Joint Design of Multicast Loss Recovery and Forwarding Service Extensions

Adam M. Costello
http://www.cs.berkeley.edu/~amc/

1999-May-05

Abstract

IP multicast is an efficient mechanism for sending a packet to multiple recipients because the network can use its knowledge of topology to prevent the packet from traversing the same link many times. But packet losses generally affect only a subset of the group members. If loss recovery is required, the efficiency of multicast is not available, because sending retransmissions to the entire group does not scale to large groups, and the network provides no other means of leveraging its knowledge of the group's topology.

This limitation can be overcome by extending the multicast service model with additional forwarding services defined in terms of the underlying multicast distribution tree. We and others have proposed loss recovery schemes that use different forwarding services in different ways, raising the question of which forwarding mechanisms are most worthwhile to support in the network, and how a given set of them can best be used in the end hosts to satisfy the loss recovery requirements of a given application.

To solve this problem, we have devised a framework that serves as a roadmap of the design space. At the network layer, in place of the assortment of forwarding services that have been proposed, such as *subcast* (forward to a subtree), *parentcast* (forward to one's parent member), and *randomcast* (forward to a random nearby member), we propose a new generalization called *treecast*, which unifies and supersedes all the others. At the end hosts, where protocols are built on top of the forwarding services, we have designed a software architecture in which different design options (like how to build a hierarchy, or how to choose between unicast and subcast repairs) are handled by separate components. Swapping out one component while keeping the others the same allows fair comparisons between protocols, and mixing and matching components generates new ones. By building this framework in a simulation environment and later as an application-level testbed with treecast over UDP, we will explore the design space and evaluate the costs and performance of treecast-based loss recovery schemes.

1 Introduction

The Internet Protocol (IP) provides two forwarding services—*unicast*, for sending a data packet from a source to a single destination, and *multicast* [2], for sending to a group of destinations. Packets are relayed from router to router along links inside the network. If unicast were used to send the same data to many destinations, identical packets would traverse the links and routers near the source many times. For one-to-many communication, multicast is more efficient because a single copy of the packet traverses each link—routers duplicate incoming packets onto multiple outgoing links, so that multicast packets travel along a *distribution tree* connecting the source and group members.

Like unicast, the multicast service is *best-effort*, meaning packets can be lost along the way (usually because of congestion). For some applications, like audio and video, data loss is tolerable, and merely results in degraded quality. But many applications (like news articles and whiteboards) require all the data to arrive. For unicast data, *loss recovery* is relatively simple: the sink sends feedback to the source about which data have or have not arrived, and the source sends repair packets containing the lost data. For multicast, loss recovery is more complicated. A packet lost at some point on the distribution tree fails to reach the members in the *loss subtree* downstream of that point. Two obvious but naïve solutions, having the source unicast a repair to each member who experienced the loss or multicasting a repair to the entire group, both suffer performance degradation as the group size grows.[1]

Two basic techniques have been described to address this problem using only unicast and multicast. One is to organize the group members into a *hierarchy* in which children send feedback to their parents, and parents send repairs to their children, as in TMTP [17] and RMTP [9]. The other is to form *local groups* [3]—additional multi-

[1] The use of forward error correction (FEC) techniques allows one repair to recover multiple losses, which reduces the inefficiency but does not really solve the problem because each repair still goes to the entire group.

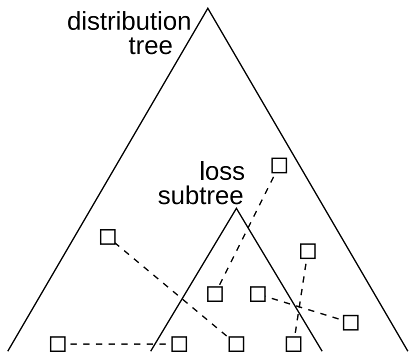


Figure 1: When a hierarchy is constructed without knowledge of the distribution tree, members of a subtree are likely to have different parents outside the subtree. During loss recovery repairs are sent from parents to children, so many separate copies of the same packet enter the subtree. which is exactly the situation multicast was designed to avoid in the first place.

cast groups each containing the members downstream of a point where losses commonly occur, so that repairs for packets lost at this point can be sent to the corresponding group. The difficulty with both techniques is that network topology is not exposed to the source and group members—the IP service model abstracts the network as a black box. In principle, this makes it impossible to construct local groups, and any hierarchy that might be constructed would be highly inefficient due to many members in a local region each getting a separate repair from a different faraway parent (see figure 1).

So there exists a gap between the desire of the endpoints to efficiently send repairs to members in a loss subtree, and the black-box unicast and multicast services offered by the network. There are basically three ways to close this gap:

- Expose topology information to the endpoints. The information can be obtained explicitly via diagnostic services like *mtrace*, as in OTERS [8] and Tracer [7], or inferred by the endpoints via observations of loss patterns [13].

- Provide transient subgroups, as in PGM [14]. Group members could inform the network of each packet loss they experience, causing "trails of breadcrumbs" to be laid down. The network could then forward the repair from the source along the trails to the members that need it, removing the breadcrumbs at the same time.

- Provide additional forwarding services defined in terms of the multicast distribution tree. For example, RMTP proposed *subcast* to send to a subtree, LMS [11] proposed *parentcast* to send to one's parent in a hierarchy constructed by the network, and we have proposed *randomcast* (section 3) to forward a packet randomly in the tree (so that it probably goes to a nearby member).

The first approach has the advantage of requiring no changes to existing routers, but is somewhat counter to the Internet architecture. The second is quite heavyweight, requiring state to be set up and torn down in routers for every packet loss. We take the third approach, which is hardly more burdensome than multicast, and does not require endpoints to know anything about network topology. Using this approach, the loss recovery solution consists of a network-layer part and an end-to-end part: new forwarding services in the network, and a feedback/repair protocol in the end hosts.

Several loss recovery schemes have been proposed that take this approach. LMS uses parentcast to send feedback in a network-constructed hierarchy, and uses subcast for repairs. OTERS is similar except that the hierarchy is constructed by the end hosts using mtrace and subcast. We have proposed two schemes that do not use hierarchies: Search Party (section 4) uses randomcast for feedback and subcast for repairs, while Rumor Mill (section 5) uses randomcast for feedback and unicast for repairs.

There are variations on each of these forwarding services (like the method of identifying a subtree, or the probability distribution used in random forwarding), and other services can be imagined (like *pachinkocast*—sending to a random member of a specified subtree). However, they all share a common structure, which has allowed us to define a general forwarding service called *treecast*, of which all the other services are parameterized instances.

The end-to-end protocols built on top of the new forwarding services share much functionality and structure, so we define a component architecture for implementing them. Reusing common components is not only convenient, but will allow fair comparisons between similar protocols. For example, LMS and OTERS are essentially the same except for the method used to build a hierarchy, which is handled by one component. This decomposition will allow us to explore the design space by varying individual components, and to generate new protocols by mixing and matching components.

Our framework for multicast loss recovery is thus a general network-layer part (treecast) and a general end-to-end part (the component architecture). We will build this framework as a simulation environment, and later as a user-level testbed (in which treecast is implemented over UDP). Using the framework, we will evaluate design tradeoffs like hierarchy versus randomization, subcast versus unicast repairs, and overhead versus delay.

Ultimately we expect to learn which forwarding capabilities yield the greatest benefits in return for the added network complexity. Also, we expect to determine, for a given set of forwarding services and application requirements, which loss recovery scheme yields the best performance.

The rest of the proposal is organized as follows. Section 2 provides an overview of previous forwarding services (subcast, parentcast, and AIM) and loss recovery protocols built on them (LMS, OTERS, Tracer, RMA). Sections 3, 4, and 5 give overviews of randomcast and our two protocols that use it, Search Party and Rumor Mill. Section 6 describes treecast and the component architecture in more detail. In section 7 we outline our plan for building a simulation framework and a testbed, and we conclude in section 8.

2 Previous Work

There have previously been several multicast loss recovery schemes consisting of network-layer support via additional forwarding services, plus an end-to-end feedback/repair protocol. All of these schemes organize the group members into a hierarchy with the source at the top. When a member detects a loss, it sends a repair request (NAK) to its parent in the hierarchy. Non-leaf members are then responsible for sending repairs to their children, or possibly to all their descendents.

Note that the hierarchy of members is not the same as the distribution tree inside the Internet. In the hierarchy, all nodes are group members, and the edges are logical parent-child relationships between them. In the distribution tree, only the leaves are group members; the internal nodes are routers and links, and the edges are the interfaces connecting routers to links[2]. However, as noted in section 1, the hierarchy will be efficient only if it is correlated with the distribution tree, otherwise many members in a local region may each obtain a separate repair from a different faraway parent, rather than having just one repair enter the region and then be replicated.

RMTP [9] assumed a hierarchy very well correlated with the distribution tree, in such a way that the set containing a member and its descendents is exactly the set of members below some point in the distribution tree. (How to construct such a hierarchy was left as future work.) Because this is exactly the kind of set affected by a packet loss, it would be common for a member to wish to send a repair to all its descendents, so RMTP proposed *subtree multicast*: an encapsulated packet is sent to a router inside the distribution tree, where it is decapsulated and thenceforth forwarded downward like a normal multicast packet. More recently the term *subcast* has come to refer to any forwarding service that sends to a subtree (i.e. all members below some point of the distribution tree); encapsulation is one of several methods for accomplishing this.[3]

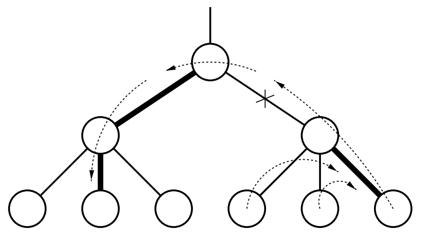


Figure 2: The paths taken by three parentcast request packets after a packet loss in LMS. The flagged edges are indicated by heavy lines.

LMS [11] proposed an elegant solution to the hierarchy-construction problem in the form of a forwarding service that was unnamed, but which we call *parentcast*. When the distribution tree is formed, each node flags one of its downward edges[4] (perhaps arbitrarily, or perhaps based on the distance to the nearest member, or on the minimum cost advertised by a member). A parentcast packet that arrives from the flagged edge is forwarded upward (toward the source), while one that arrives from any other edge is forwarded to the flagged edge. If we call the recipient of a parentcast packet the parent of the sender, then a hierarchy is defined that is perfectly correlated with the distribution tree.

The parentcast service also inserts *turning point* information into the packet. On any path connecting one leaf to another there is exactly one point where the direction changes from upward to downward. The node at this point inserts into the packet identifying information about the edge on which the packet arrived. In LMS, repair requests are sent via parentcast, and the recipient of a request can use the turning point information to subcast[5] a repair to the subtree below the identified edge, which is the largest subtree containing the requestor but not the responder. A marvelous property of the hierarchy defined by parentcast

[2]A link can be a local area network, like an Ethernet or an FDDI ring, connected to many routers.

[3]Notice that identifying which tree a packet is to be forwarded on requires both a multicast group address and a source address, because some multicast routing protocols build a separate tree for each source. So whereas a multicast packet contains two addresses, source and group, a subcast packet must contain three: a source/group pair to identify the distribution tree, plus another source address to identify the sender. This requirement applies not only to subcast, but to all the new forwarding services discussed in this proposal.

[4]Actually, LMS assumed that all links were point-to-point, and spoke of *routers* flagging downward *links*, but in recognition of the fact that a link can connect more than two routers, we have generalized the concept to nodes flagging edges.

[5]The LMS flavor of subcast is named *directed multicast.*

is that, if every member of a subtree sends a packet to its parent, then exactly one packet will escape the subtree (see figure 2). Therefore, when every member affected by a loss sends a request, all but one of the requests will stay inside the loss subtree, arriving at members who lack the data and ignore the request, while a single request will escape the loss subtree and generate a single repair sent to the entire loss subtree (or possibly a larger subtree, because the turning point could be higher than the site of the loss).

OTERS [8] uses the same sort of hierarchy, and the same end-to-end protocol, as LMS, but uses a different method to build the hierarchy. Instead of parentcast, where the network assigns parents, members use subcast and *mtrace* (a diagnostic function of multicast routers for discovering the path from the source to a member) to elect their own parents. Requests are then sent via unicast to the parent's address. Tracer [7] is similar, but uses TTL-scoped multicast[6] instead of subcast for disseminating the path information, so these messages go to a wider audience than necessary, but there is no reliance on a new forwarding service for building the hierarchy.

AIM [6] is a general forwarding service that can emulate subcast and parentcast[7]. In addition, it associates with each point in the distribution tree a *positional label*, which is a string of small integers specifying which edge must be traversed from each node to get from the root to the point in question. These labels allow the network to effectively *re-hang* the distribution tree from any point, forwarding packets as if "up" means toward that point rather than toward the root. This is useful with multicast routing protocols that build bidirectional shared trees, where "down" means away from the sender rather than away from the root. RMA, the reliable multicast protocol built on top of AIM, includes a feedback/repair protocol similar to that of LMS.

3 Randomcast

The previous loss recovery schemes all involve hierarchies, which means that for every loss there is a single member whose request will evoke a repair, and a single member who will send that repair, and the entire loss subtree is depending on the correct operation of those two members. This raises the question of robustness. While there are ways to build more robust hierarchies (as in STORM [16]), we question whether multicast loss recovery requires the use of a hierarchy at all. Therefore we have proposed *randomcast*, which can be used instead of parentcast to forward requests, but defines no hierarchy.

[6]The TTL (time-to-live) field of a packet limits how far the packet may be forwarded.

[7]The AIM flavor of parentcast is named "anycast".

Randomcast is a service that forwards packets randomly inside a multicast distribution tree. When a randomcast packet arrives at a node, it may be forwarded[8] to any neighbor in the tree except the one the packet came from. Whenever a node acts as a turning point (receives a randomcast packet from below and forwards it downward, i.e. away from the root), it can insert into the packet information sufficient to address the subtree below the arrival interface (for the purpose of sending a subcast reply, for example), just like parentcast.

The probability distribution used to select the outgoing interface is critical to the behavior of systems using the randomcast service. We have evaluated two distributions: uniform, and weighted by subtree population. We find that the weighted distribution can be better for choosing whether to forward upward, because it reduces the scope of subcast replies, but the uniform distribution is better for choosing among downward edges, because it prevents many randomcast packets from converging on the same edge and congesting it.

3.1 Up or Down?

When a randomcast packet arrives at a node X from a downward edge E, the node first decides whether to forward the packet upward or downward. If the distribution is uniform, upward is chosen with probability $1/d$, where d is the number of downward edges. (The probability is not $1/(d+1)$ because the node is forbidden from returning the packet to the edge it came from.) If the distribution is weighted by subtree population, upward is chosen with probability $L(E)/L(X)$, where $L()$ denotes the number of leaves below a point (1 if the point is itself a leaf). An essential feature of both distributions is that the sum of the probabilities over all the downward edges is 1.

Some multicast routing protocols already require routers to be aware of their neighbors in the tree, so each router could occasionally communicate with its downward neighbors, obtain a count of the number of leaves below each one, and compute the sum. The EXPRESS multicast service proposal [4] suggests that routers keep track of subtree populations. The uniform distribution does not require knowledge of subtree populations, but the weighted distribution has two attractive properties.

First, the probability that a randomcast packet originating at any leaf below node X travels above X is $1/L(X)$, regardless of the topology. This diffuses responsibility for sending requests evenly among the members, and also eases analysis: If m randomcast packets are sent from

[8]This is not a problem if the node is a router, but is slightly tricky if the node is a link (like an Ethernet), because links are generally not intelligent. The best solution is to have routers adjacent to the link perform the forwarding decisions on the link's behalf. Alternatively, links can be ignored, and routers can be considered neighbors of each other.

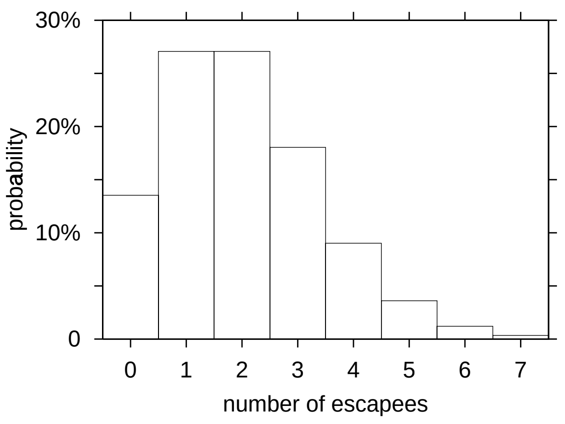


Figure 3: Probability that k randomcast packets escape a large subtree in which each leaf sends two packets (Poisson distribution with mean 2).

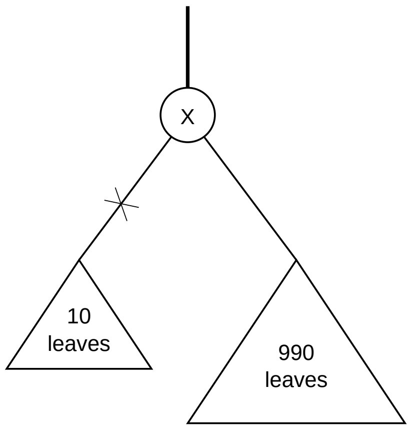


Figure 4: A packet loss in an imbalanced tree.

each of the L leaves in a subtree, the number of packets that escape the subtree has a binomial distribution with parameters $n = mL$ and $p = 1/L$, for which the expected value is $np = m$ and the variance is $np(1 - p) < m$ [15]. The variance grows with the subtree population L, but the distribution approaches Poisson with mean m and variance also m, which is fairly narrow, as shown in figure 3, meaning that the number of escapees is fairly predictable. This is analogous to parentcast, where the number of escapees is perfectly predictable (exactly m). Notice that if the uniform distribution had been used to decide whether to forward upward, the expected number of escapees would still be m (which can be shown by induction), but the probability distribution would depend on the topology, and some requests could be much more likely to escape than others.

The second, and more compelling, attractive property of the weighted distribution is locality. Suppose that repairs are subcast using the turning point information in the randomcast requests. Consider the scenario of figure 4. With a uniform distribution, there is a 50% chance that a request from the smaller subtree will be forwarded upward from X, causing the repair to be delivered to the members in both subtrees, of which 99% are not interested. With a weighted distribution, there would be only a 1% chance of this undesirable occurrence. If the loss occurs just above the larger subtree, the weighted distribution will almost always forward the request to X's parent, but the repair will still be of interest to 99% of the recipients.

3.2 Which Downward Edge?

When a packet is to be forwarded downward by a node X, either because it arrived from above or because the node has already decided against forwarding upward, there is again a choice between uniform and weighted distributions. With a uniform distribution, the probability of forwarding to downward edge E is $1/d$, or $1/(d - 1)$ if the packet arrived from below (and is therefore forbidden from being forwarded back onto the edge it came from). With a weighted distribution, the probability is $L(E)/L(X)$ if the packet arrived from above. If the packet arrived from downward edge A, the probability of forwarding to edge E is $L(E)/[L(X) - L(A)]$. It follows that before the node has decided whether to forward upward, if it is using a weighted distribution for both choices, the probability that it forwards to E is $L(E)/L(X)$.

A node using the weighted distribution for selecting children will tend to distribute randomcast packets evenly among its descendent leaves, but packets from many incoming edges can get concentrated onto one outgoing edge, and the concentration continues to increase if the same thing happens at successive nodes. With the uniform distribution, a node distributes packets evenly among its downward edges, avoiding that hazard. Because trees with more leaves tend to be deeper, the weighted distribution will tend to route packets to farther away leaves, whereas the uniform distribution will tend to choose shorter paths, leading to smaller round-trip times. For these reasons, the uniform distribution is preferable for choosing among downward edges.

4 Search Party

The previous section described randomcast as a forwarding service in isolation, but we originally designed it in the context of a particular loss recovery scheme called Search Party. The network-layer part consists of both randomcast and subcast (section 2). This section describes the end-to-end part, then briefly discusses some of the performance characteristics of the scheme as a whole. Much more detailed information, including analysis and simulation results, can be found in [1].

Members detect losses by observing gaps in the sequence numbers of arriving data packets. Data packets from the source contain timestamps, allowing members to estimate the delay variance and hence to know how long to wait after a gap is observed before concluding that it was caused by a packet loss, rather than packet reordering. After a loss is detected, the member sends requests continually via randomcast until it receives a response. Search Party requires the variant of randomcast that uses the population-weighted distribution for deciding whether to forward upward, inserts turning point information, and uses the uniform distribution for choosing among downward edges.

Responses containing the requested data are sent via subcast, using the turning point information inserted by the network into the request, just as in LMS (see section 2). Members ignore requests for data they do not have, and generally respond to requests for data they do have, but not if the request arrives closely on the heels of a response and appears to have been sent before the sender received the response, because in that case the request should have been satisfied already (the request and response crossed each other in the network). Timestamps in requests, expressed as offsets from the arrival times of data packets, aid in this determination.

After a loss occurs, every member in the loss subtree is sending randomcast requests continually at some rate. We can imagine that each member is conducting a random search for the missing data, usually searching close by, and occasionally searching farther away. Because of the way randomcast forwarding is defined, the rate of upward request traffic on any edge inside the loss subtree is the average sending rate of the members below that edge (neglecting losses of requests), and so the rate at which requests escape the loss subtree (and generate responses) is the average rate of the members inside it. Therefore, although no one knows where the loss occurred or how large the loss subtree is, the members automatically form a *search party* of just the right size so that requests reach far enough at the appropriate rate, and the recovery time is independent of the loss subtree population.

Like upward request traffic, downward traffic inside the loss subtree is also well-behaved, because of the uniform distribution used by randomcast. For example, if all members send at the same rate, then every edge carries no more than twice that rate downward. Therefore, the number of requests received by a member does not greatly exceed the number of received responses.

Unlike a hierarchy-based scheme, in which dead members high up in the hierarchy have a large impact on their descendents, Search Party is insensitive to dead members, because requests sent by any member of the loss subtree have an equal chance of escaping, and because each escaping request is likely to go to a different recipient. No matter which member is unable to send requests or responses, the impact on the rest of the group is merely a slight increase in the expected retransmission delay.

The rate at which members send requests is a trade-off between delay and overhead. If requests are sent very often, then one is likely to escape the loss subtree very soon and generate a timely response, but after the response arrives, there will still be requests in flight in the network, some of which could yet escape and generate duplicate responses (recall that the requests that do not escape are ignored because their recipients saw the first response). On the other hand, if requests are sent infrequently, then it will probably take longer for one to escape and generate a response, but there will be very few requests in flight when it arrives, so there will probably be no duplicate responses. Applications can measure the average request/response round-trip time and tune their request rate appropriately, depending on how delay-sensitive they are. Simulations have shown that Search Party can achieve average delays nearly the same as those of a hierarchy-based scheme (like LMS) if a few duplicate responses per loss are acceptable, or average delays about twice as long with about 0.7 duplicates per response, or near-zero duplication if long delays are acceptable.

Any loss recovery scheme that uses subcast for repairs risks sending the repair to a larger subtree than the loss subtree. Given that the requestor lacks the data and the responder has the data, we know the loss occurred somewhere above the requestor, and somewhere below the lowest node lying above both the requestor and responder, but we don't know exactly where. Search Party, like LMS, sends the response to the largest candidate subtree, to make sure to cover the entire loss subtree. LMS defined *exposure* as the ratio of the number of members who receive a response over the number of members in the loss subtree. In a hierarchy-based scheme, depending on where losses occur in the distribution tree, the average exposure can be large; as an extreme example, if all losses affect only the top member in the hierarchy, then all responses to go to the entire group, and the exposure is N, the number of members in the group. But in Search Party, because of the population-weighted random forwarding, the average exposure cannot exceed $1 + \ln N$, regardless of the topology or loss patterns.

5 Rumor Mill

Although we originally designed randomcast for the network-layer part of Search Party, it can be combined with a different end-to-end protocol to form another loss recovery scheme, called Rumor Mill, that is much simpler than Search Party, at the cost of somewhat poorer performance. The network-layer part of Rumor Mill is simpler because it does not use subcast—responses are unicast. Therefore, turning point information need not be inserted into the randomcast requests. Furthermore, the advantage of population-weighted forwarding (see section 3) applies only when responses are subcast, so we can use the simpler uniform weighting, which does not require the network nodes to know subtree populations, or even to know which way is "up". We call this variant of randomcast *trivial randomcast*, because every node simply forwards to any edge other than the one the packet came from, chosen uniformly at random.

Also, there may be environments in which policies prevent unprivileged hosts from sending multicast or subcast traffic, so that the network load generated by a host can be bounded by its access link bandwidth. Search Party could not be used in such an environment, but Rumor Mill could.

The basic protocol is extremely simple. Losses are detected just as in Search Party, and requests are again sent continually via randomcast until a response is received. Responses, however, are sent via unicast back to the requestor. Members ignore incoming requests for data they do not have, and always respond to requests for data they do have.

In schemes that subcast repairs, a single repair goes to the entire loss subtree. In schemes that unicast repairs, like this one, the data is relayed from member to member. In Rumor Mill, the data is relayed in a haphazard way, much like the spreading of a rumor, hence the name. The advantage of unicast repairs over subcast repairs is that the exposure is 1—members never received unsolicited repairs. The disadvantage is that the recovery time now grows with the population of the loss subtree. Whenever the data reaches another member of the loss subtree, that member becomes capable of relaying it further (by responding to incoming requests); therefore we expect the number of recovered members to grow exponentially over time, or in other words, we expect the recovery time to be proportional to $\log L$, where L is the number of members in the loss subtree. We are currently conducting simulation experiments to test this hypothesis.

As in Search Party, there is a trade-off between delay and overhead—sending requests more often leads to shorter delays but more duplicate responses. Intuitively, suppose each member sends m requests per round-trip time. During the early stages of recovery, the number of recovered members should increase by a factor of $1+m$ per round-trip time, but toward the end most members will receive m responses, $m-1$ of which are duplicates. Of course, this naïve analysis ignores the fact that every request has a different round-trip time, and ignores the effects of topology. Rumor Mill does not lend itself to analysis as well as Search Party, but simulations should provide more insight.

In Search Party, members received about as many requests as responses, but in Rumor Mill, recovery takes longer, and we expect each member to receive something like $\log L$ requests per response. Therefore, if multiple losses are outstanding, it is worthwhile to bundle multiple requests into a single request packet. We should not, however, bundle so many together that a single request packet generates more than about one response, otherwise the responder will send a burst of packets which are likely to clog the network. Also, once there are as few requests as responses, there is little to be gained by further reducing the number of requests. Members can monitor the number of requests sent versus responses received, and adapt their bundling factor accordingly.

6 Loss Recovery Framework

Our loss recovery framework consists of two parts: a network-layer part, which is a general forwarding service called *treecast*, and an end-to-end part, which is a component architecture for implementing request/response protocols.

6.1 Treecast

The various forwarding services that have been defined in terms of the multicast distribution tree—subcast, parentcast, and randomcast—all share a common structure: a packet is first forwarded to some point in the distribution tree, where information about that point is optionally inserted into the packet, then the packet is forwarded downward to some or all of the members in the subtree below that point. We call these three stages the *upward phase*, the *bounce point*, and the *downward phase* (see figure 5).

Note that an internet topology is a bipartite graph in which the nodes are alternately *routers* and *links*, and the edges are *interfaces*. It is the interfaces that have addresses. The bounce point is a node, i.e. a router or a link.

The behavior of each treecast stage can be varied independently:

Upward phase This phase is parameterized by `up_rule`, which specifies the bounce point. The possibilities include:

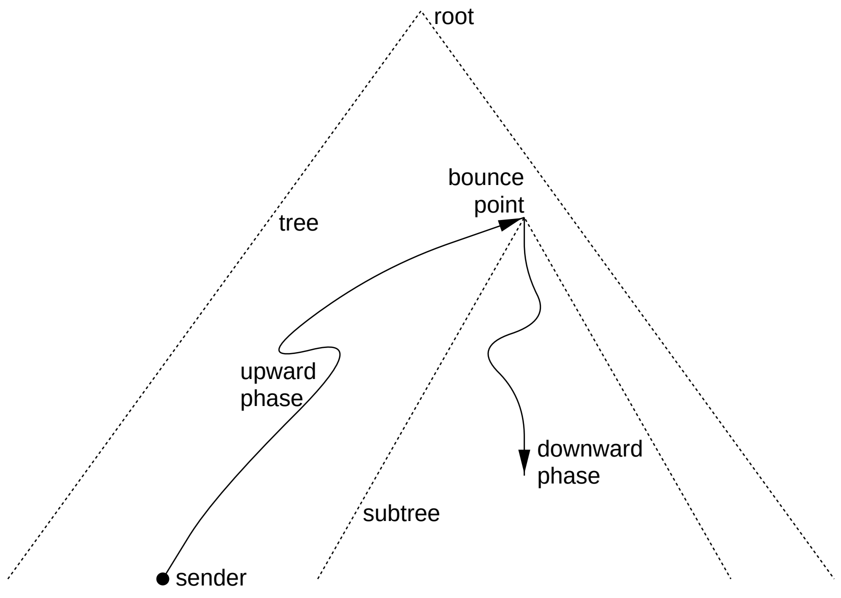


Figure 5: The treecast forwarding service.

- `root`
  The bounce point is the root of the distribution tree.

- `address` *address*
  The bounce point is the router or link immediately below the specified interface address.

- `position` *label*
  The bounce point is specified by a positional label as in AIM [6].

- `height` *distance*
  The bounce point lies on the path from the sender to the root, at the specified distance above the sender.

- `depth` *distance*
  The bounce point lies on the path from the sender to the root, at the specified distance below the root.

- `random` *distribution*
  The bounce point is chosen randomly: As the packet is forwarded toward the root, each node flips a coin to decide whether it is the bounce point. The probability of continuing upward from a node depends on *distribution*, which could be `uniform` (probability is 1 over the number of downward edges) or `population` (probability is the number of members below the arrival edge over the number of members below the node).

- `hierarchy` *metric*
  The bounce point is the first node on the path to the root for which the sender is not the least-cost member below that node. The cost function is specified by *metric*, and could for example be the distance from the node to the member, or the cost advertised by the member, or some function of the two, or a default cost assigned by the network.

Notice that for the last four upward rules the bounce point is specified relative to the sender, who must be a member of the group, whereas for the other rules the bounce point is specified absolutely, and therefore the sender need not be a member of the group. The relative rules allow the packet to be forwarded "upward" from the root, in which case the packet goes to the source, and there is no bounce point.

Bounce point This stage is parameterized by `bounce_rule`, which specifies one or more kinds of information to be inserted into the packet. The possibilities include:

- `entrance_address`
  The address of the downward interface on which the packet arrived. This is available only for relative upward rules.

- `top_address`
  The address of the bounce point's upward interface.

- `position`
  A positional label of the bounce point as in AIM, plus a suffix indicating from which downward interface the packet arrived. (The suffix is unspecified for absolute upward rules.)

- `depth`
  The distance between the bounce point and the root.

- `remaining_hops`
  The value of the packet's TTL field (time-to-live, also known as hop limit) just before it leaves the bounce point. The recipient can subtract the final value of TTL from this stored value to determine the height of the bounce point with one-hop granularity.

- `parity`
  A boolean value indicating whether the bounce point is a router or a link, useful in conjunction with `remaining_hops` to achieve half-hop granularity. (Routers are an integral number of hops away from end hosts, while links are an odd number of half-hops away.)

Downward phase This phase is parameterized by `down_rule`, which specifies which members below the bounce point should receive the packet. The possibilities include:

- `all`
  All members below the bounce point.

- `random` *distribution*
  A single member chosen at random. As the packet is forwarded downward, each node chooses one downward edge. The probability of choosing an edge depends on *distribution*, which could be `uniform` (all edges equally likely) or `population` (edges weighted by the number of members below each). If the upward rule was relative, the arrival edge is excluded from the choices unless it is the only choice

- `hierarchy` *metric*
  The least-cost member, where the cost function is specified by *metric* as for the corresponding upward rule.

Depending on which rules we use, treecast can emulate all the forwarding services discussed so far. For example, setting `up_rule` to `address`, `height`, or `depth` and `down_rule` to `all` yields flavors of subcast; setting both rules to `hierarchy` yields flavors of parentcast; and setting both to `random` yields flavors of randomcast. Additionally, new forwarding services can be generated; for example, setting `up_rule` to `address`, `height`, or `depth` and `down_rule` to `random` yields flavors of *pachinkocast*, which forwards to a random member of an explicitly specified subtree; setting `up_rule` to `hierarchy` and `down_rule` to `random` yields a service for which exactly one request will escape a loss subtree (like parentcast), but requests will not converge on parents with many children.

Rather than propose that treecast be deployed in its full generality, we view treecast as a taxonomy to organize our thoughts about what tree-based forwarding services are possible, and to structure our experiments. We expect that a small number of the many mechanisms enumerated above will prove to be useful and inexpensive enough to warrant deployment.

6.2 Request/Response Architecture

To complement network-layer support for treecast, we define a component architecture for implementing the end-to-end part of multicast loss recovery. Protocols like RMTP, LMS, OTERS, Search Party, and Rumor Mill all share a common structure: a member detects a loss and sends a request to another member (or possibly the source), who responds (if it has the data) back to the requestor or to a subtree containing the requestor.

We divide each sink (group member) into five components according to the functions that the protocols tend to share or perform differently. The first two components will be the same for all protocols:

- *network endpoint*
  Provides an abstract interface to unicast, multicast, and treecast forwarding services.

- *core*
  Provides common functions like passing data up to the application, storing received data and request/response meta-data so that responses can be sent, detecting losses, filtering out requests that appear to have been already satisfied, and demultiplexing incoming messages to the other components.

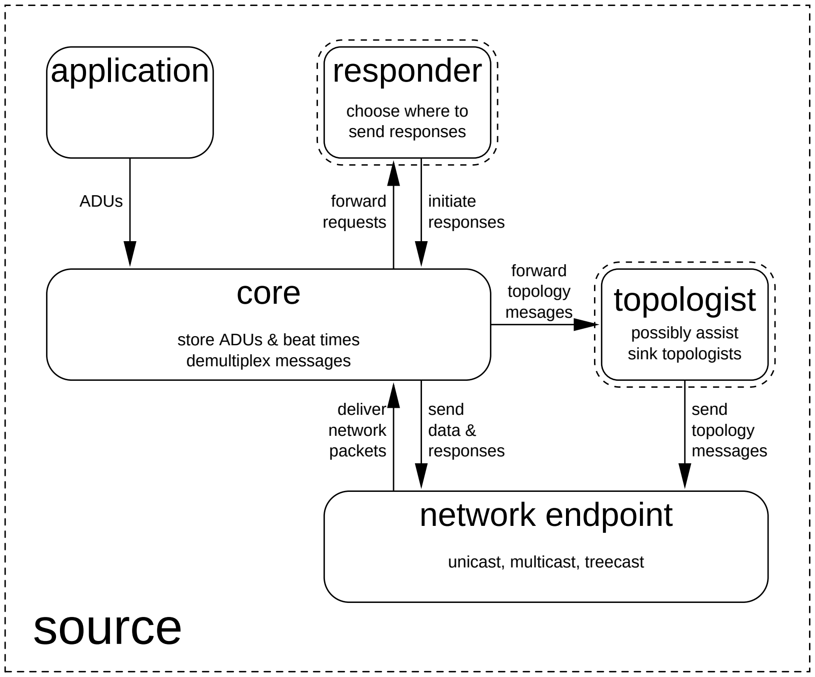
The remaining three components will have multiple implementations, which are swapped in and out to generate the various protocols:

- *responder*
  Decides where to send responses (either only to the requestor, or to a subtree, and if so, which subtree).

- *requestor*
  Decides when to send requests, and what to ask for in each request.

- *topologist*
  Decides where and how to send the requests. May also build a hierarchy, estimate round-trip times or subtree populations, etc.

The source is divided into similar components, but there is no requestor, and the core, responder, and topologist are likely to be simpler. The interfaces between the components are illustrated in figure 6.



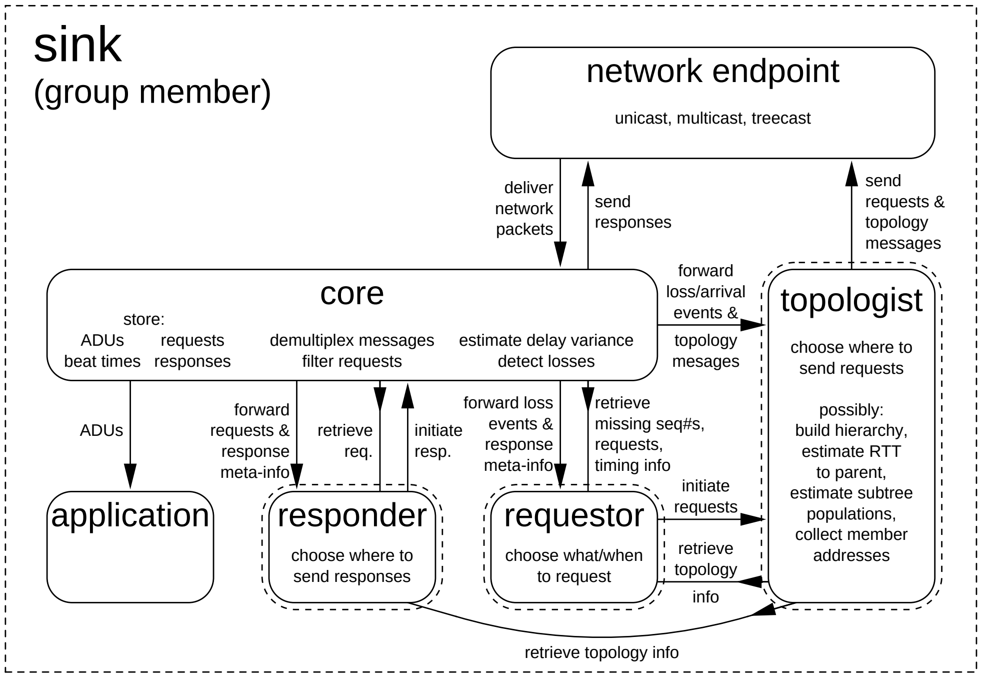

sink
(group member)
network endpoint
unicast, multicast, treecast
deliver network packets
send responses
send requests & topology messages
core
store:
ADUs
beat times
requests
responses
demultiplex messages
filter requests
estimate delay variance
detect losses
forward loss/arrival events & topology messages
topologist
choose where to send requests
possibly: build hierarchy, estimate RTT to parent, estimate subtree populations, collect member addresses
ADUs
forward requests & response meta-info
retrieve req.
initiate resp.
forward loss events & response meta-info
retrieve missing seq#s, requests, timing info
initiate requests
retrieve topology info
application
responder
choose where to send responses
requestor
choose what/when to request
retrieve topology info

Figure 6: The request/response component architecture.

All the protocols discussed so far can be cast into this architecture. For example, in an LMS sink the requestor periodically issues a request for everything that is missing, with a period based on past response times; the topologist always sends the request via parentcast; and the responder always subcasts the response to the subtree indicated by the turning point information in the request. OTERS would differ from LMS in only one of the five components—the topologist would build the hierarchy using mtrace and subcast, and send requests to the parent via unicast.

Additionally, new protocols can be generated by modifying one component at a time. For example, we can investigate the effects of using different metrics for electing parents in hierarchy construction. An orthogonal area of research is to make better use of a given hierarchy by having the responder unicast some responses and subcast others, and use the turning point information from *multiple* requests to help choose between the two options.

7 Research Plan

- 1999-May,Jun
  Our next step is to finish evaluating and writing up Rumor Mill, and to submit it to a conference.

- 1999-Jul,Aug
  Work for Sony CSL in Japan (not directly related to this proposal).

- 1999-Sep,Oct,Nov
  We will build a simulation framework in **ns** [10] implementing treecast and the request/response architecture, leveraging experience from simulations of Search Party and Rumor Mill. We will build enough instances of each component to implement the existing loss recovery protocols, plus a few new ones (like a requestor and responder that make better use of a hierarchy).

- 1999-Dec, 2000-Jan
  We will use the simulation framework to address the following questions: Given a particular subset of treecast support, what is the best way to build a hierarchy? (The issues are convergence time, maintenance overhead, and goodness of the final hierarchy.) What is the best way to use the hierarchy? How well can we do without a hierarchy? Which of the many treecast mechanisms enable the best performance for the least complexity?

- 2000-Feb,Mar,Apr
  Using the experience gained from the simulator, we will build a testbed—a user-level treecast router that talks to other instances of itself to form a treecast backbone (Tbone). We may build it from scratch, and write a few dummy applications on top of it. Or, if possible, we may modify an existing reliable multicast library, libsrm [12], to use the Tbone so that we can use existing libsrm applications with the various loss recovery schemes. In either case, we will instrument the treecast router collect instructive statistics.

- 2000-May,Jun
  We will use the testbed to conduct a more focused and realistic study of the same questions addressed in the simulation study.

- 2000-Jul,Aug,Sep,Oct
  Write the dissertation.

8 Conclusion

The existing IP multicast service model is too abstract to permit efficient loss recovery. This problem can be solved by exposing more information about topology to the end hosts, or by adding network services. We have chosen to explore loss recovery schemes that use simple new forwarding services defined in terms of the multicast distribution tree, like subcast, parentcast, and randomcast.

In this approach, the loss recovery solution consists of a network-layer part (the forwarding services) and an end-to-end part (a feedback/repair protocol). We have generalized the network-layer part by designing the treecast service, which defines a number of orthogonal mechanisms that can be composed to generate a wide variety of forwarding services. Similarly, we have generalized the end-to-end part by designing a request/response architecture, which defines a number of components that can be assembled to generate a wide variety of protocols. Together, these building blocks form a framework that we will implement first as a simulation and later as a testbed. We will use the framework to fairly compare the performance of different loss recovery schemes, and to evaluate the effects of varying individual mechanisms and policies within a scheme, and to explore the design space by designing new components and assembling them in new combinations.

Our goals are to determine which forwarding capabilities yield the greatest benefits in return for the added network complexity, and to determine, given a set of forwarding services and application requirements, which loss recovery scheme yields the best performance.

References

[1] Adam M. Costello and Steven McCanne. Search party: Using randomcast for reliable multicast with local recovery. In INFOCOM'99 [5]. http://www.cs.berkeley.edu/˜amc/research/search-party/.

[2] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems (TOCS)*, 8(2):85–110, May 1990.

[3] Markus Hofmann. Local group concept. http://www.telematik.informatik.uni-karlsruhe.de/˜hofmann/lgc/.

[4] Hugh Holbrook and David Cheriton. EXPRESS multicast: An extended service model for globally scalable IP multicast. In *Proceedings of ACM SIGCOMM'99*, September 1999. http://gregorio.stanford.edu/holbrook/express/.

[5] *Proceedings of IEEE INFOCOM'99*, March 1999. New York.

[6] Brian Neil Levine and J.J. Garcia-Luna-Aceves. Internet multicast based on group-relative addressing. http://www.cse.ucsc.edu/˜brian/pubs/, November 1998.

[7] Brian Neil Levine, Sanjoy Paul, and J.J. Garcia-Luna-Aceves. Organizing multicast receivers deterministically according to packet-loss correlation. In *Proceedings of ACM Multimedia'98*, September 1998. http://www.cse.ucsc.edu/˜brian/pubs/.

[8] Dan Li and David R. Cheriton. OTERS (on-tree efficient recovery using subcasting): A reliable multicast protocol. In *Proceedings of IEEE ICNP'98*, pages 237–245, October 1998. http://www-dsg.stanford.edu/DanLi.html.

[9] J. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *Proceedings of IEEE INFOCOM'96*, March 1996. http://www.bell-labs.com/project/rmtp/.

[10] UCB/LBNL/VINT network simulator **ns** (version 2). http://www-mash.cs.berkeley.edu/ns/.

[11] Christos Papadopoulos, Guru Parulkar, and George Varghese. An error control scheme for large-scale multicast applications. In *Proceedings of IEEE INFOCOM'98*, March 1998. http://www.ccrc.wustl.edu/˜christos/PostScriptDocs/Infocom98-final.ps.Z.

[12] Suchitra Raman and Yatin Chawathe. libsrm: A generic toolkit for reliable multicast transport. http://www-mash.cs.berkeley.edu/mash/software/srm2.0/.

[13] Sylvia Ratnasamy and Steven McCanne. Inference of multicast routing tree topologies and bottleneck bandwidths using end-to-end measurements. In INFOCOM'99 [5]. http://www.cs.berkeley.edu/˜sylviar/.

[14] Tony Speakman, Dino Farinacci, Steven Lin, and Alex Tweedly. PGM reliable transport protocol specification, August 1998. http://www.ietf.org/internet-drafts/draft-speakman-pgm-spec-02.txt.

[15] Peggy Tang Strait. *A First Course in Probability and Statistics*. Harcourt Brace Javanovich, second edition, 1989.

[16] X. Rex Xu, Andrew C. Meyers, Hui Zhang, and Raj Yavatkar. Resilient multicast support for continuous-media applications. In *Proceedings of NOSSDAV'97*, May 1997. http://www.cs.cmu.edu/˜hzhang/STORM/.

[17] R. Yavatkar, J. Griffioen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of ACM Multimedia'95*, November 1995. http://www.dcs.uky.edu/˜tmtp/.