An Error Control Scheme for Large-Scale Multicast Applications

Christos Papadopoulos
christos@dworkin.wustl.edu

Guru Parulkar
guru@arl.wustl.edu

George Varghese
varghese@askew.wustl.edu

Washington University,
Campus Box 1045,
One Brookings Drive,
St. Louis, MO 63139
Phone: (314) 935-4163
Fax: (314) 935-7302

ABSTRACT

Retransmission based error control for large scale multicast applications is difficult because of two main problems: request implosion and lack of local recovery. Existing schemes (SRM, RMTP, TMTP, LBRRM) have good solutions to request implosion, but only approximate solutions (e.g., based on scoped multicast) for the local recovery problem. Our scheme achieves finer grain fault recovery by exploiting new forwarding services that allow us to create a dynamic hierarchy of receivers. We use a new paradigm, where routers provide a more refined form of multicasting (that may be useful to other applications), that enables local recovery. The new services, however, are simple to implement and do not require routers to examine or store application packets; hence, they do not violate layering. Besides providing good local recovery, our scheme integrates well with the current IP model, has small recovery latencies (it requires no back-off delays), produces fewer duplicates than other schemes, and isolates group members from details of group topology.

We have shown how our scheme can be used with a variety of multicast routing protocols, including DVMRP and PIM. We have implemented our scheme in NetBSD Unix. The implementation is about 250 lines of new C-code and is included in the appendix. The implementation requires two new IP options and one additional byte in IGMP reports. The forwarding overhead for the new services is actually lower than forwarding normal multicast traffic.

Key words: reliable multicast, error control

1. INTRODUCTION

The exponential growth of the MBONE and other multicast-capable networks has led to the widespread deployment of multicast applications such as video-conferencing, distributed interactive simulation, and news distribution. Many of these applications require data delivery guarantees not provided by IP Multicast [1]. Thus, we require multicast transport protocols that work on top of the network multicast service to provide delivery guarantees.

Large scale applications (e.g., DIS, bulk data distribution) complicate the problem. These applications have large numbers (hundreds or even thousands) of participants who may be distributed over a wide geographical area (spanning one or more continents). In addition, the highly dynamic nature of the topology and population poses new, difficult challenges to traditional error control schemes. Such control schemes (e.g., TCP [6]), used primarily in point-to-point applications, do not scale to meet the demands of large-scale multicast. In schemes like TCP, the receiver sends an acknowledgment (*ACK*) to the sender after receiving each uniquely numbered message. If the same approach is used for multicast, each receiver must individually acknowledge each message. If the message is lost by a group of receivers, the sender is responsible for deciding which receivers have lost the message (based on missing ACKs) and retransmitting. Such a scheme would lead to *ACK implosion* with a large number of receivers, and also burden the sender with the problems of loss detection and retransmissions. In an effort to alleviate these problems, the following evolutionary steps have been proposed thus far:

- A shift has been proposed from sender-based to receiver-based error control schemes [5]. Receiver-based schemes move the responsibility of error detection from the sender to the receivers. A receiver detects loss when it receives sequence number N while the previous message received in sequence was $M < N - 1$. After the receiver detects such a *gap* (i.e., the sequence numbers $M + 1,.. N - 1$), the receiver sends a request for the missing data. This paradigm significantly reduces state and processing at the sender. However, for large-scale multicast applications, we can have *request implosion* when a large number of receivers lose a message and each sends a request.
- To further move processing away from the sender, it has been proposed that the effort of processing requests and sending retransmissions be shared among *all* members of a group, not just the sender [2]. This significantly improves scalability; however, it also necessitates mechanisms to prevent duplicates from ill-coordinated members.
- In yet another attempt to improve scalability, it has been proposed that the scope of recovery messages be restricted to the region that suffered loss, thus preventing the whole group from being pestered by recovery messages from members in loss regions [2, 4, 9]. Proposed techniques include the use of unicast with multicast after a threshold, and limiting messages to a radius using the time-to-live (TTL) field in the IP header.

The above are important steps towards achieving scalability in large-scale multicast applications. Indeed, with these steps large-scale multicast can potentially achieve better reliability (since more retransmission points are available), and lower average latency (since lost data can often be recovered from a nearby neighbor) compared to unicast. Moreover, performance will actually improve as the group gets larger. However, a closer examination reveals that the adoption of the above steps introduces several new important problems absent from unicast error control. We enumerate five such problems:

1. **Request implosion**: the problem that occurs when the loss of a packet triggers simultaneous requests from a large number of receivers, overwhelming the sender and/or other receivers.
2. **Duplicate replies**: the problem that occurs when many endpoints multicast the same reply in response to a request.

3. **Recovery latency**: the latency experienced by a member from the instant a loss is detected until a reply is received.
4. **Recovery isolation (or exposure)**: if a loss affects only a small number of receivers and the sender multicasts the reply to the entire group, then recovery is not isolated to the members experiencing the fault. The repair of a local fault should ideally stay local. Exposure can be quantified by comparing the number of messages used for repair to the size of the area affected by the loss.
5. **Adaptability to dynamic membership changes**: a measure of how the efficiency (in terms of loss of service, duplicate messages and added latency) of error recovery is affected by changes in the group topology and membership.

The currently proposed solutions [2, 3, 4, 9] do not perform well with respect to *all* the metrics listed above. Randomized backoff schemes like SRM have problems with local recovery, and reduce duplicates at the expense of latency. Static hierarchical schemes like RMTP do not adapt well to membership changes. Dynamic hierarchical schemes like TMTP provide only an approximate form of local recovery.

A key obstacle in these earlier schemes appears to be the difficulty in obtaining information about topology. Some knowledge of topology is useful for multicast error control in order to locate an endpoint which is willing to retransmit, and contain replies within the region that lost the original transmission. Topology information, however, is maintained by the routers and is not easily accessible to endpoints. Involving the routers in error recovery has been dismissed as a violation of the *end-to-end argument* [7], and not scalable. Current solutions either maintain topology information without any help from the routers, or impose their own static topology.

We have designed, simulated and are implementing in NetBSD Unix a multicast error control scheme that addresses the five problems enumerated above, and offers improvements on the performance offered by existing schemes. Our scheme follows a new paradigm, where routers offer a small set of new forwarding services to the members of a multicast group. Applications develop error control schemes that can leverage off these new services to provide reliability. The services, however, do not impose a specific error control scheme. The implementation of these services at the routers eliminates the need for endpoints to learn about group topology. Additionally, it allows easy access to topology information, which leads to an efficient implementation. These forwarding services are conceptually simple, and while they do require some changes at the routers, they do not violate the end-to-end argument, and merge well with IP routing and group management protocols. In return, the presence of these services enables the implementation of scalable multicast error control schemes that overcome the five problems listed earlier.

This paper is structured as follows. In Section 2, we summarize related work. In Section 3, we present an overview of our scheme and in Section 4, we present the mechanisms in more detail. In Section 5, we discuss some limitations of our scheme, ways to overcome them, and some optimizations. In Section 6, we present our simulation results, and in Section 7 our implementation in NetBSD Unix. Finally, Section 8, concludes this report.

2. RELATED WORK

Most of the proposed solutions for large-scale multicast error control fall into one of two major categories: *randomized backoff* schemes or *hierarchical* schemes. Hierarchical schemes can be further subdivided into schemes which employ a static or dynamic hierarchy. We briefly discuss these schemes and compare their performance with our scheme in Table 1.

2.1 Randomized Backoff Schemes

An important solution to the implosion problem is the Scalable Reliable Multicast (SRM) scheme. SRM attempts to compute a dynamic leader (requestor) for each group of nodes that detect a gap and attempts to compute a dynamic leader (replier) among the group of nodes that have a copy of the missing data. The leader election is done per message by using a combination of distance tiebreakers (e.g., the closest node to the site of the fault should request) and random tiebreakers in case of equal distances. The dynamic leader election is actually implemented using randomized backoff timers; the dependence on distance is achieved by making the timer value depend on the propagation delay from the sender to the receiver.

SRM performs well in suppressing requests but slightly worse in suppressing replies. However, SRM has the following disadvantages (Table 1):

- The backoff delay for requests is set to some multiple of the unicast delay to the sender. Thus, on average, recovery delay will be higher than unicast.
- The randomization only ensures a unique requestor or replier with a certain probability. In topologies where the distance based tiebreaker is ineffective (e.g., a star), an unfortunate tradeoff must be made. Using large random numbers can make the probability of a unique requestor or replier high but increase the recovery latency; using small random numbers can make latency small but increase the probability of duplicates.
- The "multicast to everyone" approach provides excellent fault tolerance, but also exposes recovery to *all* members of the multicast group. This situation is compounded if multiple requestors and repliers are elected.
- A new receiver joining the group must measure the propagation delay to every existing receiver in the group in case the new receiver is elected as a replier. Also, if adaptive timers are used, several request-reply rounds are needed before timers stabilize.

Simulation results on random topologies with fixed timer values show that SRM typically requires about 3 times the unicast round-trip delay to recover a lost packet and produces around 2 - 10 duplicates in the process. Using adaptive timers reduces the number of duplicates after the timers are tuned. To avoid multicasting all messages to all members, SRM proposes the use of the TTL field in the IP header to limit the scope of recovery messages. However, this approach limits the scope of messages within a radius, while losses affect a subtree. Thus, it still allows duplicates to reach other regions, as shown in Figure 1.

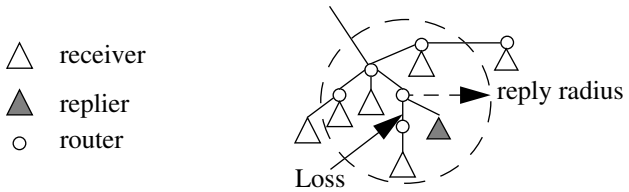


Figure 1: Limiting the reply scope using TTL does not provide good isolation

Another option to provide local recovery is to create separate local recovery groups for requests and replies. Creating a group dynamically for every loss is too costly and slow; therefore, it is better to precompute such groups for each loss region, or create them on demand as the group topology changes. However, creating a new group is a slow process and requires prune messages to propagate throughout the entire network. In groups with highly dynamic membership, such a scheme will incur significant overhead.

2.2 Hierarchical Schemes

In hierarchical schemes, members are organized in a tree hierarchy. Each member is assigned a parent and zero or more children. Request implosion is controlled by allowing requests from children to their parents only. Duplicate replies are reduced by either unicasting from parents to children or multicasting after some threshold of requests is exceeded. Parent discovery is a crucial step in hierarchical schemes. Static schemes fix the parent/children allocation at start-up. Dynamic schemes allow members to reorganize themselves as the group topology changes. Thus, dynamic schemes are more flexible but require more complex parent discovery mechanisms.

The Reliable Multicast Transport Protocol (RMTP) [4] is an example of a static hierarchical scheme. The source multicasts data to all receivers, but only a few Designated Receivers (DRs) return acknowledgments. Losses in RMTP are recovered from DRs. Retransmissions are either unicast or multicast depending on how many requests were received. This, however, is a crude solution because it performs well only at the extremes (if there are too many or very few losses). Otherwise, it incurs significant overhead, either in terms of network traffic or exposure. The Log-Based Receiver-reliable Multicast (LBRRM) [3] is another example of a static hierarchical scheme, aimed at distributed interactive simulation (DIS) applications. LBRRM uses a primary logging server and a static hierarchy of secondary logging servers which log all transmitted data. Data is multicast from the source to all logging servers and all receivers; however, only the primary logging server returns acknowledgments to the source. The receivers request lost data from the secondary logging servers; in turn, the secondary logging servers request any lost data from the primary logging server. Similar to RMTP, retransmissions in LBRRM are either unicast or multicast, or multicast based on a threshold. Both RMTP and LBRRM are based on a static hierarchy and thus require explicit set-up of DRs or logging servers before new regions can be added to the group.

The Tree-based Multicast Transport Protocol (TMTP) [9] is an example of a scheme using a dynamic hierarchy. In TMTP, every region has a Domain Manager (DM). When a DM joins a group, it searches for a parent using an expanding ring search. During the search, the new DM repeatedly broadcasts a "SEARCH_FOR_PARENT" request by increasing the time-to-live (TTL) value. When one or more DMs respond, the new DM selects the closest DM as its parent. Thus, the DMs form a dynamic hierarchical control tree. Each endpoint maintains the hop distance to its DM, and each DM maintains the hop distance to its farthest child. These values are used to set the TTL field on requests and replies to limit their scope. To further limit request implosion at the DMs, TMTP uses randomized backoff for requests, which, however, increases latency.

In summary, static hierarchical schemes like RMTP and LBRRM do not adapt to rapid membership changes or changes in topology. Dynamic hierarchical schemes like TMTP rely on an approximate method (expanding ring search using the TTL field) to discover parents and send replies. Thus, it suffers from the same exposure problems illustrated in Figure 1. The use of expanding ring search for parent selection in TMTP can lead to other forms of suboptimality, as well. For example, the parent chosen by a receiver R can be downstream, with respect to the source of receiver R. This can increase recovery latency compared to an optimal choice of parent.

Table 1 compares our scheme to SRM and hierarchical schemes with respect to the five metrics described above. All schemes do well in suppressing request implosion; TMTP and SRM do well in suppressing duplicate replies. All schemes except LBRRM require higher than unicast latency. Most schemes have poor isolation and, at best, are only fair in adapting to dynamic topology changes. In contrast, our scheme is as good as any existing scheme in request and reply suppression, but excels in latency, isolation

(bringing exposure down from 2^n to n, as shown in Table 2), and adaptability to topology changes. We now proceed to give more details.

Table 1: Comparison between Error Control Schemes

Scheme	Request suppression	Reply suppression	Recovery Latency	Recovery Isolation	Adaptability to dynamic changes
SRM	good	fair	higher than unicast	none, or approximate	fair: needs propagation delay and timer adaptation
RMTP	good	poor, unless very little or extreme loss	higher than unicast	poor, unless very little or extreme loss	none
LBRRM	good	poor, unless very little or extreme loss	lower than unicast, on average	poor, unless very little or extreme loss	none
TMTP	good	excellent: single reply	higher than unicast	approximate	fair: needs parent search
OUR SCHEME	good	excellent: single reply	lower than unicast	very good: visible within a subtree only	excellent: responds instantly to group changes

3. SOLUTION OVERVIEW

In a multicast environment, loss of a packet in the network results in failure to deliver a copy of the packet to all receivers located in the subtree rooted at the branch sprigging from the point of loss, as shown in Figure 2(a). A natural solution to recover the packet is the following (Fig. 2(b)):

1. the receiver directly below the loss sends a request to the receiver immediately above the loss
2. the receiver immediately above the loss multicasts the lost packet to the affected branch.

Note that both steps require some knowledge of topology. We claim that this solution is optimal because, (a) only one request and only one reply are generated, (b) replies are visible only within the affected branch, and (c) requests and replies traverse the shortest possible distance. We attempt to duplicate these steps in our scheme. Since topology information is required, we turn to the network for help. Routers, with their knowledge of topology, are the ideal candidates to provide services to navigate a request upstream until a willing replier is found and multicast replies to the subtree that experienced loss.

It is important to note that such services can be implemented without requiring routers to be aware of error recovery. Routers do not have to keep any state, like sequence numbers, or provide any guarantees to endpoints using the services. Routers simply provide the means for receivers to reach other receivers in a manner that happens to be useful for error recovery. Receivers are free to build their own recovery mechanisms on top of these services. We believe that such services do not violate the end-to-end argument if they are pure forwarding services and do not require packet examination. Moreover, we believe that adding such services to the network is justified if they result in more efficient error recovery than currently proposed schemes (provided they do not introduce unacceptable overhead in the network).

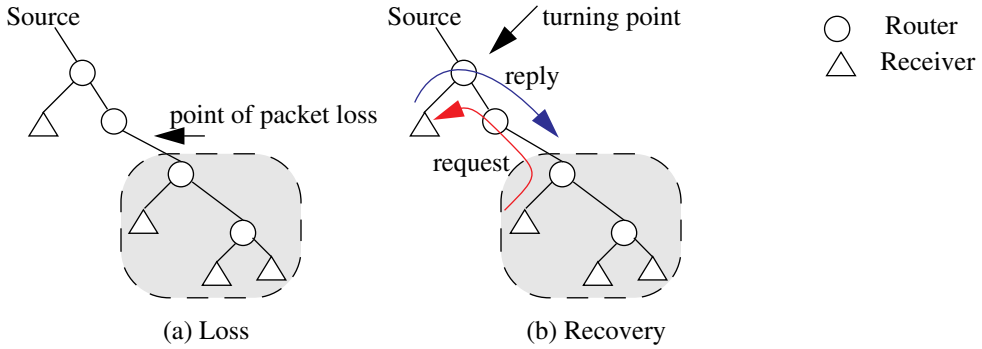


Figure 2: Recovery steps with topology knowledge

In order to provide the above services, we introduce three new concepts. First, we use routing to calculate a leader, called a *replier*, for each subtree. We use the replier to respond to requests made by other endnodes in the replier subtree, thus creating a hierarchy. Second, we use routing to define a *turning point*. A turning point is the point in the topology (Fig 2) at which a request moving upstream is moved downward towards the replier. When a replier sends a retransmission, the retransmission moves up the multicast tree until it reaches the turning point. The router at the turning point performs a *subcast*[1] to multicast the retransmission to the subtree defined by the turning point. We use these three concepts (electing repliers, defining turning points, and using subcast for retransmissions) to closely approximate the optimal recovery steps depicted in Fig. 2.

It is important to quickly see why these three ideas help our scheme do well with respect to all five measures described in the introduction and in Table 1. The use of a replier hierarchy prevents request implosion. Duplicate replies are eliminated by allowing only one replier to respond. Our scheme has near-optimal recovery latency, which is typically much *smaller* than the delay to unicast a retransmission from the source, because backoff delays are not needed and replies come from nearby repliers. (Note that in SRM[2], although replies can also come from any nearby endnode, the backoff delays are set proportional to the distance from the source; thus, recovery latencies are comparable or *larger* than the corresponding unicast delay). The use of turning points and subcasts help isolate recovery to the subtree affected by the fault. Finally, membership and topology changes are easily adapted to by routers as group membership changes by calculating new multicast trees and repliers for each subtree, if needed, without endnode involvement.

We now present an overview of these three major steps before providing more details in Sections 4 and 5. We first describe how repliers are selected and how requests are routed to them; then we describe how turning points are precisely defined and computed; finally, we describe how replies are locally multicast by the replier using a subcast.

3.1 Adding a Replier to Each Router

To help in locating a suitable replier, every router on the multicast tree selects one of its links as the *replier link* (similar to routing, a router only needs to maintain the next hop leading to a replier, not the actual replier address). Each router selects a replier from its downstream links; the only exceptions are routers with only one downstream link, which select the upstream link as the replier link, and the router adjacent to the source, which selects the source link as its replier link. A router can easily distinguish

1. This term was coined by Adam Costello, inspired by “subtree multicast” described in [4].

between upstream and downstream links by examining the information maintained by the multicast routing protocol[1].

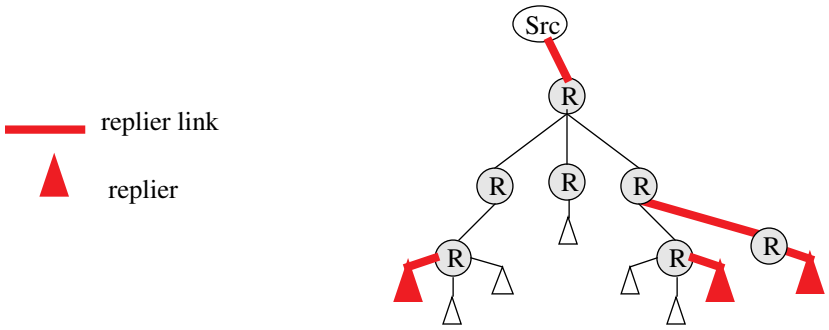


Figure 3: Replier assignment to routers

An example of repliers-to-routers assignment is depicted in Figure 4. To aid in replier selection, receivers who are willing to act as repliers notify the routers when they join or refresh their membership in the multicast group; that is, along with the join or refresh message, a receiver indicates its willingness to act as a replier. Whenever its replier state changes (e.g., as a result of a membership change), the router propagates the change upstream so that internal routers can update their replier links. Thus, the maintenance of replier state incurs minimal overhead and is automatically updated as the group membership changes.

3.2 Step 1: Sending Requests

After all repliers are in place, sending a request proceeds as follows: a receiver after detecting a loss (e.g., a gap), sends a request to the router. Routers forward all incoming requests to the replier link, except for the request arriving on the replier link, which is forwarded upstream. Thus, a router forwards at most one request upstream (reducing request implosion), and requests travel upstream only until they reach the next replier (ensuring isolation). These operations are shown in Figure 4(a). In the figure, loss occurs on

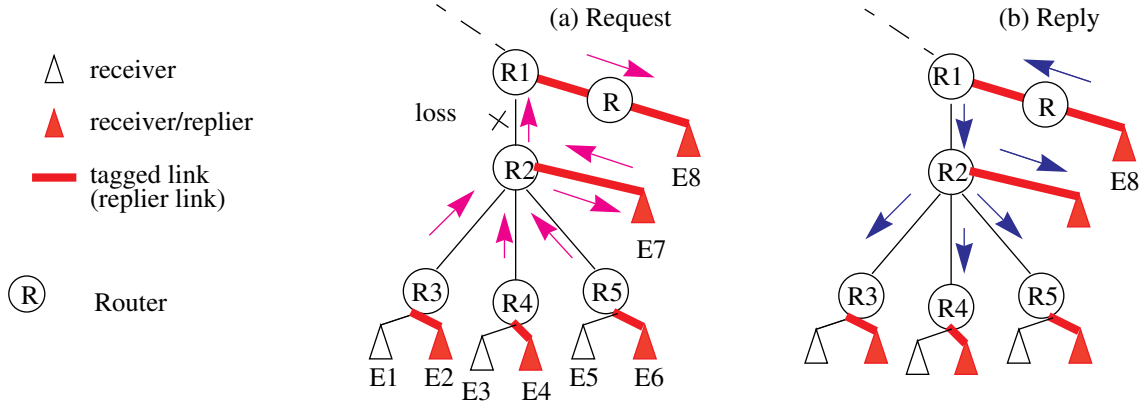


Figure 4: Request - Reply operations for multicast error recovery

1. Most popular multicast routing protocols, including DVMRP and MOSPF, maintain a (upstream link, downstream links) mapping, which is used in routing multicast packets.

the link between R1 and R2 (marked with an "*X*"). Endpoints E1 through E7 detect the loss and send requests. Then, the following events take place:

- E7 sends a request, which R2 forwards to R1 because E7 lies on R2's replier link.
- E1 sends a request which is forwarded by R3 to E2. Similarly, requests from E3 and E5 are forwarded to E4 and E6 by R4 and R5.
- The request from E2 is forwarded to R2, because E2 is on R3's replier link. Similarly, the requests from E4 and E6 are also forwarded to R2.
- R2 forwards requests from E2, E4 and E6, to E7.
- The request from E7 reaches R1, which forwards it towards E8. E8 has the requested data.

At this point a request has reached an endpoint, which has the data and is willing to retransmit.

3.3 Step 2: The Turning Point

This is an important concept in our scheme. We define the *turning point* in the request's path towards the replier as the router which forwards the request to a replier. For this to happen, the request must arrive at the router on a downstream link other than the replier link. Thus, in the previous figure, the turning point for the request sent by E7 is R1, and the turning point for the requests sent by E2, E4 and E6, is R2. When a request passes through its turning point, the router inserts into the request the router's address and the identifier for the link on which the request arrived. Other routers on the path to the replier do not change this information. Thus, requests traveling downstream carry their turning point with them. The reason why will become clear shortly.

3.4 Step 3: Sending Replies using a Subcast

If a replier receives a request but does not have the requested data, the replier ignores the request since it must have sent a similar request of its own. If a replier receives a request and has the requested data, the replier retransmits the data using a subcast. To do so, the replier creates a reply containing the data and the link identifier carried in the request. The replier then unicasts the reply to the router at the turning point. When the router receives the unicast, it extracts the data and multicasts it on the specified link. This process is depicted in Figure 4(b). Assume that E8 has just received a request from E7. Then:

- E8 creates a multicast message containing the reply. E8 encapsulates the message in a unicast message and sends it to R1 (the request's turning point).
- R1 decapsulates the multicast message and multicasts it on the link leading to R2.
- From that point on, all downstream routers and endpoints treat the reply as a regular multicast message coming from the source.

It should be clear from Figure 4 that the turning point is the root of the subtree which has lost the requested data. Thus, establishing the turning point before multicasting a reply is a crucial step in containing replies to the loss region and allows our scheme to achieve very good isolation.

4. PROTOCOL DESCRIPTION

The previous section has given an overview of the recovery steps in our scheme. In this section, we describe the mechanisms of our scheme in detail. We list the state, control messages, and actions taken by the routers and the endpoints to establish and maintain the replier state, and send requests and replies. Finally we discuss some limitations of the scheme.

4.1 Establishing Replier State

In this subsection we describe how repliers are established and maintained by the routers. We begin by outlining the state required at each router to maintain a replier; then we list the control messages used to build this state and the router actions for each control message.

4.2 Router state

Following the IP multicast model, the router replier state is soft state to ensure robustness. The state is required per sender, per multicast group and consists of the following:

- the upstream link
- a list of downstream links
- the replier link
- the cost to reach the current replier (e.g., hop count or current replier loss)
- A timer to age replier entries

Note that the state that needs to be added to the routers is significantly less than the above. The upstream and downstream links are already maintained by the routing protocol. The timer (used to age and eventually expire replier entries that have not been refreshed) is similar (or may be the same) to the timer used by the group membership protocol. Thus, the replier link and cost are the only new items. The purpose of cost is to capture differences between multiple potential repliers, so that the best one can be selected.The cost may be the loss rate experienced by the current replier so that the most reliable replier is selected, or the router-replier distance so that the closest replier is selected.

Since replier state is maintained on a per sender basis, it is possible that in the worst case a router may have to select a replier for every sender in a multicast group. However, we expect that in reality many receivers will advertise that they are willing to act as repliers for several (or maybe all) senders. A router can then select a single replier for all senders who share the same upstream link, further reducing the required replier state.

4.2.1 Control messages

Two types of control messages are required to create and maintain the replier state. Both may be combined with existing group membership protocol messages eliminating the need to create new messages.

- "**Want_to_be_replier**" message: This message is sent periodically to the routers by receivers who are willing to act as repliers. With these messages, the receivers advertise a cost to help the router select the replier with the least cost.
- "**Replier_gone**" message: this message is sent by receivers who are leaving a multicast group, but only if they were advertising "Want_to_be_replier" messages while participating in the group.

4.2.2 Router actions

The above messages trigger the following actions at the router:

"Want_to_be_Replier" message:

- The router notes the link the message came from.
- The router examines the cost advertised by the message.

- If the new cost is higher than the current cost, the message is ignored.
- If the message came from the replier link and the new cost is equal to the stored cost, the router refreshes the replier expiration timer.
- If the new cost is less than the current cost, the old cost is discarded and the new value is stored. If the link the message was received on is different than the current replier link, the router updates its replier link. The replier expiration timer is then reset.

"Replier_gone" message:

- If the message did not come from the router's replier link, the message is discarded; otherwise, the router clears its replier field (replier is now upstream).

Whenever a router's replier state changes, the router propagates the change to the upstream routers, which repeat the same operations. Thus, replier state propagates to all routers on the multicast tree. As an optimization, it is beneficial (but not required) that routers cache the next best replier link so they can instantly switch to the cached link if the current replier leaves the group or fails.

Note that a router is not required to store the replier's address because a replier does not have to be notified when it is selected by a router. Similarly, upstream routers do not have to notify downstream routers if they select them as part of the replier path. This saves complexity and state, and allows routers the flexibility of switching repliers at will.

The mechanism for selecting repliers can be easily integrated with the group membership protocol (e.g., IGMP) so that the replier state is created while the group is being formed. For example, when a member joins or refreshes its membership in a group, it may also refresh its replier status with the same message. Thus, very little work is needed for creating and maintaining the replier state. In return, the replier state is updated instantly as the group grows or shrinks.

4.3 Sending Retransmission Requests

When a loss occurs, receivers must send a request which the routers will deliver to an upstream replier. To do so, we create a new type of control message to carry retransmission requests, which is examined by every router in its path. The new control message is called "**Forward_to_replier**" message. Routers identify this control message via a hop-by-hop option in the multicast header.

We describe the function of these messages with the following example. We assume that some form of gap-based loss detection is used. When a receiver detects loss, the receiver immediately creates a retransmission request containing the sequence numbers of the lost packets, which is inserted into the body of a "Forward_to_replier" message. The control information of the message contains two items. The first item is the <source, multicast address> for the group. This information is required to identify the appropriate source tree when routing the message to a replier. The second control item is a <router address, link identifier> entry, used to mark the turning point. This entry is initially empty. A header option is then added to force routers to examine the packet, and the control message is multicast to the group in the normal fashion.

4.3.1 Handling of "Forward_to_replier" messages at the routers

Upon reception of a "Forward_to_replier" message, a router examines the control information to determine what actions are required. If the router has no knowledge of the multicast group, the message is silently discarded. Otherwise, the router performs the following actions, depicted in Figure 5:

- If the message came from a downstream link and either (a) the router has no replier link, or (b) the message came from the replier link, then the router forwards the message on the upstream link leaving the control information unchanged.
- If the message came from a downstream link other than the replier link, then this is the turning point. The router fills in the <router addr, link id> fields and forwards the message on the replier link.
- If the message came from the upstream link and (a) the <router addr, link id> fields are not empty, and (b) a replier link exists, the router forwards the message to the replier link unchanged. If the <router addr, link id> fields are empty, the message is silently discarded. If no replier link exists, the router signals an error (some upstream router erroneously thinks that this path leads to a replier).

Looking at Figure 5, requests from a receiver first follow path (a) until they reach the turning point. At this point the router performs action (b), filling in the <router addr, link id> fields and forwarding the message towards the replier. Finally, downstream routers on the path to the replier perform action (c) until the message reaches the replier.

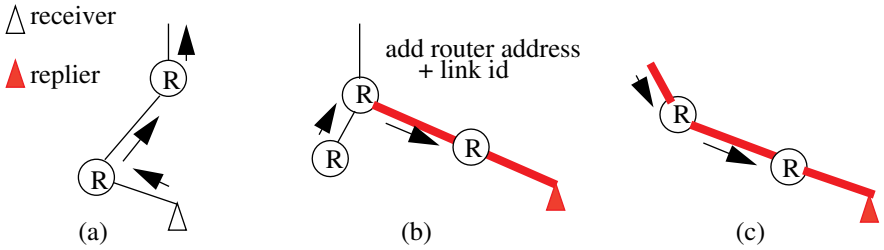


Figure 5: Router actions for “Forward_to_Replier” messages

Requests must currently be examined by all routers along their path due to their non-standard forwarding. However, we believe that integrating the request processing in the fast path of a router is feasible because the overhead for forwarding these messages is very low. The only additional state required on top of the normal routing state is the replier link. This state may be added to the routing state so that it is readily available once the routing lookup is performed. The remaining operations require on the order of 10 instructions or less. Only the router at the turning point has to actually touch the header, which may be done in another 10 instructions or so.

4.4 Sending Replies

The above description shows how a request reaches a replier. If the replier wants to respond, it uses a service called *subcast* to deliver the data to the appropriate subtree. We describe this process next.

The subcast service is a crucial element in maintaining isolation. It allows a host to ask a router to perform a *subtree multicast* on behalf of the host. A subcast consists of two parts: a unicast from the replier to the router, and a subsequent multicast by the router on one of the router’s links. Thus, a subcast reaches only receivers in the subtree routed at a downstream link of a router. To perform a subcast, the replier creates a multicast packet, encapsulates it in a unicast packet and sends it to the router. In the unicast, the replier specifies which of the router’s links the packet should be multicast. The router decapsulates the packet, performs some validity checks (described below) and multicasts it on the requested link. A subcast can be summarized as follows:

1. A replier receives a “Forward_to_replier” message, which contains a retransmission request and

the turning point information.

2. The replier scans its buffers for the requested data. If the data is not found, the request is ignored.
3. If the data is found, the replier creates a multicast packet containing the reply. The multicast packet is then unicast to the router at the turning point.
4. The router decapsulates the multicast packet and checks the validity of the group and link specified in the message. If the checks succeed, the router multicasts the packet on the specified downstream link.

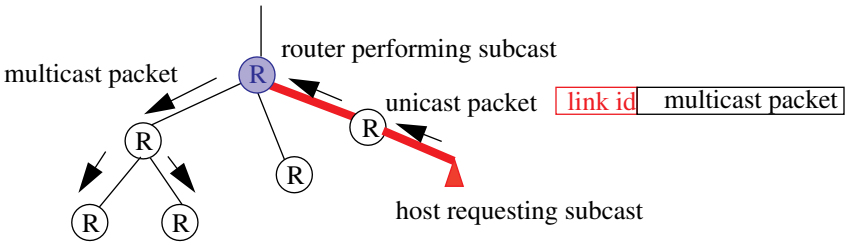


Figure 6: Host performing a subcast

Note that in a subcast only the router at the turning point performs operations beyond normal forwarding. The overhead incurred by these operations, however, is comparable to the overhead required to forward a regular multicast packet.

In the rare case when a replier receives a retransmission request after its buffers have been purged, the replier should not discard the request but forward it to the next replier with a new "Forward_to_replier" message. This time, however, the turning point information should be copied from the original request instead of being left empty. If the router at the turning point finds these fields non-empty, the router forwards the request to the replier without changing the turning point information. Thus, the new replier sends a subcast to the original turning point, preserving isolation.

A subcast may be used to preserve isolation even when only the original sender is allowed to retransmit (e.g., for security reasons, or to reduce the buffering and processing requirements at the receivers). To do so, repliers unicast the request to the original sender, including the turning point information. The sender subsequently performs a subcast to the router at the turning point.

4.5 Source Spoofing

Some routing protocols (e.g., DVMRP) create a separate multicast tree for each sender. With such protocols, the multicast reply resulting from a subcast must contain the original sender's address as the source address, otherwise it will not reach the appropriate receivers. To avoid this problem, we allow repliers to use the original source's address in the multicast packet (i.e., perform *"source spoofing"*). However, to allow receivers to distinguish spoofed from real packets, routers ensure that spoofed packets are marked and include the replier's address in the message. Thus, source spoofing poses no additional security concerns since the real sender can always be identified by the recipient. Source spoofing is unnecessary with routing protocols that create shared trees.

4.6 Limitations

Under certain conditions, our scheme may generate some duplicate requests, and may deliver retransmissions to endpoints that do not need them. In this subsection, we describe when this happens and how these problems can be mitigated.

4.6.1 Duplicate data packets

Since there is always one replier in our scheme, a receiver will only receive one reply; thus, in our scheme, there are no duplicate replies. It is possible, however, that a receiver may receive a reply it does not need, as a result of recovery initiated by other receivers. We will refer to these packets as duplicates.

An example where duplicates are created is depicted in Figure 7. In this example, a packet is lost on

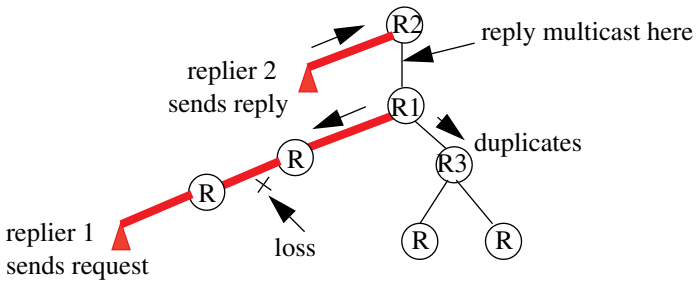


Figure 7: Loss on replier path causes duplicate messages

the path between R1 and replier 1. Replier 1 sends a request which reaches replier 2. In response to the request, replier 2 sends a subcast to R2, which multicasts the reply on the downstream link leading to R1. The reply reaches all of R1's downstream links, causing duplicates on the subtree routed at R3.

Even though this problem does not inhibit recovery, it may lead to the "crying baby problem," where excessive loss experienced in one branch causes duplicates at a large number of other receivers. We deal with this problem by using the cost field to select a replier that advertises the least loss. For example, R1 will select a replier from the right-hand-side branch if this branch experiences less loss, even though the replier on the left-hand-side branch may be closer.

4.6.2 Duplicate Requests

Recall that a router forwards at most one request on its upstream link. Thus, the maximum number of requests a replier can receive is typically bounded by the number of downstream links of the router at the turning point. However, it is conceivable that in some pathological cases where many routers have selected the same replier, the replier may receive a potentially large number of requests (Figure 7). Here a large

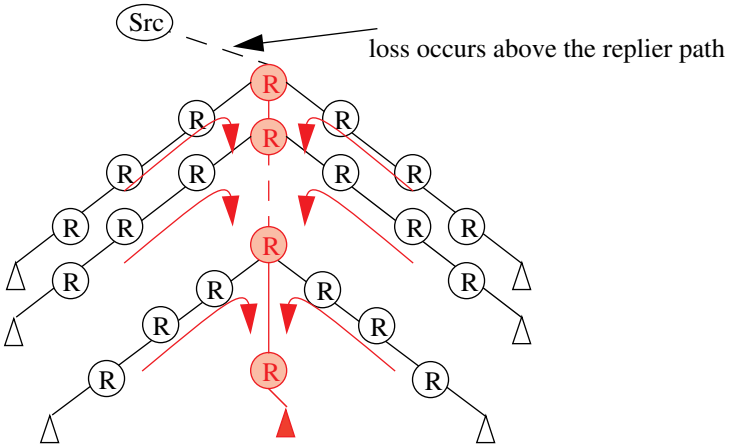


Figure 8: Duplicate requests

number of neighboring routers (shaded) have selected the same replier (also shaded), forming a long *replier path*. Every request reaching a router on the replier path is now forwarded to the same replier, making the number of requests at the replier proportional to the sum of the downstream links of all the routers on the replier path.

We believe that such pathological scenarios are rare. However, the problem can be solved by modifying the "Want_to_be_replier" messages to carry the sum of the children of all routers on the replier path. At each hop, a router adds the number of its children (minus the replier link) to the sum. If the sum exceeds some threshold, the next upstream router is forced to select a different replier link, thus shortening the replier path. The threshold is specified by the replier, allowing the replier to control the maximum number of requests it can receive.

4.6.3 Dealing with lack of per-source state

While most of the multicast capable routers on the MBONE today use DVMRP, some new routing protocols like PIM-SM and CBT create shared multicast trees around a *core* or *rendezvous point*, and thus do not maintain per-source information. This enables them to scale to groups with many sources. In order to handle these protocols, we propose the following changes to our scheme, as depicted in Figure 9:

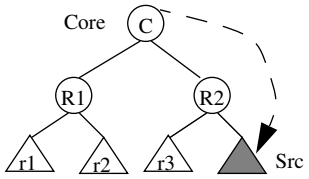


Figure 9: Dealing with shared trees

We calculate subtree leaders (repliers) as before for the core based tree. A request is directed by the routers to the leaders as before. Requests from the leaders are directed towards the core. In order to guarantee that the source will eventually receive the request, whenever the core receives a request, it unicasts it to the source. The source in turn performs a subcast to the turning point as before. So rather than have the source directly connected to the root as in DVMRP schemes, the source is connected by a unicast path to the root in CBT.

The above modification works well for PIM, which appears to be the most likely future multicast routing protocol. Sources in PIM usually send data first to the core via register messages, which are then multicast by the core. If the core has requested the source to join the group, then intermediate routers maintain per source information, which leads to a similar situation as with DVMRP. However, in schemes like CBT, data fans out to the tree from the source, and no per-source state is maintained at the routers. This means that routers can no longer distinguish upstream and downstream links with respect to a source, and thus may not direct requests appropriately. The loss of per-source information may result in some loss of isolation. We are evaluating the impact of this problem.

5. FURTHER DETAILS

In this section, we discuss additional issues that have been glossed over in the previous sections. We specify how our scheme generalizes to LANs, especially those containing multiple receivers. We discuss lost retransmissions and requests, and other failure modes. Finally, we show how we can prevent request implosion on the replier link for routers with a large number of links.

5.1 Selecting Repliers in a LAN

For simplicity, the previous sections have assumed that only one receiver resides at each router link. This, of course, is not always true. In cases where routers are connected to a LAN, receivers on the LAN run a simple election algorithm to select a replier. The election takes place without any involvement from routers. Receivers use local multicast (i.e., a multicast with the TTL value set to 1) for the election. The first receiver on the LAN becomes the replier; new receivers check for a replier by sending a local multicast. If a replier exists, it responds with another local multicast. When the replier leaves the group, it sends a local multicast announcing its departure, which triggers the remaining receivers to elect a new replier.

Once a replier is elected, it periodically multicasts a "Want_to_be_replier" message. Remaining receivers monitor the replier to ensure that it is alive. The router does not need to know which receiver is currently acting as the replier. When a request arrives, the router delivers the request to all receivers via a local multicast, but only the replier responds.

When receivers on a LAN detect loss, they use a back-off scheme to delay sending requests to the replier. The replier multicasts its request immediately, which cancels other receivers' requests. If loss was internal to the LAN, the replier repairs the loss with a local multicast. However, if loss was specific to the replier, the replier's request will cause a duplicate to arrive on the LAN.

5.2 Dealing with Loss of Requests and Replies

Recovery may fail if a retransmission request is lost before it reaches a replier, or if a reply is lost before it reaches a receiver. To detect lost requests and replies, receivers set a timeout after sending a request, and resend the request if no reply is received when the timeout expires. Receivers detect a retransmission failure as follows:

- After sending a request, each receiver sets a timeout proportional to its distance to the sender.
- If the reply arrives before the timer expires, the timer is cancelled.
- If the timer expires before a reply is received, another request is sent and a new timeout is set; the process repeats up to a specified maximum number of attempts.

Note that if some receivers received the first reply, additional recovery attempts will involve only receivers which have not yet received the reply. This make it very likely that a different replier will be involved in the next round.

5.3 Dealing with a Replier Failure

The failure of a replier may disrupt recovery for some period of time. Note that a failed replier will not always cause problems; replier failure becomes problematic only if the failed replier is located directly above or below the point the data was lost (see Figure 2, earlier). If the failed replier is located anywhere else, recovery will proceed unaffected. Soft state allows routers to eventually detect failed repliers. However, detection via soft state may take too long; to enable fast detection of replier failure, a receiver may do the following:

- the receiver sends a request explicitly asking the replier to immediately acknowledge its reception. The receiver sets a timer to wait for the acknowledgment.
- If the timer expires, the receiver sends another request to the router at the turning point, requesting that a new replier be selected.

While the router is uncertain that a replier exists (for example, after receiving requests from receivers to switch repliers), the router may switch to a cached replier, or "no replier" (i.e., forward requests upstream), until its replier state is refreshed.

5.4 Routers with a Large Number of Links

If a router has a large number of links, the router's replier may receive a large number of requests from downstream repliers. To avoid this problem, the router may partition its links into smaller groups and select a replier for every group, as shown in the example in Figure 10. In this example, requests from links

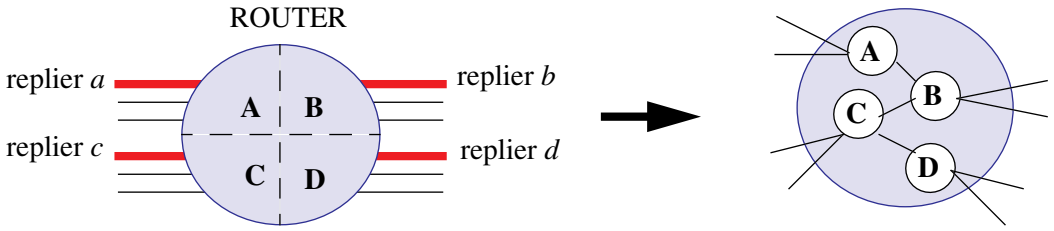


Figure 10: Partitioning of the links at a router with a large number of links

in group *D* go to replier *d*, but requests from replier *d* go to replier *c*, requests from replier *c* go to replier *b* and so on. Requests from replier *a* are forwarded upstream. By partitioning links this way, the maximum number of requests a replier can receive is significantly reduced.

6. RESULTS

In this section we evaluate our scheme with respect to the five problems identified in the Introduction, namely request implosion, duplicate replies, latency, exposure and adaptability. Request implosion is controlled by using the replier hierarchy. The number of duplicate requests a replier may receive is determined by the sum of the downstream links of each router on a replier path. We have shown a method to ensure that this sum never exceeds the replier's threshold of implosion (see section 4.6.2). There are no duplicate replies in our scheme because there is always only one replier. We have not yet evaluated the adaptability of our scheme to topology changes; however, we expect it to be very good since the replier state changes as group membership changes.

Two problems remain to be investigated in order to justify our claims in Table 1. These are exposure and latency. We present numeric and simulation results to investigate the performance of our scheme in terms of exposure and latency.

6.1 Numeric Results: Exposure

We define exposure as:

$$\textit{Exposure} = \frac{\textit{\# receivers that received a reply}}{\textit{\# receivers that should have received the reply}}$$

We calculated the exposure in our scheme and compared it to a scheme which does not have local recovery (like SRM). The exposure depends heavily on the topology of a group because the topology and the location of loss determine which receivers lost a packet. We chose a binary tree as our target topology. Exposure was then calculated as follows:

- drop a packet on a link at height *h*

- calculate the resulting exposure due to that fault
- repeat the above until a packet was dropped on all links of height *h*
- report the average exposure

Table 2: Exposure in SRM and in our scheme with a binary tree topology

Height	SRM (without scope control)	Our Scheme
1	2	1
2	4	1.5
3	8	2
4	16	2.5
...	...	...
h	2^h	O(h)

The results are presented in Table 2. From the table, we see that without local recovery the exposure increases exponentially as losses move closer to the leaves. In contrast, with the local recovery offered with our scheme, exposure increases only linearly. Note that recent studies of losses on the MBONE indicate that most losses tend to occur at the leaves [8].

6.2 Simulation Results

We created a simulation of our scheme to measure *Nuisance* and the recovery latency in our scheme. The nuisance factor is defined as follows:

$$\textit{Nuisance} = \frac{\textit{\# unwanted replies received by R}}{\textit{total number of losses}}$$

Thus, schemes without local recovery have nuisance = 1. Recall that since in our scheme there is only one replier, a duplicate in our scheme is an unwanted message received as a result of recovery in some other part of the tree. Thus, in contrast with exposure, which measures how many receivers were exposed to recovery in some other region, nuisance measures how often a receiver is pestered by messages from recovery elsewhere.

In our simulation we simulated the following:

- A single multicast tree with one sender.
- All the router functionality, including replier setup, forwarding messages to repliers, the turning point and subcasts. The cost used by routers to select repliers is currently the hop distance.
- Gap-based error detection at the receivers (assuming that packets are delivered in FIFO order by the network). All receivers participate in error recovery.
- Loss of original data packets only. Requests, retransmissions, and other control messages are not lost.

The inputs to the simulation are the topology, link parameters (bandwidth, latency, loss) and the type of loss to be simulated (random or deterministic). The outputs were the average recovery latency and the

average nuisance. Runs were made on various topologies. Here, we present results from two types of topologies: binary trees and a WAN-like topology.

6.2.1 Simulations with Binary Trees

Binary trees, although not a very realistic topology, are useful for providing insight to the behavior of the scheme in a controlled environment. Binary trees actually represent a difficult case for our scheme because the lack of internal repliers increases the replier paths, which in turn increases duplicates and latency.

For this set of results, we assigned all links identical bandwidth, latency, and loss probability. A binary tree of height 3 is one example topology used in the simulations and shown in Figure 7. The replier links are shown in bold. Other trees used in our simulations have similar structure.

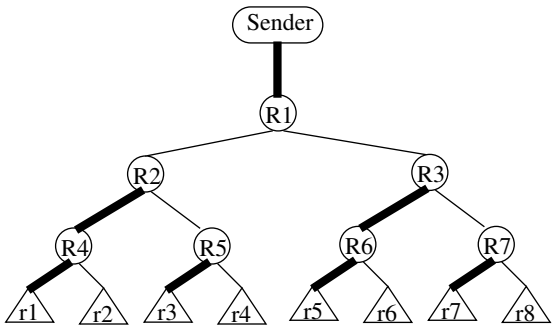


Figure 11: Binary tree of height 3

We simulated our scheme with binary tree topologies of various heights. The results were obtained as follows: a packet was dropped on a link; the unwanted replies (if any) and recovery latency were measured at all receivers; a packet was then dropped on a different link and the measurements were repeated until one packet was dropped on every link. The above is equivalent to the case where each link has equal loss probability. The results were then averaged over all receivers to calculate the average nuisance and average latency. We report the normalized latency obtained by dividing the real latency with the RTT between the receiver and the sender.

Table 3 shows the results of our simulations with different binary trees. The results show that average latency stays very close to RTT. Nuisance is low, despite the fact that there are no “good” repliers in these topologies.

Table 3: Nuisance and latency for binary trees

Tree Height	Nuisance	Avg normalized latency (avg latency / RTT)
3 (8 receivers)	0.11	0.92
4 (16 receivers)	0.10	0.93
5 (32 receivers)	0.08	0.95
6 (64 receivers)	0.06	0.96

It should be clear from previous discussion that adding receivers to the internal routers can only improve performance in our scheme. The reason is that recovery messages need to travel a shorter distance and subcasts can be more accurately aimed at the loss region. Thus, we examine what happens next if we add more receivers at the leaves. This better approximates real topologies, where receivers are typically concentrated at the edges. The simulation is run as before, with Table 4 showing the results. The original

Table 4: Nuisance and latency for different receivers per leaf router

Tree height	Receivers per leaf router	Nuisance	Avg normalized latency (avg latency / RTT)
3	2	0.11	0.92
3	3	0.09	0.889
3	4	0.08	0.875
4	2	0.1	0.93
4	3	0.08	0.908
4	4	0.07	0.897

result (2 receivers per leaf router) is taken directly from Table 3. The table illustrates that adding more receivers at the edges decreases both nuisance and latency. This was expected, because adding receivers where a replier already exists does not increase exposure; on the contrary, it allows new receivers to recover quickly and without exposing their recovery to others.

6.2.2 WAN simulation

For our final set of results, we simulate the WAN topology depicted in Figure 12. The topology is imaginary, but attempts to capture some elements of a typical WAN. There are 14 routers and 33 receivers. The selected replier links are shown in bold. If a router's replier link leads directly to a receiver, the receiver is shown in bold. The propagation delays between routers are shown on the links. The propagation delay between any router and a receiver is assumed to be 1 ms, typical of the propagation delay in a LAN. The simulation is performed as before, by dropping one packet on each link, and the results are shown on Table 5. The second row shows how the performance improves if the source is moved to R3, bringing it closer to the "middle" of the multicast tree.

Table 5: WAN

WAN: 14 routers, 33 receivers	Nuisance	Avg normalized latency (avg latency / RTT)
Source at R0	0.1	0.6
Source at R3	0.04	0.49

It is important to note that the above results were obtained by selecting repliers based on distance, since we assumed that all links have the same loss probability. In reality, routers will pick repliers that experience the least loss, so exposure and nuisance will improve significantly.

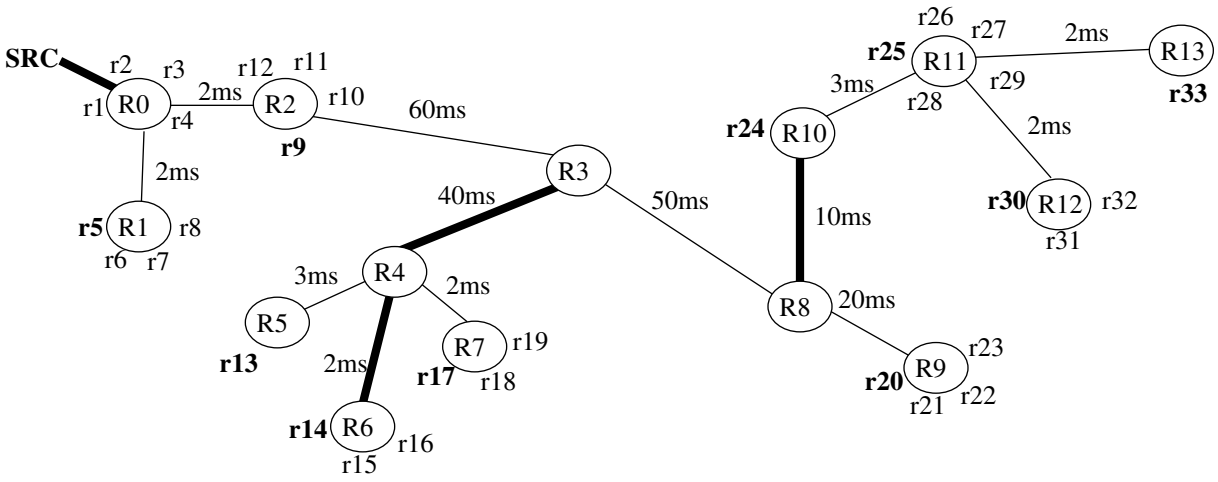


Figure 12: Imaginary WAN topology

7. IMPLEMENTATION

We have modified the NetBSD Unix kernel to support handling of requests and subcasts. The kernel modifications include the addition of two new IP multicast options, namely IPOPT_MREQ and IPOPT_SUBCAST. Requests carry the IPOPT_MREQ option in the IP header; replies are encapsulated in unicast packets which carry the IPOPT_SUBCAST option. Control information from the user are conveyed to the protocol as *ancillary data* (described below). The kernel components that were modified are: UDP output processing, IP and UDP input processing, and kernel multicast forwarding. We describe the modifications in detail next.

7.1 Ancillary data in NetBSD socket interface

Recall that when a user process sends a request or a reply, some control information must be passed to the kernel along with the data. For a request, this control information includes the `<source, group>` pair, which is required to route the request to the appropriate replier. For a subcast, the information includes the address of the router at the root of the subcast, and the link the subcast should be performed. The NetBSD socket interface provides two ways of passing control information to the kernel. The first is via the system call `setsockopt`, which allows parameters to be set on a socket that will affect all future packets sent on that socket. Examples include the multicast group to be joined at the socket and the value of TTL for all outgoing multicast packets. The second method of passing control information to the kernel, is via the `sendmsg` and `recvmsg` system calls. These calls accept as arguments control parameters (called ancillary data) that is passed along with the normal data when a packet is sent, as depicted in Fig 13. These calls can be used to pass control information that affect an individual packet, rather than a stream

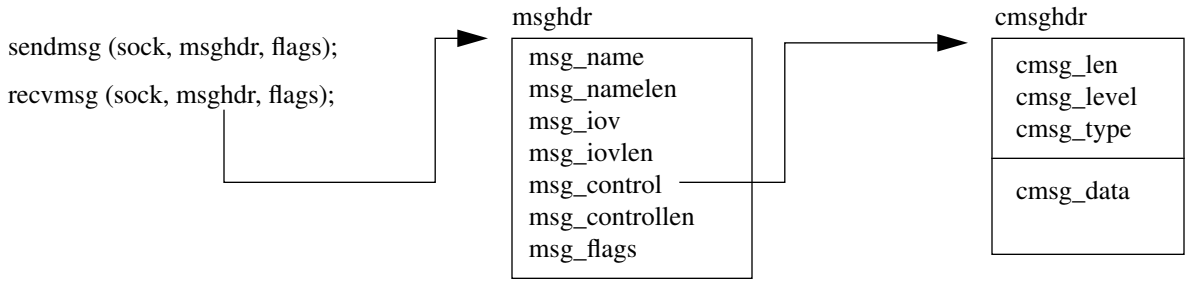


Figure 13: Ancillary data in NetBSD

of packets as with setsockopt, and thus can be used to send and receive control information associated with a request or a subcast. This is the same socket mechanism being considered for setting and retrieving information carried in IPv6 options [11].

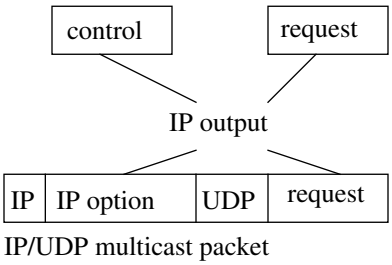
We now proceed to describe how sending and receiving requests and subcasts are implemented using these calls.

7.2 Sending/receiving requests

Recall that in addition to information about lost packets, requests also carry the address of the original source and information about the turning point. The latter consists of the address of the request's *outgoing* interface, and the index of the request's *incoming* link at the turning point. This information is carried by new IP option, IPOPT_MREQ.

Sending a request is done as follows: upon detection of a gap, the application creates a retransmission request with a list of the missing packets. Then, the application allocates a cmsghdr structure, sets cmsg_type = IPOPT_MREQ, and writes the address of the original multicast source in cmsg_data. The application allocates a msghdr structure, attaches the request to the msg_iov field and the cmsghdr structure to the msg_control field and calls sendmsg to send the request.

In the kernel when the call reaches UDP, the control information is converted into an IP option and formatted by a call to `ip_pcbopts`. UDP then passes the packet along with the option to `ip_output`, as shown in Fig 14. The packet is multicast through the normal path.



Send overhead	Receive overhead
20 bytes at sendmsg	26 bytes at recvmsg
call to ip_pcbopts	10 bytes of control data
12 byte IP option	10 byte IP option

Figure 14: Sending/receiving a request at an endpoint

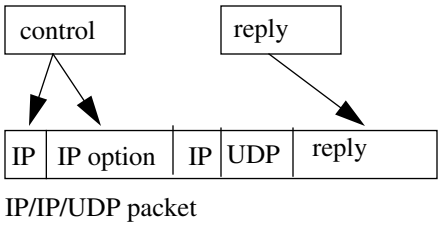
A request is received by all members on the requestor's LAN. This is desirable in order to suppress similar requests. At a router, however, a request is either forwarded upstream or to the replier link. When a request finally reaches a replier, the turning point information is extracted from the IP option by UDP and passed to `sbappendaddr` which appends it to the receive socket buffer as control information. The application retrieves the request and control information with a call to `recvmsg`.

The overhead of sending and receiving a request is summarized in Fig 14. The table shows overhead *in addition* to the normal sendmsg/recvmsg overhead. At the sending side, the overhead consists of 20 control bytes, out of which 4 bytes are for the address of the original source and 16 bytes are for the cmsghdr. This is followed by a call to `ip_pcbopts` to prepare the IP option which is 12 bytes (option type and length, src addr, router addr and router link). At the receiving side, 10 bytes of the IP option are copied into a cmsghdr structure and delivered to the application via the `recvmsg` system call.

7.3 Sending/receiving subcasts

Sending a subcast consists of unicasting an encapsulated multicast packet to the router located at the root of the loss subtree. There the packet is decapsulated and multicast on the specified link. The unicast part of a subcast contains an IP option which carries the link index. Similar to sending a request, the appli-

cation creates a reply and a control message with the turning point information, and calls `sendmsg` to multicast the packet. UDP, however, intercepts the multicast packet and passes it to a new function, `ipudp_encap`. There, the packet is encapsulated in a unicast packet, whose destination is taken from the control information. The link id is copied from the control information to a new IP option, IPOPT_SUBCAST. The process is depicted in Fig 15.



Send overhead	Receive overhead
22 bytes at sendmsg call to ip_pcbopts encapsulation IP hdr 4 byte IP option	None

Figure 15: Sending a subcast from an endpoint

The overhead for sending a subcast is similar to sending a request, but with a 4-byte IP option (type, length and link index), and the IP encapsulation header. However, there is no additional overhead for receiving subcasts at the endpoints, as they are regular multicast packets.

7.4 Handling of requests at a router

We have described so far how an application sends and receives requests and subcasts. Note that to the user these look like normal multicast packets, except for the control information. We now proceed to discuss how requests and subcasts are handled at the routers.

In order to handle requests correctly, a router with more than two interfaces (which are members of the multicast group) must maintain a replier entry for each source. This requires the addition of a new field to `struct mfc`, the *multicast forwarding cache entry*. The new field, `vifi_t mfc_replier`, contains the index of the child interface which was selected as the replier interface. This field will be set by `mrouted` in response to IGMP messages (see below).

When a multicast packet carrying the IPMOPT_MREQ option arrives at the router, it is passed to a new function, `ip_mfwdrequest`. This function performs a route lookup based on the source address carried in the option and the multicast group address; this returns the cache entry for packets from the original source. Then, the `mfc_replier` field of the entry is compared to the incoming interface. If they match, the packet is sent out to `mfc_parent`; otherwise it is sent out to `mfc_replier`. In the latter case, before forwarding the packet, `ip_mfwdrequest` fills the turning point information in the IP option.

7.5 Handling subcasts at a router

Unlike requests, subcasts arrive at the router encapsulated in unicast packets. When such a packet arrives, it is passed to a new function, `ip_subcast`. This function retrieves the interface index from the option, strips the unicast IP header plus the option from the packet, and, if the interface index is valid, forwards the multicast packet out the interface. Although not included in our implementation, a route lookup should be done to ensure that a route actually exists.

The source code for both new forwarding functions described above (`ip_mfwdrequest` and `ip_subcast`) is shown in the appendix. Their overhead is lower than the regular IP multicast forwarding function, `ip_mforward`.

7.6 Setting replier information: modifications to IGMP

This part of the implementation has not been completed at the time of writing. Our experiments were run using a fixed replier entry. We plan to modify IGMP to carry information to aid `mrouted` set the replier. The modifications will be completed shortly and results can be obtained from http://dworkin.wustl.edu/~christos/.

8. CONCLUSIONS

In this paper we have described an error control scheme for large multicast groups, based on a new set of forwarding services provided by routers. These new services are simple to implement (can be included in the router's fast path) and easy to integrate with the current Internet multicast model, while incurring minimal overhead at the routers. In return, these services allow receivers to implement efficient and fast error recovery, while maintaining their isolation from the topology of the group. These services are very general and can be cleanly implemented without exposing routers to the details of the particular error recovery mechanism used by the receivers.

Our scheme shows significant performance gains compared to other schemes, with respect to the five metrics outlined in Table 1, as shown in Section 6. The biggest gains are in terms of lower latency and isolation. The scheme typically recovers from errors in one round-trip delay or less. The number of receivers exposed to recovery is reduced from a factor of 2^h without local recovery, to h, where h is the height of the multicast tree. Even though we did not simulate the scheme's adaptability to a changing group topology, we believe that it is very good. The scheme adapts to a changing group topology very quickly, without requiring heuristics or learning. As we have shown in the appendix, our scheme requires about 250 lines of C-code in the kernel, and forwarding requests and subcasts requires less overhead than normal traffic.

We believe that these new services can be useful for other purposes in addition to error control. For example, members of a group can implement a positive acknowledgment reliable transport service by selecting a fixed set of repliers to collect acks from receivers and avoid the ack implosion problem. The subcast service can also be used by such repliers to deliver state. Other applications could also use subcast to provide a much tighter form of scoped multicast than what is available today with TTL based scoping. In addition, our services can be used as the basis for a congestion control scheme that utilizes loss information from repliers.

Routers enhanced with our services may be deployed on the MBONE using a mechanism similar to the one used in RSVP [10], utilizing path messages. Note that non-enhanced portions of the multicast tree will also benefit from recovery performed by the enhanced tree. We are investigating the feasibility of such integration.

REFERENCES

[1] Deering, S., "Host Extensions for IP Multicasting," RFC 1112, January 1989.

[2] Floyd, S., Jacobson, V., McCanne, S., Zhang, L., Liu, C., "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," Proc. of ACM Sigcomm '95, pp. 342-356, September 1995.

[3] Holbrook, H., Singhal, S., Cheriton, D., "Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation," Proceedings of ACM Sigcomm '95, Vol. 25, No. 4, pp. 328-341, October 1995.

[4] Paul, S., Sabnani, K.,. Buskens, R., Muhammad, S., Lin, J., Bhattacharyya, S., "RMTP: A Reliable

Multicast Transport Protocol for High-Speed Networks," Proceedings of the Tenth Annual IEEE Workshop on Computer Communications, September 1995.

[5] Pingali, S., Towsley, D., Kurose J., "A Comparison of Sender-initiated and Receiver-initiated Reliable Multicast Protocols," SIGMETRICS '94.

[6] Postel, J., "Transmission Control Protocol - Darpa internet Protocol Program Specification," RFC 793, September, 1981.

[7] Saltzer, J.H., Reed, D.P., Clark, D.D., "End-to-End Arguments in System Design," ACM Transactions on Computer Systems, Vol. 2, No. 4, November 1984. pp 277, 288.

[8] Yajnik, M., Kurose, J., Towsley, D., "Packet Loss Correlation in the MBONE Multicast Network: Experimental Measurements and Markov Chain Models," Infocom '96.

[9] Yavatkar, R., Griffioen, J., Sudan, M., "A Reliable Dissemination Protocol for Interactive Collaborative Applications," Multimedia '95.

[10] Zhang, L., Braden, B., Estrin, D., Herzoz, S., Jamin, S., "Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification," RFC in preparation.

[11] Stevens, W., Thomas, M., "draft-stevens-advanced-api-03.txt", work in progress.

APPENDIX

```
/*
 * Reliable multicast functions
 * Christos Papadopoulos, 07/26/97.
 */

/* ENDPOINT FUNCTION
 * Send a subcast to a router (IP encapsulated,
 * UDP multicast packet).
 *
 * Multicast addr is in "nam", unicast addr in "control"
 */

int
#if __STDC__
ipudp_encap (struct mbuf *m, ...)
#else
ipudp_encap (m, va_alist)
   struct mbuf *m;
   va_dcl
#endif
{
   struct inpcb *inp;
   struct mbuf *control, *nam, *opts = 0;
   struct ip *ip;
   struct udphdr *udp;
   struct sockaddr_in *sin;
   struct ipmopt_sbc *ipsbc;
   int      hdrlen, len = m->m_pkthdr.len;
   int      error = 0;
   va_list ap;

   va_start(ap, m);
   nam = va_arg(ap, struct mbuf *);
   inp = va_arg(ap, struct inpcb *);
   control = va_arg(ap, struct mbuf *);
   va_end(ap);

   /*
    * Create space for
    * encapsulation, IP and UDP headers.
    */
   hdrlen = 2*sizeof (struct ip) + sizeof(struct udphdr);
   M_PREPEND(m, hdrlen, M_DONTWAIT);
   if (m == 0) {
      error = ENOBUFS;
      goto release;
   }
   /*
    * Compute the packet length of the IP header, and
    * punt if the length looks bogus.
    */
   if ((len + hdrlen) > IP_MAXPACKET) {
      error = EMSGSIZE;
      goto release;
   }
   /*
    * Fill the encapsulation IP header
    * with info from ipsb.
    */
   ipsbc = mtod(control, struct ipmopt_sbc *);
   ip = mtod(m, struct ip *);

   bzero(ip, sizeof (struct ip));
   ip->ip_p = IPPROTO_IPIP;
   ip->ip_len = hdrlen + len;
   ip->ip_ttl = inp->inp_ip.ip_ttl; /* XXX */
   ip->ip_src = zeroin_addr;
   ip->ip_dst = ipsbc->sbc_root;

   /*
    * Encapsulated packet.
    * Fill in IP and UDP headers from
    * information in ipsbc.
    */
   sin = mtod(nam, struct sockaddr_in *);
   ip++;
   ip->ip_v = IPVERSION;
   ip->ip_hl = sizeof (struct ip) >> 2;
   ip->ip_tos = inp->inp_ip.ip_tos;
   ip->ip_len = htons(sizeof (struct udpiphdr) + len);
   ip->ip_id = htons(ip_id++);
   ip->ip_off = 0;
   if (IN_MULTICAST(sin->sin_addr.s_addr))
      if (inp->inp_moptions)
         ip->ip_ttl=inp->inp_moptions->imo_multicast_ttl;
      elseip->ip_ttl = 1;
   elseip->ip_ttl = inp->inp_ip.ip_ttl; /* XXX */
   ip->ip_p = IPPROTO_UDP;
   ip->ip_src = ipsbc->sbc_src;
   ip->ip_dst = sin->sin_addr;

   udp = (struct udphdr *)(ip + 1);
   udp->uh_sport = inp->inp_lport;
   udp->uh_dport = sin->sin_port;
   udp->uh_ulen = htons((u_int16_t)len +
      sizeof (struct udphdr));
   udp->uh_sum = 0; /* XXX */
   /*
    * prepare options
    */
   if ((error = ip_pcbopts (&opts, control)) != 0)
      {
      error = EINVAL;
      goto release;
      }
   /*
```

```
     * pass to IP
     */
    error = (ip_output (m, opts, (struct route *)0,
        inp->inp_socket->so_options & (SO_DONTROUTE
            | SO_BROADCAST),
        inp->inp_moptions));
    m_freem (opts);
    return (error);

release:
    m_freem(m);
    if (opts)
        m_freem (opts);
    return (error);
}


/* ROUTER FUNCTION
 * Forward a multicast packet out the replier link
 * after possibly filling in the turning point info
 */
int
ip_mfwdrequest (m, opt)
    struct mbuf *m;
    u_char *opt;
{
    struct ip *ip = mtod(m, struct ip *);
    struct ipmopt_sbc *sbc;
    struct mfc*rt = 0;
    struct ifnet *ifp = m->m_pkthdr.rcvif;
    vifi_t   vifi = 0xffff;
    struct vif *vifp;
    u_char *ipmoptions;
    int      s;

    /*
     * Don't forward a packet with time-to-live of zero
     * or one, or a packet destined to a local-only group.
     */
    if (ip->ip_ttl<=1 ||
        IN_LOCAL_GROUP(ip->ip_dst.s_addr))
        return (0);

    sbc = (struct ipmopt_sbc *)opt;

    /*
     * Get route entry from the forwarding cache table
     */
    s = splsoftnet ();
    MFCFIND(sbc->sbc_src, sbc->sbc_group, rt);
    splx (s);

    if (rt == 0)
        return (1);

    /*
     * If it arrived from the replier interface,
     * forward to parent.
     * Otherwise, forward to replier after filling
     * the incoming link index and local addr
     * of outgoing interface.
     * if these fields are already filled, leave alone.
     */
    if (viftable[rt->mfc_replier].v_ifp == ifp)
        {
        vifi = rt->mfc_parent;
        }
    else{
        vifi = rt->mfc_replier;
        if (in_nullhost(sbc->sbc_root))
            {
            /* fill turning point info */
            vifi_t vf;

            /* search viftable for index of incoming interface */
            for (vf = 0; vf < numvifs; vf++)
                {
                if (viftable[vf].v_ifp == ifp)
                    {
                    sbc->sbc_vifi = htons(vf);
                    break;
                    }
                }
            if (vf == numvifs)
                return (1);
            elsesbc->sbc_root = viftable[vifi].v_lcl_addr;
            }
        }
    /*
     * Send packet on selected interface
     */
    vifp = &viftable[vifi];
    if (vifp->v_flags & VIFF_TUNNEL)
        encap_send(ip, vifp, m);
    elsephyint_send(ip, vifp, m);

    return (1);
}

/* ROUTER FUNCTION
 * Decapsulate and subcast a packet
 */
int
ip_subcast (m, opt)
    struct mbuf *m;
    u_char *opt;
{
    struct ip *ip = mtod(m, struct ip *);
    struct ipmopt_sbc *sbc;
```

```
    int      hlen = ip->ip_hl << 2;
    vifi_t   vifi = 0xffff;
    struct vif *vifp;

    sbc = (struct ipmopt_sbc *)opt;

    /*
     * get the interface index from options
     */
    if ((vifi = ntohs(sbc->sbc_vifi)) >= numvifs)
        return (1);
    /*
     * Not much else to do: strip IP header and subcast
     */
    m->m_data += hlen;
    m->m_len -= hlen;
    m->m_pkthdr.len -= hlen;
    m->m_pkthdr.rcvif = 0;

    ip = mtod (m, struct ip *);
    NTOHS(ip->ip_len);
    NTOHS(ip->ip_off);
    vifp = &viftable[vifi];

    if (vifp->v_flags & VIFF_TUNNEL)
        encap_send(ip, vifp, m);
    elsephyint_send(ip, vifp, m);

    return (1);
}
```