would be that each receiver refreshes all triggers on the path from the source to itself. For example, in Figure 7, R_6 would be responsible for refreshing triggers (id_g, id_3), (id_3, id_5), and (id_5, R_5), respectively. The main problem with this approach is that the number of refreshes received by a trigger increases exponentially as we move up in the hierarchy. In particular, a trigger will be refreshed by all receivers covered by the sub-tree rooted at that trigger. For example, trigger (id_g, id_3) will be refreshed by receivers R_3, R_4, R_5, and R_6, respectively.

We use two techniques to address this problem. First, we introduce a new control message, denoted REFRESH_ACK, to suppress the redundant refresh messages. When receiver R refreshes trigger (id_i, id_j) it also sends a REFRESH_ACK message to id_j. Since this message will be multicast in the entire sub-tree rooted at id_j, it will eventually reach all receivers covered by this sub-tree. Upon receiving a REFRESH_ACK message for a given trigger, a receiver simply restarts the timer to refresh that trigger.

The above technique alone is not enough to reduce the number of refresh messages. Indeed, if all receivers refresh a given trigger at the same time, REFRESH_ACK will do nothing to suppress any of these refreshes. The classic solution to address this problem is to randomize refresh timers. In particular, a receiver chooses a random timer that is uniform distributed in the interval $[0, T)$. Let n be the number of receivers in the sub-tree rooted at trigger t, and let RTT denote the round-trip time between a receiver in this sub-tree

and trigger t (for simplicity assume that all receivers have the same RTT). Then, the expected number of refresh messages received by trigger t during a refresh period is brought down to $n \times RTT/T$ from n. This is because it takes RTT from the moment the first refresh is sent to t until all receivers in the sub-tree learns about it (via the associated REFRESH_ACK message), and the probability that a refresh timer will trigger during this time interval is RTT/T.[4] To quantify the savings, if $T = 30$ sec and $RTT = 100$ ms, this simple technique can reduce the number of refreshes by two order of magnitude. While this represents a significant reduction for most practical cases, it might not be enough for very large multicast groups with millions of receivers.

To further improve the algorithm, we choose timers such that a trigger will be almost always be refreshed by the receivers at the first and second levels, and the refreshes of receivers at the lower levels will almost always be suppressed. In particular, each receiver chooses the timer associated with a trigger that is one or two levels up the tree with respect to itself in the interval $(\alpha T, \beta T)$, and the timers associated to the triggers that are higher than two levels in the hierarchy in the interval $(\beta T, T)$.

In practice, we choose $\alpha = 1/2$ and $\beta = 3/4$. Simulation results in Section 5.2 show that this simple scheme results in a significant reduction of the number of redundant refreshes, and more importantly, the number of redundant refreshes is virtually independent of the tree size.

So far we have implicitly assumed that the refreshes are no lost. To accommodate message losses we decrease the timer intervals by a factor k. In practice, we choose $k = 3$.

4 Reliable Multicast

Developing reliable multicast solutions has proven a difficult and challenging problem. Even when the assumptions allow changes in the infrastructure, the resulting solutions are complex and exhibit undesirable tradeoffs. For instance, with SRM, there is a clear tradeoff between the number of duplicates and the time to recover from failure.

This section demonstrates the flexibility of our multicast architecture, by presenting a solution for reliable multicast that is both simple and scalable. To reduce the number of duplicate packets, our solution leverages the ability to multicast a packet to a sub-tree in the trigger hierarchy. For instance, in the topology in Figure 7(a) one can multicast a packet only to receivers R_4, R_5, and R_6 by sending the packet to identifier id_3. In addition, to avoid the NACK implosion problem our solution leverages the anycast capability offered by $i3$.

Figure 9 shows the pseudocode of the recovery procedure.

[4]With timer chosen uniformly at random in the interval $[0, T)$, the expected number of refreshes is in fact $2 \times n \times RTT/T$ as the expected number of refreshes in each refresh period is two in the absence of suppression.

```
// function called on packet loss
on_packet_loss(seq_num)
  // send packet to anycast group at my level to respond
  // to loss of seq_num, reply to be sent to my_addr
  request_repair(repair_group[my_level], seq_num, my_addr, TO)

request_repair(anycast_id, seq_num, req_id, to)
  send_repair_req(anycast_id, seq_num, req_id)
  set_timer(curr_time + to, request_repair,
            anycast_id, seq_num, req_id, to)

on_receiving_repair_req(q)
  // repair request already received ?
  if (pending_repair_req[q.seq_num] = FALSE)
    // get repair packet
    if (r = get_packet(q.seq_num, packet_queue))
      send_repair(r, q.req_id)
    else
      // doesn't have the repair packet; send request up the tree
      request_repair(repair_group[my_level − 1],
                     q.seq_num, parent_id, TO)
      pending_repair_req[q.seq_num] = TRUE

on_receiving_repair(r)
  pending_repair_req[r.seq_num] = FALSE
  remove_timer(request_repair, seq_num)
```

Figure 9: Pseudocode of the recovery procedure.

The main idea is to associate a recovery *anycast* group with each internal node in the hierarchy. Each receiver subscribes to exactly one anycast group, that is, the anycast group associated with its parent. In the example in Figure 6(a) there will be two recovery anycast groups: one associated with the source S consisting of receivers R_1, R_2, and R_3, and another one associated with receiver R_3 consisting of receivers R_4, R_5, and R_6, respectively.

We assume that all data packets have a unique sequence number and that losses are detected when packets arrive out of sequence.[5] When a receiver R detects a packet loss, it sends a repair request for the lost packet to its anycast group. Upon receiving a repair request, a receiver R_a checks whether if it has the requested packet, and if yes, sends the packet directly to the requester (R) via IP unicast. If not, R_a assumes that everyone in its anycast group has lost the packet, and it takes the responsibility for recovering the lost packet by sending a repair request to its parent's anycast group. If the recipient of the repair request at the higher level has the packet, it sends the packet to everybody in the subtree rooted at the node corresponding to the requester's anycast group. If the packet is absent at the higher level, the recovery procedure proceeds recursively.

To illustrate the repair procedure consider the multicast tree

[5]In practice, we may allow a certain number of out of sequence packets before concluding that a packet was lost. This would allow us to tolerate packet reordering.

in Figure7(a). Assume receiver R_4 loses a packet. As a result it sends a repair request to the anycast group consisting of nodes R_4, R_5, and R_6, respectively. Assume this repair request is delivered to R_6. If R_6 has the packet, it sends it directly to node R_4, and we are done. If not, R_6 forwards the repair request to its parent anycast group, that is, to the anycast group consisting of R_1, R_2, and R_3. Assume the repair request is delivered to R_2 and that R_2 has the requested packet. Then R_2 will send a repair to $id3$. As a result, the repair will be multicast to all receivers in the sub-tree routed at $id3$ including R_4 and R_6.

One remainder question is how to construct the anycast identifiers. Consider a receiver R which is connected to the multicast tree by triggers (id_{i-1}, id_i) and (id_i, R), respectively. Then R_i will insert a trigger (id_a, R) where $id_a = H_r(id_{i-1})|loss\_rate(R)$. The sign "|" represents the concatenation symbol. $H_r()$ is a well-known hash function that returns 128 bit values, and $loss\_rate(R)$ represents R's loss rate as measured by R.

When a receiver R loses a packet, it sends a repair request with identifier $id_{req} = H_r(id_{i-1})|0$. Since $i3$ uses the longest prefix matching scheme to match the packet and the trigger identifiers (see Section 2.3), the repair request will be delivered to the sibling of R which experiences the lowest loss rate, that is, to the receiver that is most likely to have received the requested packet.

One potential problem with this scheme is that when the packet is lost at a higher level of the hierarchy, *all* receivers belonging to the same recovery anycast group at a lower level will send repair request messages. Let R_a be the receiver in the anycast group that experiences the lowest loss rate. Then, in the worst case, R_a receives up to $2 \times D$ repair request messages, where D is the maximum out-degree of the multicast tree: D repair requests from all member of R_a's anycast group (including R_a itself), and D repair requests from the level immediately below. Next, we present two simple solutions to alleviate this problem.

With the first solution, a receiver that detects a packet loss waits for a random period of time, uniformly distributed in the interval $[0, T)$, before sending a repair request for the missing packet. Assume a packet is lost at a higher level. Let t_0 denote the time when the first repair request for that packet is sent, and let $t_0 + \Delta$ be the time when the repair arrives at end-hosts which lost the packet. Then all repair requests that were scheduled to be sent after time $t_0 + \Delta$ will be suppressed. As a result, the number of repair requests delivered to the receiver with the lowest loss rate from a recovery anycast group decreases from $2 \times D$ to roughly $2 \times D \times \Delta/T$. This is because the probability that a receiver (which has lost the packet) to send a repair request during a time interval of length Δ is Δ/T. On the downside, this solution increases the time to receive the repair. If only one receiver losses a packet, it will take the receiver an additional $T/2$ time to receive the repair on average.[6]

So far we have assumed that the repair requests are always delivered to the member of the anycast group which experiences the lowest loss rate. The idea of the second solution is to spread the responsibility to answer the repair requests among the members of the repair anycast group. To achieve this we redefine the anycast identifier of receiver R as $id_a = H_r(id_{i-1})|H_r(IP_R)$, where IP_R represents the IP address of receiver R. When a receiver R loses the packet with the sequence number seq_n, it sends a repair request with identifier $id_{req} = H_r(id_{i-1})|H_r(seq\_no)$. This way, all repair requests for the same packet are delivered to the same receiver, while repair requests for different packets are delivered to different receivers, thus achieving load balancing. The downside of this approach is that the receiver to which the repair requests are delivered may have poor reception characteristics. Section 5.3 uses simulation experiments to compare these two solutions.

5 Simulations

In this section, we evaluate our algorithm using simulation. Specifically, our results focus on (1) the efficiency of tree construction, (2) scalability of the trigger refresh mechanism, and (3) the efficiency and scalability of the recovery mechanism. In Section 6, we give a preliminary evaluation of our prototype implementation on a cluster of servers and in a small wide-area testbed.

Our simulator is based on the $i3$ protocol described in [26]. In our simulations, we consider a 10,000-node transit stub topology generated using the GT-ITM topology generator [15].[7] Link latencies are randomly chosen between 1 and 3 ms for intra-transit domain links, between 5 and 10 ms for transit-stub links, and between 20 and 50 ms for inter-transit links. $i3$ servers and multicast members are randomly attached to stub nodes. In all experiments, we assume 8192 $i3$ servers, and that each multicast member knows a set of identifiers that maps on the closest $i3$ server. These identifiers can be discovered by using offline sampling [26] or through service discovery mechanisms.

To test the recovery procedure, we assign random loss rates uniformly distributed between 0-4% to $i3$ hops, and between 0-8% to the hops from $i3$ servers to end-hosts. Both data and control packets (e.g. refresh messages) are dropped with the same probability. Since our main goal is to study the scalability and the efficiency of our approach, we do not consider $i3$ server failure. However, we notice that $i3$ server failures will be transparent to the end-hosts modulo the time it takes end-hosts to refresh the triggers that were stored at the failed

[6] The additional $T/2$ represents the estimated delay of sending the repair request.

[7] We have also ran simulations using power-law network topologies, and obtained very similar results.

servers.[8] Finally, since each end-host is responsible of refreshing the entire chain of triggers from source to itself, end-host failures will have no effect on the trigger hierarchy.

5.1 Tree construction

To test the scalability and the efficiency of the tree construction algorithm described in Section 3, we ran a number of simulations varying the multicast group size from 16 to 65536. In our evaluation, we use four metrics, which were previously introduced in [8, 6]:

Latency Stretch, the ratio of the latency from the source to receiver in $i3$ to the latency along the shortest path (IP latency).

Ratio of the Maximum Delay (RMD), the ratio between the maximum delay using $i3$ multicast and the maximum delay using IP routing.

Ratio of the Average Delay (RAD), the ratio between the average delay using $i3$ multicast and the average delay using IP routing.

Link stress, the number of copies of each multicast packet that travels through that link.

Figure 10 plots the 90^{th} percentile latency stretch, the RAD, and the RMD between the source and the receivers. As expected, the stretch increases only logarithmically with the size of the network. In addition, these results indicate that the resulting multicast trees could meet the latency requirements of a large number of applications. Indeed, even for a group of size 65536, the RMD and RAD do not exceed 3.5, and the 90^{th} percentile of the latency stretch does not exceed 5.

Implementing multicast at the application layer results in multiple copies of a packet being sent on the same physical link. Figure 11 plots the maximum stress observed on any link. While the maximum stress increases with the group size, we note that this increase is sub-linear and that even for a group of size 2^{16}, it does not exceed 130. Still this value is much larger than the out-degree of the multicast tree, which in our case is 8. The reason for this disparity is that triggers with different identifiers end up on the same $i3$ server. This is equivalent to multiple internal nodes in the multicast tree mapping on the same $i3$ server. If k different trigger identifiers are stored at the same server, the stress on the server's outgoing link can be as high as $k \times D$. As the size of the group increases the probability that more than one trigger identifier is stored at an $i3$ server increases considerably. Recall, that there are only 8192 $i3$ servers, while the size of the multicast group can be as high as 65536.

[8] All triggers stored at a failed server will be instantiated next time they are refreshed by end-hosts. Note that, to increase the robustness, one can assume trigger replication at the level of the $i3$ infrastructure [26].

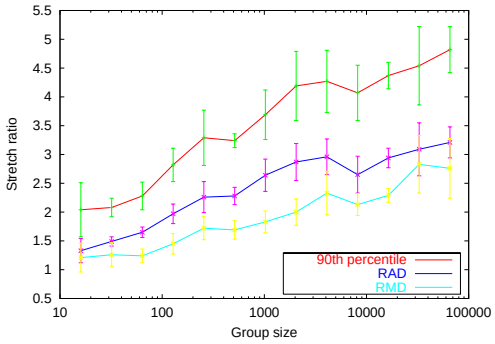


Figure 10: The latency stretch of the multicast tree

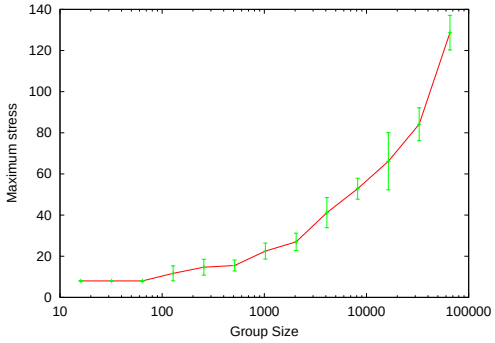


Figure 11: The maximum stress caused by the overlay multicast tree on the links of the underlying network

5.2 Scalable trigger refreshing

In this section, we evaluate the refresh schemes described in Section 3.3. With the first scheme, called *refresh-unif*, each receiver refreshes each trigger in its chain by using timers uniformly distributed in the interval $[T/2, T)$, where T is the refresh period for triggers in $i3$. In contrast, with the second scheme, called *refresh-level*, a receiver uses timers uniformly distributed in the interval $[T/2, 3T/4)$ to refresh the triggers one and two levels above it in the hierarchy, and timers uniformly distributed in the interval $[3T/4, T)$ to refresh all the other triggers in its chain.

Figure 12 plots the maximum and the average number of refreshes per trigger during a time interval of length T, for both schemes as a function of the multicast group size. As expected, the *refresh-level* scheme reduces the maximum number of refreshes per trigger significantly. Moreover, the difference increases as the size of the group increases. This is because, as explained in Section 3.3, with *refresh-level* the refreshes of the receivers placed at more than two levels down the hierarchy are suppressed with a very high probability. In contrast, the average number of refreshes are roughly the same for both schemes (i.e., about 1.5), with *refresh-unif* performing slightly better. This is because receivers near a trigger are more aggressive in the case of *refresh-level* (their

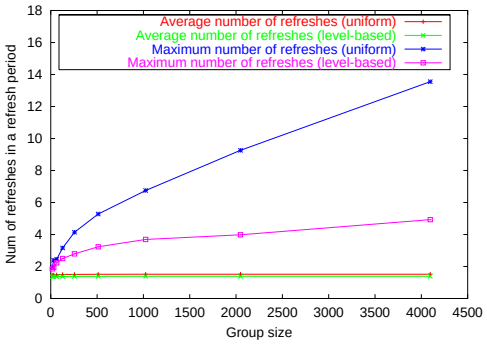


Figure 12: Maximum and average number of refreshes per refresh period

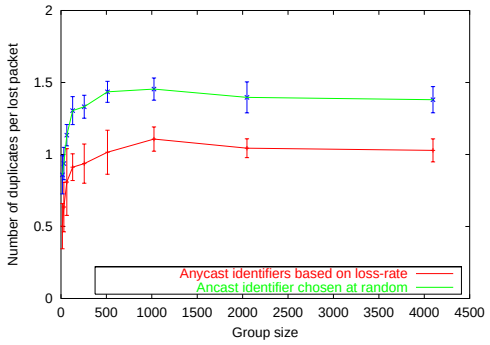


Figure 13: Total number of duplicates as a fraction of total number of drops.

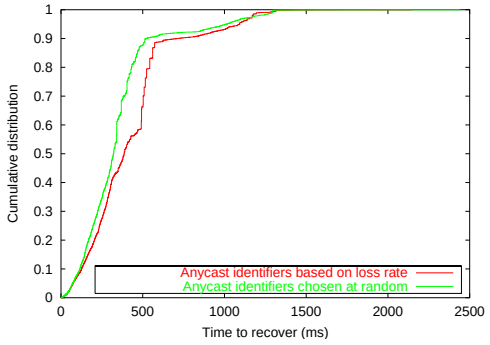


Figure 14: Cumulative distribution of time to recover for 1024 receivers.

timers are three times smaller on average), which results in a slightly larger number of refreshes.

5.3 Reliability

In this section, we evaluate the recovery procedure described in Section 4. As an optimization, upon detecting a packet loss, a receiver sends a repair request after a time period randomly chosen between 0 and 300 ms. This optimization decreases the chance of a receiver sending a repair request before the repair data triggered by another repair request is received.

The timeout for re-sending a repair request is 800 ms. We evaluate the two schemes presented in Section 4: *rel-loss*, where the repair request is delivered to the receiver which experiences the lowest loss rate, and *rel-random*, where the repair request is delivered to a random receiver within the same recovery anycast group. In our evaluation, we consider three metrics: (1) the number of packet duplicates per packet loss, (2) the time it takes to receive the repair, and (3) the number of repair requests received by an end-host.

Figure 13 plots the number of packet duplicates per packet loss. Ideally, this value should be zero. However, due to the fact that repairs might be multicast, receivers who haven't lost the packet would receive duplicates. As shown in Figure 13 both schemes perform well; in either case the number of duplicates per packet loss is less than 1.5[9] and practially remains constant at large group sizes. However, as expected, the *rel-loss* perform slightly better. This is because, the receiver that gets the repair requests is the one that experiences lowest loss rate and is hence more likely to have the packet.

Figure 14 plots the cumulative distribution function (CDF) of the recovery time, that is, the time it takes to receive a repair from the moment the loss was detected. There are two things worth noting. First, in 85% of the cases, the repair arrives within 600 ms, thus obviating the need to send a second repair request. Second, in our experiments, all repairs arrive before 2400 ms, so a third repair request is almost never needed.

Finally, Table 1 gives the *average* number of repair requests per packet loss, and the *maximum* number of repair requests per sequence number received by an end-host for both schemes. In terms of the average number of repair requests, both schemes perform similarly, with each of them generating about 1.2 repair requests per packet loss. The reason for which *rel-loss* performs slightly better is because the end-hosts receiving the repair requests are more likely to have the lost packet, which reduces the chance to forward the request at the higher level. On the other hand, in terms of maximum number of repair requests per packet loss, *rel-random* performs much better. In particular the maximum number of repair requests per packet loss is about five times larger in the case of *rel-loss* for 4096 receivers. This is again because, with *rel-loss*, the repair requests are delivered to the same receiver, while with *rel-random* repair requests are spread among different receivers based on their sequence number.

In summary, while *rel-loss* reduces the number of duplicate packets and the total number of repair requests, *rel-random* is more effective in avoiding hotspots due to repair requests being delivered to the same receiver.

[9]For comparison, in SRM [13], if only one receiver loses a packet, all receivers would receive a duplicate packet.

Group size	# repairs per drop (*rel-loss*)	# repairs per drop (*rel-random*)	max repairs per seq # (*rel-loss*)	max repairs per seq # (*rel-random*)
16	1.12 (0.12)	1.15 (0.16)	1.00 (0.07)	0.18 (0.03)
64	1.07 (0.16)	1.11 (0.12)	1.92 (0.28)	0.31 (0.04)
256	1.16 (0.05)	1.18 (0.12)	2.87 (0.25)	0.44 (0.06)
1024	1.18 (0.08)	1.19 (0.07)	3.89 (0.24)	0.64 (0.05)
4096	1.13 (0.04)	1.16 (0.070)	5.09 (0.54)	0.84 (0.08)

Table 1: Number of repairs as a fraction of drop rate, and maximum number of anycast requests per sequence number for rel-random and rel-loss

Group size	Depth of the tree
8	2 (0)
16	2.8 (0.13)
32	3.4 (0.22)
48	3.9 (0.21)
64	4.5 (0.22)

(a)

Level	# of messages
1	55.9 (2.19)
2	10.1 (0.62)
3	2.36 (0.68)
4	0.88 (0.40)
5	0.066 (0.03)

(b)

Table 2: Results from experiments on Millennium showing the depth of the tree and the number of $query\_distance$ messages processed (standard deviations in parentheses).

6 Experiments

We have implemented a prototype of the multicast protocol described in Section 3 on top of $i3$ [26].[10]

To test our protocol, we conduct experiments on two platforms: Millennium, a large cluster of PCs at UC Berkeley [1], and a small wide-area test-bed consisting of thirteen end-hosts. While the experiments on Millennium are aimed at testing the implementation for medium size groups, the second set of experiments demonstrate the performance of our protocol in a more realistic scenario.

6.1 Millennium Experiments

Our experiments on the Millennium test-bed involve 32 $i3$ servers and 64 multicast clients. To test our implementation for trees with multiple levels, we use $D = 4$ instead of $D = 8$ (as used in simulations).

Table 2 (a) shows the depth of the resulting multicast tree as function of the group size. As expected, the depth of the tree is roughly logarithmic in the group size. In particular, the depth of the tree for 64 clients is 4.5 which is only 50% larger than the depth of a perfectly balanced tree with 64 nodes.

To evaluate the overhead of the joining procedure, in Table 2 (b), we give the number of control messages (i.e., $query\_distance$ messages in Figure 8) processed by a receiver as a function of its level in the tree. The number of messages increases exponentially with the height of the receiver in the tree. This is because the joining procedure starts always from the root, and, as a result, receivers at the first level will be contacted by every new receiver joining the tree. Since our current implementation takes about 200 μs to process a $query\_distance$ message, we can support a few thousands of new clients joining every second. While this number is large enough for most practical applications, for very large groups the processing of $query\_distance$ messages can still be a bottleneck.

[10] At this point, the current implementation does not include the recovery mechanism presented in Section 4.

6.2 Distributed Test-bed Experiments

To evaluate our protocol in a more realistic scenario, we use a wide-area network test-bed consisting of thirteen end-hosts at various location in the Internet as shown in Figure 15. The end-hosts use a variety of access technologies: two of the end-hosts have cable modem access, one has ADSL access, two are outside continental US (Germany and Australia), and the rest are located at US universities and are connected to Internet2. Figure 15 also shows two representative trees built by our algorithm.

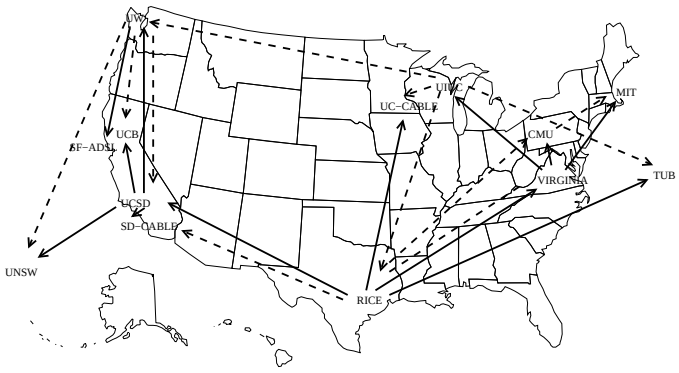


Figure 15: The various hosts used in the wide area simulations. Two instances of the multicast tree built are also shown, one (source at RICE) using solid lines and the other (source at UIUC) with dashed lines.

In each experiment, we run an $i3$ server on each host, and then instantiate one sender and twelve receivers in random order. Note that since each end-host runs an $i3$ server, each receiver will pick a trigger located on the same end-host. In each experiment, we wait until the multicast tree is fully constructed.

Figure 16(a) plots the cumulative distribution function (CDF) of the latency stretch over 20 experiments. As can be observed, in 85% of cases the stretch is less than two, that is, the latency in the multicast tree is only twice as worse as the IP latency.

To provide more insight into the relationship between the

multicast latencies and IP latencies, Figure 16(b) plots the latencies in the multicast tree versus the IP latencies for each (sender, receiver) pair. The points above the line correspond to a stretch larger than one, while the points below the line correspond to a stretch smaller than one. The main reason for which we observe several point corresponding to a stretch lower than one is not due to IP routing inefficiencies, but due to the variability of our measurements to estimate network distances taken a few minutes apart.

7 Related work and Discussion

As discussed in the introduction, most current multicast proposals can be roughly categorized as either infrastructure-based or host-based. Among the infrastructure-based solutions are IP level multicast solutions such as IP Multicast [10], CBT [3], PIM [12] and EXPRESS [16] and application-level (or overlay-network) solutions such as Overcast [18], Scattercast [7] Yoid [14], and ALMI [22]. While these approaches all have significant differences from each other, they have a high degree of scalability and a low degree of flexibility. In response, a new class of solutions that implement the multicast functionality exclusively at end-hosts has emerged. Among these solutions are Narada [8], Peercast [11] and the NICE application level multicast protocol [4]. While these solutions are highly flexible, they tend to suffer from scalability and robustness problems.

However, there are other approaches that are worth mentioning. Recent advances in peer-to-peer lookup and routing protocols [23, 27, 25, 30] had led to the development of efficient multicast solutions [24, 6, 31, 19]. While these solutions are highly scalable, it is unclear how easy it is to add new functionality on top of the basic multicast protocol.

In addition, there have been many attempts to provide reliable multicast functionality. SRM [13] adds support for reliability by multicasting the repair requests and retransmitting the data. While this solution (and others such as TMTP [29]) address the problem of NACK implosion, the number of duplicates can be very high. This is because repairs are multicast to all members of the group. LMS [21] presents a finer grained recovery service by creating a dynamic hierarchy of servers. Similarly, STORM [28] uses a self organizing overlay network to provide recovery. However, these solutions are quite complex since they require the application to build and maintain an overlay network for recovery purposes only. In addition, they are based on IP multicast which limits their deployability. [11]

Jannotti *et al* [17] have proposed a multicast approach that is quite close in spirit (but very different in detail) from what we have proposed. This work shows that various network services, including multicast, can be scalably implemented by adding router support for two simple primitives. The first primitive *path reflection* allows end-hosts to request redirection and replication at nearby routers, while the second primitive, *path painting* allows end-hosts to find the intersection point of their paths to a common destination. An interesting and open question is whether path painting and selection primitives could be used to more efficiently implement $i3$ primitives.

Fundamentally, this paper is based on two principles. First, we should not focus solely on particular communication abstractions like multicast, but instead should recognize that there will be a plethora of desired communication abstractions. Second, to achieve the flexibility needed to support these other communication abstractions, we should only embed a few basic general-purpose primitives into the infrastructure. In this paper, we have chosen the $i3$ primitives as the candidates, and illustrated their applicability to implementing scalable and reliable multicast. However, we view this as merely an initial stage in the search for, and evaluation of, these basic communication primitives.

8 Conclusions and Future Work

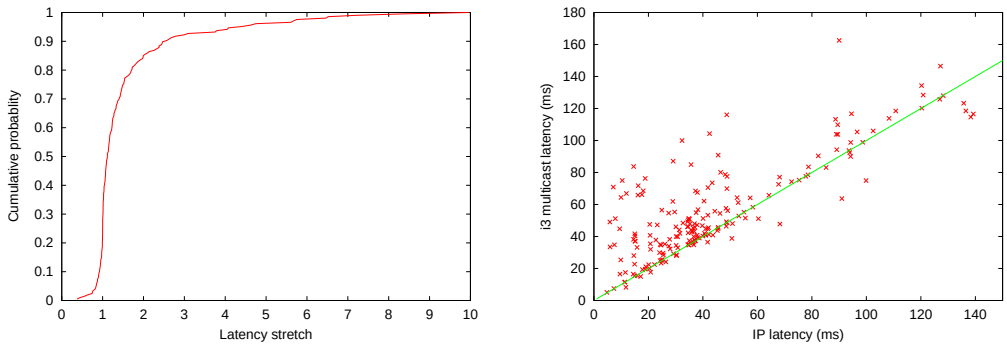
In this paper we propose a multicast solution that is scalable, flexible, and incrementally deployable. The key idea of our approach is to carefully split the functionality between the infrastructure and end-hosts. We rely on the Internet Indirection Infrastructure ($i3$), which provides a very basic set of basic communication primitives (see Section 2.3). These $i3$ primitives enable the end-hosts to create efficient multicast trees, and at the same time gives the end-hosts flexibility to implement additional features such as reliability. To demonstrate the feasibility of our approach, we have implemented a system prototype and performed experiments on two test-beds: a PC cluster, and a small size Internet-wide test-bed consisting of 13 end-hosts. The preliminary results suggest that our protocol is indeed scalable and efficient.

However, much more remains to be done. At the implementation level, we plan to evaluate the reliability mechanism in a wide area network (so far, we have evaluated this mechanism only by simulation). At the protocol level, we plan to experiment with providing the functionality as a third party service. In particular, a third party can take responsibility for building and maintaining the hierarchy of triggers, thus relieving the end-hosts of this burden. We also observe that we can leverage this approach to effectively provide access control. The provider of the service can keep the group identifiers and all the triggers that are part of the hierarchy a secret and thus prevent arbitrary senders from sending to the identifiers.

[11] Digital Fountain [5], WEBRC, and RLM [20] provide reliability and/or congestion control through coding. Since these schemes would work on any multicast protocol that does best effort delivery of data, this can be deployed on top of our multicast protocol.

References

[1] `http://www.millennium.berkeley.edu`.
[2] Fastforward networks. `http://www.ffnet.com`.



(a) (b)

Figure 16: (a) CDF of the stress from the experiments and (b) the relationship between IP latency and multicast latency.

[3] A. J. Ballardie, P. F. Francis, and J. Crowcroft. Core based trees. In *Proc. of ACM SIGCOMM'93*, pages 85–95, San Francisco, 1993.

[4] S. Banerjee, B. Bhattacharjee, and S. Parthasarathy. A protocol for scalable application level multicast, 2001. CS-TR 4278, University of Maryland, College Park.

[5] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[6] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlay networks, unpublished work, 2002.

[7] Y. Chawathe, S. McCanne, and E. Brewer. An architecture for internet content distribution as an infrastructure service, 2000.

[8] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS'00*.

[9] 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.

[10] S. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, May 1990.

[11] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over a peer-to-peer network, 2001.

[12] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast – sparse mode (pim-sm) : Protocol specifi cation, Jun. 1997. RFC-2117.

[13] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[14] P. Francis. Yoid: Extending the internet multicast architecture, 2000.

[15] Georgia Tech Internet Topology Model. `http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html`.

[16] 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.

[17] J. Jannotti. Network layer support for overlay networks.

[18] 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 OSDI 2000*, San Diego, California, October 2000.

[19] J. Liebeherr and M. Nahas. Application-layer multicast with delaunay triangulations. In *IEEE Globecom*, 2001.

[20] S. McCanne. *Scalable Compression and Transmission of Internet Multicast Video*. PhD thesis, UC Berkeley, Dec. 1996. UCB/CSD-96-928.

[21] C. Papadopoulos, G. M. Parulkar, and G. Varghese. An error control scheme for large-scale multicast applications. In *Symposium on Principles of Distributed Computing*, page 310, 1998.

[22] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure, 2001.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001*.

[24] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Networked Group Communication*, pages 14–29, 2001.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001*.

[26] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *To appear in SIGCOMM, 2002*.

[27] 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.

[28] X. Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient multicast support for continuous-media applications. 1997.

[29] R. Yavatkar, J. Griffoen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *ACM Multimedia*, pages 333–344, 1995.

[30] 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.

[31] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV, 2001*, 2001.