Scalable Reliable Multicast with Hierarchy and Polling

Marinho Barcellos

C6-Centro de Ciências Exatas e Tecnológicas
UNISINOS-Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950 - São Leopoldo, CEP 93022-000 - Brazil
http://www.inf.unisinos.tche.br/~marinho
e-mail: marinho@exatas.unisinos.tche.br

Abstract

The IP multicast architecture enabled large-scale applications of multicasting on the Internet. Many of these applications require reliable dissemination of a stream of data (e.g., software distribution or stock updates) to a large number of receivers. They face, however, a scaling limitation, known as the *feedback implosion problem*: the sender is overwhelmed by feedback packets, leading to packet losses. Recent reliable multicast protocols have been designed with scalability in mind. They usually follow a *receiver-initiated* approach: the sender is unaware of receivers; it transmits to a group id without waiting for feedback and handles retransmission requests when possible. Receiver-initiated schemes trade in efficiency and reliability for improved scalability.

This paper describes an alternative approach to scalable reliable multicasting: instead of resorting to receiver-initiated schemes, the scalability of sender-initiated protocols is enhanced through polling feedback and hierarchy. A novel polling-based implosion avoidance mechanism reduces the amount of feedback packets to desired levels, and thus avoids implosion, while the hierarchical organization is harnessed for increased scalability, local recovery, as well as improved flow and congestion control. The resulting protocol is called PRMP: Polling-based Reliable Multicast Protocol.

1 Introduction

Network-supported multicast allows the efficient transmission of packets to a large group of receivers. Packets are distributed from sender to receivers through a multicast routing tree which is set up by the network. There is a substantial gain over multiple unicasts, as follows. Consider a complete d-ary tree of height h; to deliver a packet to all d^h receivers using multiple unicasts, the *network cost*, that is, the number of edges that need to be traversed, is $d^h \times h$ (under the simplifying assumption that all edges have equal weight). In contrast, using multicast, as each edge is traversed only once, the network cost is equal to the number of edges: $\sum_{h}^{i=1} d^i$. This gain of multicasting has been realized in the Internet by the IP multicast architecture ([Deering91]), whose popularization created the potential for new multicast applications. Examples include software distribution, dissemination of "hot" web-pages, off-line video distribution, live audio/video stream broadcast, remote learning and multimedia remote conferencing. These applications differ in organization, traffic, reliability and requirements; they can be separated in two groups [Bagnall97]: *soft real-time* multicast and *fully-reliable* multicast. The former transmits time-sensitive data (usually multimedia) and can sacrifice some degree of reliability (i.e., receivers can accept some losses) in favor of timely delivery, whereas for the latter reliability is more important, and data must be *exactly reproduced* to all receivers.

Traditional reliable unicast protocols, like TCP ([Stevens94]), do not scale well for reliable multicast due mainly to "implosion losses" caused by excessive rate of feedback packets arriving from receivers. [Pingali94] has coined these protocols as *sender-initiated,* and devised a new*, receiver-initiated* approach: scalability is achieved by making the sender *independent from receivers*; the sender does not know the membership of the destination group. This fits well the receiver-oriented model of IP multicast. However, this benefit comes with a hidden cost: the lack of knowledge at the sender about receivers has negative implications with respect to throughput, network cost, and degree of reliability offered to applications.

This paper focuses on scalability of multicast at transport level, discussing the main issues faced during the design of a scalable fully-reliable, one-to-many multicast protocol. It describes an alternative approach that, instead of adopting the receiver-initiated scheme, *greatly enhances* the scalability of the sender-initiated scheme by means of polling and hierarchy. The resulting protocol is named PRMP: Polling-based Reliable Multicast Protocol.

The paper is organized as follows: Section 2 provides an overview of the protocol. Section 3 addresses the scalability limitations and shows how polling and hierarchy are used to overcome them. Section 4 describes the core of the protocol, the multicast sliding window mechanism, and how it is used to implement error, flow and congestion controls. Section 5 briefly addresses related work, informally comparing PRMP with other protocols. The paper concludes with final remarks in Section 6.

2 Overview of the Protocol

The task of PRMP is to accept a data stream generated by a *sending application* and to reliably disseminate it to a designated list of *GS receiving applications.* The protocol takes the necessary actions needed to ensure that an *exact* reproduction of the generated data stream is made available to all receiving applications. To accomplish that, the *source* takes data from the sending application, multicast it via network layer, and periodically requests confirmation of receipt of packets from all receivers (polling-based feedback error control). If no confirmation is received after several requests, the connection with that application is deemed *broken.* Data is transmitted through an arbitrary number of fixed-size packets (apart from the last). Receivers store data and make it sequentially available for *consumption* by the local receiving application.

The source does not directly communicate with all receivers; instead, receivers are logically organized according to a tree (i.e., hierarchically) with the source at the root (see Figure 1). It is assumed that a connection setup phase precedes the data transmission, during which the tree is formed[1]. Data is produced by the sending application (SA), and transmitted in packets by the source, at the root of the tree, to the first-level nodes. Each of these nodes may have one or two roles: to deliver received data to a local receiving application (RA), and/or to forward (via multicast) data packets to its own children. Feedpack packets (containing status from receivers) are sent by child nodes (via unicast) to their parent. Therefore, the source is a sender, the leaf nodes are receivers, and the internal nodes are both senders and receivers[2]. Internal nodes need not have a receiving application, in which case they only forward packets.

Failures during communication between a parent node and its children result in packets being lost or corrupted and discarded by the network. The parent detects loss of data packets through feedback packets containing negative acknowledgments (NACKs). Data losses are recovered through retransmissions; the parent keeps a copy of each transmitted data packet in its buffer until an acknowledgment (ACK) for that packet is obtained from all children (in which case

[1] it is out of the scope of this paper to address the tree formation process (see [Hofmann96]).

[2] the words "sender" and "parent" are used interchangeably in the text, as well as "child" and "receiver".

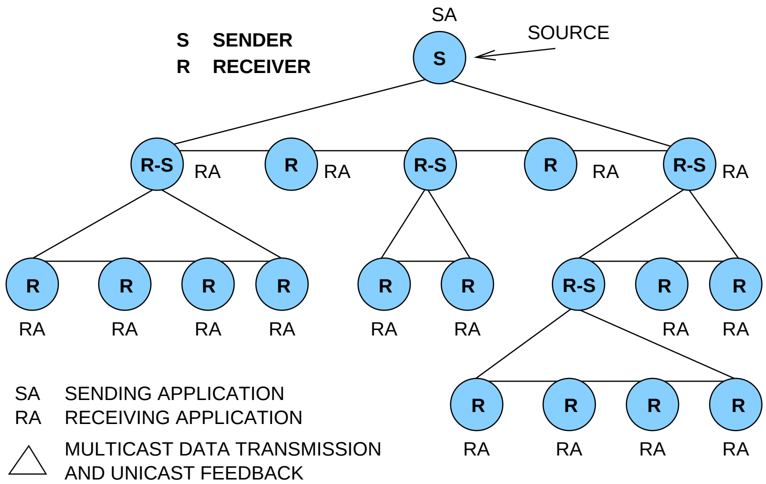

Figure 1: Example illustrating the general tree structure.

the packet becomes *fully acknowledged*). Unless the maximum degree of any node is restricted to small values, the volume of feedback packets generated may be sufficient to cause implosion losses. For this reason, PRMP employs a polling-based implosion avoidance scheme ([Hughes94]): a child only sends feedback when told to do so (through a *poll request*). PRMP introduces a novel polling-based implosion avoidance scheme whereby a parent *plans* the polling of its children so that the flow of feedback (*poll responses*) through time is "adequate": enough to allow parent to benefit from fresh status, but not enough to cause implosion. The loss of poll requests or responses is detected through timeouts, and results in the transmission of new requests.

The status kept by a sender about reception of data and its consumption at receivers is maintained through a sliding window scheme. PRMP extends this well-known concept to one-to-many multicast. Each of the GS receivers (R_i) keeps a receiving window (rw_i). The sender keeps a set of GS sending windows (sw_i), one per receiver. Status from the set of sw_i's is aggregated in a global sending window (sw). The feedback sent by a R_i (a response) contains a copy of rw_i, allowing the sender to update sw_i.

The protocol mechanisms for flow control, error control, and congestion control derive from the window scheme, and are applied independently in each level of the tree between a parent and its children. PRMP's flow control scheme prevents overrunning losses and the unnecessary retransmissions that would ensue: a parent only transmits new data when it can assure that all its children have a buffer allocated to receive such data (according to sw). Further, packet transmission is throttled by an inter-packet gap, or *ipg* (transmissions are separated by at least one *ipg*). This is a purposely conservative scheme, because it aims at saving network bandwidth at potential expense of throughput. The error control scheme saves bandwidth too, but by handling retransmission requests collectively. In reliable multicast, an arbitrary subset of receivers may require the retransmission of a given data packet; in PRMP a parent chooses between multiple unicasts and a single multicast retransmission according to the number of receivers requiring retransmission. Finally, the congestion control scheme is similar to Van Jacobson's TCP scheme ([Jacobson88]), superimposing a *congestion window* on top of sw in order to temporarily restrict the transmission of new data packets. Sliding window schemes in general scale poorly, because of implosion and amount of protocol state at the source. PRMP, however, is designed for scalability; the next section addresses scalability limitations and how PRMP overcomes them.

3 Scalability

As shown by [Pingali94], a simplified sender-initiated extension of TCP to window-based multicast will not scale. If the sender transmits a window of L packets to GS receivers, and every receiver returns an ACK upon receipt of a data packet, the sender would receive an "avalanche" of $L \times GS$ ACKs. The number of implosion losses that would result depends on many factors, such as currently available host and network capacity. Depending on the configuration, a simple sender-initiated multicast protocol can cause implosion in groups with even less than ten receivers ([Barcellos98a]).

The scalability of a protocol can be mainly evaluated according to the impact that group size has in the network cost and throughput of the protocol. Ideally, the protocol should behave as efficiently and as economically as unicast. For example, loss detection should be similar to unicast: recovery time should not greatly exceed one RTT between source and receiver, and only the subset of receivers that missed the packet are supposed to receive the retransmission. However, unicast throughput and cost cannot be achieved in practice due to a number of reasons, but mainly because the probability of a receiver missing a packet in a multicast transmission increases with group size (considering that losses have negative impact on throughput and network cost). There are other limiting aspects that may be linked with group size, depending on the protocol: amount of status at the source, feedback flow of ACKs, NACKs, or session information (e.g., membership changes in dynamic groups), etc. Finally, some protocols may impose certain requirements, such as knowing in advance data stream size, or storing the data in a disk, which are acceptable for a restricted class of applications only.

Group topology is another concern regarding scalability: large groups will be typically dispersed over several networks, possibly spanning the globe. RTTs between source and receivers may be very large (in the order of seconds). Bandwidth may be very scarce. In such scenario, throughput and cost might be seriously affected by error control design. Loss detection and recovery should be mostly isolated from the rest of the group (network).

As mentioned in Section 1, receiver-initiated protocols are highly scalable because their design makes the sender *independent from receivers,* that is, the sender does not control (or is aware of) the group members. In this model, the sender transmits to a group using a group identifier (an IP class D address) without waiting for ACKs; instead, when receivers detect a loss, they send a NACK (a retransmission request). Observe that a one-to-many reliable dissemination is inherently driven by the sender, since it is the sender, not receivers, that decides when to (re)transmit, to which receivers retransmit, and when to safely release packets from buffers. This brings limitations to receiver-initiated protocols, as explained below.

There are two ways of implementing error control: forward error control (FEC) or feedback-based error control. In the former case, redundant information is added to the data stream (typically at the end of the stream) to allow receivers to reconstruct lost data, while in the latter the sender keeps a copy of transmitted data in case it needs to be retransmitted. There are limitations with FEC, including the processing effort required to compute the codes and the kind of losses it fits (independent, non-correlated losses). The feedback-based error control in receiver-initiated schemes generally relies on probability: the sender keeps a packet for an arbitrarily long time, sufficient (in some probability) to allow any potential NACK to successfully reach the sender in time, despite being delayed or lost and retransmitted. As it is always possible that a delayed NACK arrives requesting the retransmission of a packet which has been already discarded from the buffers, receiver-initiated schemes *cannot provide full reliability.*

The lack of status at the sender also brings problems for flow control and congestion control mechanisms, as follows. The sender aims to generate a packet flow equals the consumption rate of the slowest receiver, so that it neither overruns receivers nor slows down unnecessarily. The sender cannot know the consumption rate of the slowest receiver because it does not keep information about receivers. To attack congestion, a reliable multicast protocol must reduce the load generated by the protocol whenever congestion is detected, and periodically probe for potentially available load otherwise. Further, it should behave similarly to TCP to achieve fairness among other multicast flows, as well as with TCP flows. In multicast congestion control, the sender would need to keep track of individual flows and behave conservatively according to the set of monitored flows (i.e., act to clear out congestion of the worst flow). In receiver-initiated protocols, it is hard for the sender to monitor flows because it does not know the membership and hence does not store status like flow information about individual receivers.

In conclusion, *the lack of status about receivers at the sender limits the protocol efficiency and increases its network cost*. The design of PRMP follows a different approach: the sender knows the membership and maintains status about receivers in a sliding window (see Section 4). The sender profits from this information to drive the transmission efficiently. Scalability is enhanced with polling feedback and hierarchy, as discussed below.

3.1 Polling feedback

To avoid implosion, the rate of incoming feedback must be uniformly distributed and not exceed a given threshold (an "implosion threshold"). PRMP's polling mechanism reduces and distributes the feedback through time, as follows. The sender *plans* the transmission of polling requests to receivers according to expected arrival times of triggered responses. When the planned time arrives, the request is sent. A polling request nominates a subset of receivers[3] to send feedback and may be transmitted in a control packet (POLL) or piggybacked onto a data packet (DATAPOLL). Upon reception of a packet containing a polling request nominating itself, the receiver unicasts a response (RESP) with a copy of its receiving window. The three fundamental aspects about the polling scheme are: when receivers need to be planned a poll, how this poll planning is done, and how the planned polls are carried out.

WHEN A POLL NEEDS TO BE PLANNED. The polling process is driven by the *need* to obtain feedback from receivers. The protocol plans the sending of a request to a given receiver to happen at an "adequate" time (see below) and records this information in a table. A receiver can only have a single poll planned at a time. There are three situations that prompt the polling of a receiver (i.e., this planning):

- *data (re)transmission*: to guarantee reliable delivery of a packet of a given sequence *seq*, the sender needs to receive from each of the receivers at least one response acking *seq*. When a packet is retransmitted to recover a loss experienced by a subset of receivers, only that subset needs to confirm its reception. So, whenever the sender transmits "new" data to a given set of receivers (potentially all), the receivers in the set which do not have a planned poll yet will be given one.

- *flow control*: when feedback reports that all packets of the window have successfully arrived at the receiver but none of them have been consumed by the application yet (full buffers), the sender must wait and periodically poll the blocked receiver until the receiver announces new free buffer(s) (i.e., that new data packets can be taken).

- *retransmission timeout*: packets carrying polling requests or responses can be lost. The sender must re-send a polling request to a receiver when the receiver's response fails to arrive in "reasonable" time (a retransmission timeout is calculated according to RTT estimates). So, when a timeout of a request/response pair occurs, the receivers that failed to respond to the request are planned a new poll (with priority over "standard" polls).

HOW THE POLL PLANNING IS DONE. To achieve uniform distribution of response arrivals, the poll planning scheme divides time into *epochs*, intervals of equal length, and associates a

[3] typically implemented with a bitvector.

(proportional) *response quota* to be allowed within each epoch. In Figure 2, quota is represented by boxes, filled or empty, and is equal to 5 per epoch. The sender maintains a vector to keep track of the number of responses that are being expected (i.e., have been assigned) in epochs ahead, so not to exceed this quota. In Figure 2, the current epoch is full, the next has 3 responses left, the third and fourth are full, and the fifth has 2 left. Examining from one RTT ahead, the sender allocates a response to the earliest epoch with available quota that it can find. In Figure 2, although epoch $x+1$ has quota available, the request/response round-trip time would not allow a response to be sent and received before epoch $x+2$. When allocating a response to an epoch, the response arrival may have to be delayed depending on the current occupation of epochs. In Figure 2, the arrival of the response has to be delayed until epoch $x+4$ (delay denoted as t_d); the transmission of the request (one RTT earlier) will be delayed accordingly, that is, in time t_d. So, the (transmission of the) poll is planned to occur at time t.

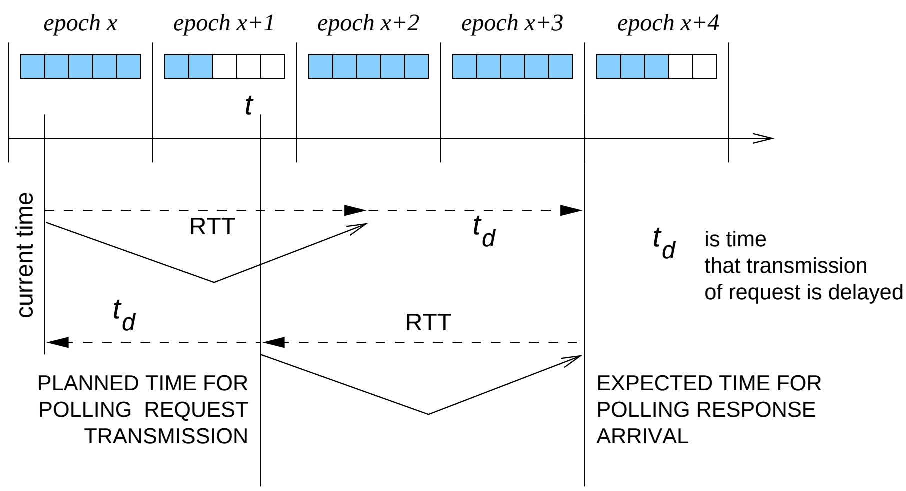

Figure 2: Example illustrating the poll planning scheme.

How the planned polls are carried out. After the above planning has been performed for a receiver R_i, the first outgoing data packet to be sent to R_i at or after t will carry a polling request piggybacked on it (DATAPOLL). At time t (or shortly after), if there is no outgoing data packet to send, a control packet (POLL) with the polling request is sent. The sender *periodically* examines the set of planned polls and sends those which are "due" ($clock \geq t$). More precisely, it checks for due planned polls (a) whenever a data packet is about to be transmitted (typically at every *ipg*), so to determine whether it can carry a piggybacked request, or (b) whenever there is no data to be transmitted, when the next, if any, planned poll will be due.

The epoch length and response quota will determine the feedback rate and its uniformity. Given a pre-set feedback rate, the shorther the epochs, the smaller the quota per epoch. The shorther the epochs, the more uniform the arrival of responses, but the larger the protocol state required to store the vector.

3.2 Hierarchic organization

The polling feedback suppression enhances the scalability of sender-initiated mechanisms, allowing PRMP to successfully extend the window-based one-to-one communication paradigm to one-to-many. However, with polling alone the scalability of PRMP remains limited, for the following reasons. Firstly, recall that the implosion avoidance mechanism reduces the flow of feedback to the sender so that no feedback packets are wasted because of implosion losses. However, feedback is required by the sender in order to slide the window (see Section 4). Since the feedback rate that the sender and its surrounding network can safely take is finite, there will be a given group size which will be large enough to make the employed response rate start blocking the window and become the bottleneck in the communication (unless a buffer the size of the data stream is used at all elements, like in RMTP [Paul97] and MFTP [Miller97]). From then on, throughput would decrease and network cost would increase (more POLL packets required). Secondly, the protocol state, enlarged with the polling mechanism, grows linearly with the number of receivers, eventually straining sender's resources. Thirdly, the polling scheme only addresses scaling issues related to group size, not topology; in wide-area networks, data and control packets may have to travel to and from distant receivers, making loss detection and recovery slower and more expensive. Finally, in wide-area multicasting, it is prohibitive to globally retransmit packets to all receivers when only a few tend to share the same loss.

As previously indicated, these scalability limitations are eliminated by PRMP with the help of hierarchy. Tree-based schemes generally scale well for reliable multicast ([Levine97]) because the responsibility for reliable delivery is placed not solely on the source but also on every parent in the tree. This decentralization of responsibility results in three major advantages that help promote scalability:

- *status*: the amount of protocol status which the source needs to keep about receivers is reduced;

- *implosion avoidance*: the amount of feedback packets flowing to the source is reduced as the number of receivers the source interacts with is reduced;

- *localized error control*: allows a receiver to recover losses from a nearby (parent) node rather than from the distant sender, thus speeding up recovery and reducing the network cost.

In PRMP's case, the tree structure is used not only for error recovery but also for propagation of data: a child node not only sends its responses to its parent node, but also receives data from its parent. (The source multicasts data to its children, not to the complete tree, and each of these children forward (via multicast) the data they received to their own children, and so on.) This arrangement has the advantage of *localized flow and congestion control*: when a parent node is in charge of forwarding packets to its children, it can swiftly adjust the transmission rate if a child appears to be experiencing losses. That is, since a parent does both forwarding of data packets and reception of feedback, it is in a better position to detect and deal with problems regarding its receivers (such as congestion) more quickly and effectively.

In this hierarchic distribution of processing, nodes may have a "sending role" (source at root), "receiving role" (leaf receivers), or both (internal receivers). In Figure 1, roles correspond to letters "S" and "R", respectively. The sending role is to transmit packets to receivers (child nodes), wait for acknowledgments, and retransmit packets if required; the receiving role is to receive packets from the sender (parent node), return acknowledgments when packets contain a polling request, and deliver data to a local receiving application if one is present. Note that only data is forwarded, not control information such as polling requests. The workings of PRMP, the interaction between a parent node and its children, is dictated by a sliding window mechanism, as shown in the next Section.

4 A Multicast Sliding Window

To save network bandwidth, a reliable multicast protocol must efficiently: (a) prevent unnecessary losses due to overrun receivers; (b) prevent unnecessary retransmissions due to false loss detection; (c) minimize reception of unwanted retransmissions at receivers. This is achieved in PRMP through a multicast sliding window scheme.

Recall from Section 2 that a parent keeps a sending window sw and each child R_i, a receiving window rw_i. All windows are of length L packets. A rw_i is characterized by the following attributes: *left* and *right edges* (le and re, respectively); the next expected data packet (ned); the *highest received* packet (hr); and a bitvector v (of length L). The value of ned is the next yet-to-be-received data packet (not necessarily missing). The left edge le is either the next packet yet to be received (equals ned) or the earliest unconsumed packet (packet has been received but application has not consumed it yet); the right edge re is always equal to $le+L-1$ (re is used for illustration purposes only). The highest referenced sequence, hr, represents the knowledge of R_i about which packets have been transmitted by the parent so far. The boolean vector v is indexed by packet sequence seq, though only the packets with sequence seq such that $le \leq seq < le + L$ are "directly represented" in v. Packet $v[ned]$ is absent by definition ($v[ned] = 0$); for all packets $ned < seq \leq hr$, R_i has received the data packet seq if $v[seq] = 1$; the receiver is unaware of any packets $seq > hr$.

The sender, on its turn, keeps a set of GS sending windows, one sw_i for each R_i. The sending window sw_i is the sender's latest knowledge of rw_i at R_i. Like rw_i, it is characterized by the following sequence numbers: *left* and *right edges* (le and re); the *next expected acknowledgment* (nea); the *highest referenced* sequence (hr); and a bitvector v (of length L). The attributes $sw_i.le$, $sw_i.re$, $sw_i.hr$ and $sw_i.nea$ are the sender's knowledge of $rw_i.le$, $rw_i.re$, $rw_i.hr$ and $rw_i.ned$, respectively. For any seq such that $sw_i.nea < seq \leq sw_i.hr$, $sw_i.v[seq] = 1$ indicates that R_i has acked data packet seq; packets that have not been acked may have been nacked or not (see error control below). When the sender receives a response packet (RESP) from R_i, it updates its variables related to R_i as follows: $sw_i.le \leftarrow \max\{sw_i.le, \text{RESP}.rw.le\}$, $sw_i.hr \leftarrow \max\{sw_i.hr, \text{RESP}.rw.hr\}$ and only then, for all seq, $\text{RESP}.rw.le \leq seq \leq \text{RESP}.rw.hr$, $sw_i.v[seq] \leftarrow sw_i.v[seq] \vee \text{RESP}.rw.v[seq]$.

The set of GS sw_i's is aggregated in a global window, sw, which inherits all attributes of an sw_i apart from v, but on the other hand adds three new attributes: hs, $Acked_{seq}$, and $Nacked_{seq}$. The value of $sw.hs$ represents the highest data packet (sequence) sent so far. $Acked_{seq}$ and $Nacked_{seq}$ are receiver sets compiled on demand from the set of sw_i's, representing the subset of receivers that have acked seq and nacked seq, respectively. The sw attributes le, re, and nea are compiled upon demand as follows: $sw.le = \min\{sw_i.le\}$, $sw.re = \min\{sw_i.re\}$, $sw.nea = \min\{sw_i.nea\}$. The value of $sw.nea$ represents the first non-fully acked packet, whereas $sw.re$ the highest packet that can be safely received by all children.

The above multicast sliding window scheme is the core of the error, flow, and congestion control mechanisms of PRMP. Below, the description of the window scheme is extended while such mechanisms are discussed.

4.1 Error Control

Recall that the loss of control packets (poll requests and responses) is distinguished from the loss of data packets. The former is detected through timeouts and simply recovered by sending a new poll request to the receivers that failed to respond. To set a proper timeout, the sender estimates the RTT between itself and each receiver; for that purpose, it includes a timestamp when sending a poll request, to be returned unmodified within all generated responses.

The design of the error control scheme for data losses is based on the fact that a response from receiver R_i brings, through a copy of rw_i, **multiple** ACKs and NACKs **together** (besides acking all packets before $rw.ned$). At any point in time, it is possible (likely) that a parent is expecting two or more RESP packets from a given receiver. So, a packet of sequence seq may be acked and nacked multiple times, and in arbitrary order because of network reordering.

Data losses are detected through "identification" of NACKs in the rw_i that exist in a response. ACKs in a response are easy to identify, being the 1's in v: $\text{RESP}.rw.v[seq] = 1$. NACKs, however, depend on the *causal relation* between (re)transmissions and responses: a given response RESP can only ack or nack packets that have been (re)transmitted *before* (or with) the poll request that triggered RESP. In other words, receivers need a chance to be polled before they can ack or nack a given packet transmission. The mechanism employs the value of $sw.hs$ and timestamps to accomplish that, as follows. First consider the simplified case of sequential transmission of data, which occurs at the source. The value of $sw.hs$ is included with every poll request to make receivers aware of which packets should have been received. A receiver updates $rw_i.hr$ with the maximum sequence received (using the copy of $sw.hs$ in the request). This allows the sender to infer from sw_i that R_i has, for any seq such that $seq \leq sw_i.hr$, acked packets with $sw_i.v[seq] = 1$ and nacked those with $sw_i.v[seq] = 0$. All packets in RESP.$rw.v$ with seq such that $sw_i.hr < seq \leq sw.hs$ were sent *with* or *after* the poll that triggered RESP.

In the hierarchy of PRMP nodes, the source is the only sender that is guaranteed to transmit data packets in order, since the sending application will be locally generating a data stream. Internal nodes, in contrast, are allowed to forward data packets out-of-order: if $seq+1$ has been received ($rw[seq+1] = 1$) but not seq ($rw[seq] = 0$), and $seq+1$ can be safely stored by all children ($seq+1 \leq sw.re$), then packet $seq+1$ is multicast. In these circumstances, the value of $sw.hs$ still indicates the highest packet transmitted, but becomes insufficient to identify NACKs. This design decision increases throughput but complicates error control. To verify the causal relation between a response RESP and the most recent (re)transmission of seq, the sender records (re)transmission times (t_{seq}) of all packets such that $sw.nea \leq seq \leq sw.hs$ and, when required, compares t_{seq} with the timestamp in RESP (denoted as RESP.ts, it is the same transmission time that is employed in RTT estimation). When a response is received, the NACK seq are any seq such that: RESP.$rw.v[seq] = 0 \wedge seq \leq$ RESP.$rw.hr \wedge t_{seq} \leq$RESP.$ts$. If not a NACK seq, it is because seq may have been (re)transmitted *after* the poll that generated the response or not even sent yet.

An example of error control involving a three-level tree is illustrated in Figure 3: it shows the reliable transmission of data from the source to an internal receiver and the forwarding that takes place from the internal receiver to a leaf receiver. The parent of receiver R_s, the source S, transmits four packets: DATA.$seq = 31$, DATA.$seq = 32$, DATA.$seq = 33$, which is lost by the network, and DATAPOLL.$seq = 34$, which requests a response from R_s to acknowledge all four packets. Packet DATA.$seq = 31$ arrives at R_s and soon is forwarded as DATAPOLL.$seq = 31\, hs = 31$. DATA.$seq = 32$ also arrives at R_s and is forwarded, at around time 80, as DATA.$seq = 32$. DATAPOLL.$seq = 34$ arrives at R_s, and is forwarded to $R_{s,i}$ at time 90 as DATAPOLL.$seq = 34$; note that packet $seq = 33$ is currently missing at R_s and thus has not been forwarded yet (t_{33} is set to ∞). Just before time 110, a retransmission of $seq = 33$ from S arrives at R_s, and is forwarded to $R_{s,i}$ as DATA.$seq = 33$; R_s sets its t_{33} to 110. Soon after forwarding $seq = 33$, R_s receives a response from $R_{s,i}$ with RESP.$rw.v[33] = 0$, RESP.$rw.hr = 34$ and RESP.$rw.ts = 90$; R_s compares the timestamp in the response, RESP.$ts = 90$, with $t_{33} = 110$, and finds out that this response does not reference $seq = 33$ (poll/response pair at 90 precedes transmission at 110). The response *does* reference, however, packets $seq = 31$, $seq = 32$, and $seq = 34$, which have all been transmitted before or with the polling request at time 90. If evaluated by R_s, $Nacked_{33}$ will not contain $R_{s,i}$, but $Acked_{31}$, $Acked_{32}$ and $Acked_{34}$ will.

RECOVERING LOSSES COLLECTIVELY. Once a NACK seq has been identified in a response, the loss of seq (by a given R_i) needs to be recovered through a retransmission and eventually a request/response pair. Such data retransmission may be delayed in order to collectively handle the loss of a packet seq which was multicast. This delay is part of a configurable "wait-and-see" mechanism which aims to select the best choice in terms of retransmission according to the proportion of receivers (out of GS) that requested recovery. The more receivers require a retransmission, the more advantageous is to re-multicast the packet; however, isolated losses are better treated with multiple unicast retransmissions.

The recovery process of seq is triggered by the arrival from any of the receivers of the first NACK seq (i.e., $Nacked_{seq} \neq \phi$), and persists until seq becomes fully acked (i.e., $Acked_{seq} = all$).

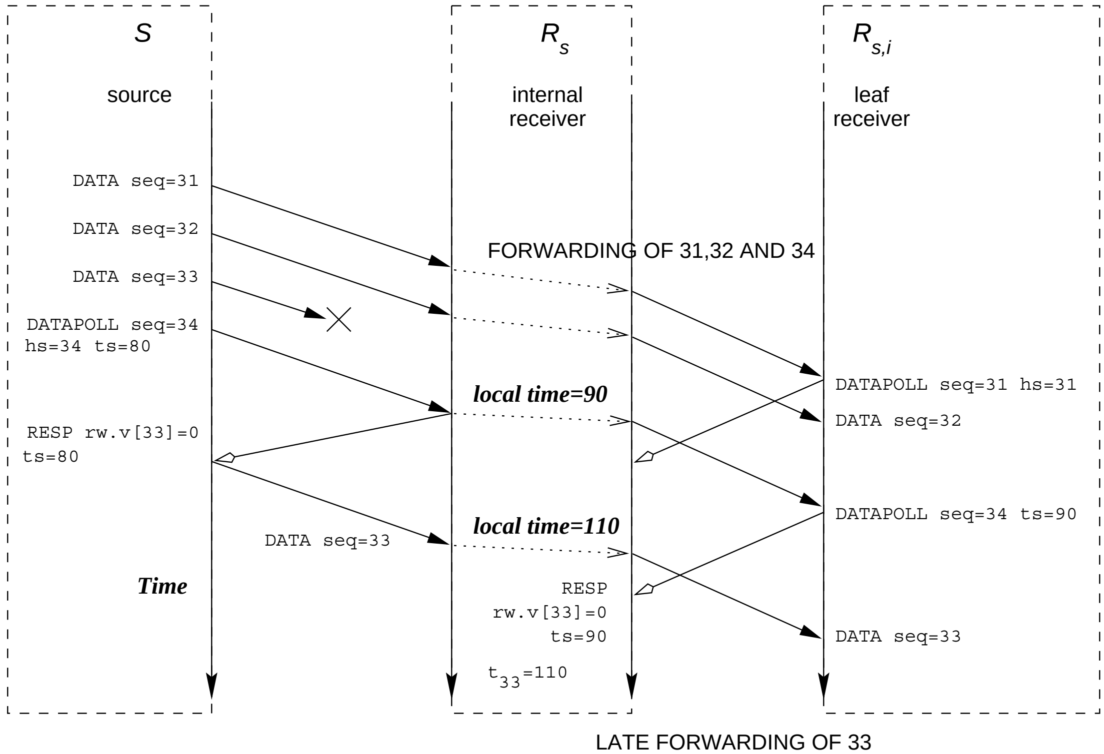

Figure 3: Example of communication involving three levels: S, R_s, and $R_{s,i}$.

It has two stages, *collection* and *retransmission*; collection stage consists in waiting for the potential arrival of responses from other receivers with a NACK to the same packet. When collection ends, there is a retransmission, and retransmission stage starts. The switch from collection to retransmission is triggered by one of the two conditions below:

(c1) the sender has collected a number of NACKs for *seq* which is sufficient to justify a multicast retransmission (when cardinality of $Nacked_{seq}$ exceeds some threshold);

(c2) the sender has collected (from all receivers) responses with ACKs and NACKs regarding *seq*, which means that there will be no additional NACKs for *seq* ($Nacked_{seq} \cup Acked_{seq} = all$).

Conditions (c1) is tested whenever *seq* is nacked for the first time by a given receiver, while (c2) is tested (after (c1)) whenever *seq* is acked or nacked for the first time by a given receiver. If (c1) is true, the packet is re-multicast. If (c1) is false but (c2) true, the sender performs multiple unicast transmissions, one for each receiver that has requested recovery. If both are false, there is no progress of stage. After the retransmission, *seq* stays in retransmission stage until fully acked. If retransmissions themselves are lost, the collection stage is not repeated: NACKs of retransmissions are dealt with swiftly with unicast retransmissions.

4.2 Flow Control

One of the purposes of the sliding window scheme is to record current allocation of packets to buffers. The range of packets that can be present in the buffers is delimited by left and right edges. For a receiving role, the progress of $rw_i.le$, and thus of $rw_i.re$, will be dictated by the sequential consumption of data by the local receiving application. For a sending role, the sw_i will slide forward according to the consumption of data reported by R_i and other children through $rw_i.re$ in responses. R_i's parent uses $sw.re$, the smallest right edge reported by any child, to determine the highest transmittable packet. So, before multicasting *seq* the parent makes sure that all receivers have a buffer readily available for *seq* (i.e., $seq \leq sw.re$). This scheme is an extension of TCP's window-based flow control: a conservative approach intended to prevent *any* overrun losses.

In internal nodes, the buffer of L packets is shared between sending and receiving roles, making the sending role's sw and receiving role's rw interdependent. Any non-fully acked packet *seq* in sw has to be kept in the buffers of the internal receiver because it might have to be retransmitted by the sending role. Thus, rw cannot slide forward if packet $seq = rw.le$ cannot be released because *seq* has not been fully acked in sw (i.e., $Acked_{seq} \neq all$ and $seq = sw.nea$). In other words, the internal node can only release a packet from the buffers when it has been both consumed by the local application (if present) *and* fully acked. When the receiving role of an internal node reports the highest receivable packet to its parent, it subtracts from the normal value ($rw_i.re$) the number of packets in the buffer that have to kept because of the sending role (packets not fully acked).

If the consumption stops at a child node, the sending window of its parent will get eventually stuck (when all packets have been fully acked, but child reports that there is no space for a new packet to be received). This flow control scheme ensures that if a given receiver "falls back" (is slow), there will be "backpressure" through multiple levels towards the source.

4.3 Congestion Control

PRMP embodies two distinct congestion control schemes to deal with congestion in communications involving large internetworks: rate-based and window-based[4]. The window-based scheme is an adaptation of the Van Jacobson's congestion control scheme used in TCP (in fact, this discussion applies particularly to the Internet context). Based on the assumption that most losses in the Internet are caused by queue overflow in congested routers, and not by packet corruption ([Jacobson88]), congestion is detected through packet losses reported by receivers. A response with a NACK is seen as "hint" of congestion somewhere between the parent and a child. When a given packet is nacked for the first time ($Nacked_{seq} \neq \phi$) it indicates congestion, and thus results in load reduction; when a packet becomes fully acked ($Acked_{seq} = all$), it indicates successful transmission and results in load increase.

To vary the load, the transmission of new data packets is restricted by a "congestion window" ($sw.cwnd$). The value of $sw.cwnd$ is subject to *multiplicative decrease* (divide by 2) when the detection mechanism indicates congestion, and *to additive increase* (add $1/sw.cwnd$) when the detection mechanism indicates potential available load. Congestion control adds to flow control and alters the way the highest transmittable packet (i.e., the right edge of the sending window) is calculated, reducing $sw.re$ so that the transmission of new data packets is refrained. Because PRMP employs selective retransmission, only those packets in sw which are not acknowledged are counted as outstanding data. POLL packets (which tend to be small) are unaffected by the congestion window.

Like TCP, PRMP employs *slow start*. The value of $sw.cwnd$ starts with 1 packet, and during slow start, it is increased by 1 packet every time a packet gets fully acked, until a NACK is received (a packet in sw has been nacked for the first time). This is the point when the detection mechanism indicates that the right load has been reached, leading the value of $sw.cwnd$ to be halved, and then onwards additively increasing the value of $sw.cwnd$ by 1 packet every window of data. PRMP employs "fast recovery" ([Stevens94]): slow start is applied only at the beginning of a session, and not after every loss.

[4] due to space restrictions, only the window-based scheme is presented; please see [Barcellos98c] for a description of the rate-based scheme.

5 Related Work

Many reliable multicast protocols have been proposed in the recent past. This section briefly compares PRMP with three other representative reliable multicast protocols: SRM, RMTP, and MFTP.

The SRM-Scalable Reliable Multicast Protocol ([Floyd95]) follows the *Application Level Framing* approach and is supposed to be part of a many-to-many multicast application. PRMP, in contrast, is a generic one-to-many protocol. The error control mechanism of SRM is based on the multicasting of NACKs and retransmissions to the entire group; any receiver is able to retransmit as long as it has the packet currently stored. One one hand, this can greatly reduce packet loss recovery times, but on the other hand tends to flood the network with unwanted packets, specially if the number of shared losses is small. To prevent an "explosion" of redundant NACKs and retransmissions, receivers run a distributed random-based suppression scheme, which reduces redundancy but may inflict substantial overhead as it requires each node to periodically estimate the RTT between itself and all other nodes. Even if the suppressing mechanisms achieves *perfect* random delays, and there is no loss of feedback packets, the *best* SRM scheme can achieve in terms of cost is 2 multicast operations (1 NACK and 1 retransmission) per recovery. As discussed in Section 3, the larger the group, the higher the probability a given packet will require recovery (by one or more receivers). For example, in a study of the Mbone by [Yajnick96], 47% of transmissions in the experiment required recovery. PRMP, instead, handles packet losses (a) hierarchically: a parent will recover losses experienced by nearby children and, (b) individually: a parent will use unicast to recover losses that a few children experienced. These SRM shortcomings have been recognized and are being addressed in SRM by adding hierarchy to its symmetric structure ([Sharma98]).

RMTP-Reliable Multicast Transport Protocol ([Lin96],[Paul97],[Buskens97]) is more similar to PRMP. RMTP is a one-to-many protocol that relies on hierarchy and periodic (timer-based) transmission of feedback from child receivers for enhanced scalability. RMTP is organized as a two-level logical tree: receivers are grouped into "local regions", each with a special receiver, the "Designated Receiver" (DR). The source, at the root of the multicast tree, employs IP multicast to send data packets to *all* receivers (including DRs) in the tree (unlike PRMP). A receiver or DR sends feedback only to its parent, and it does that periodically, according to a timer. To build the logical tree, receivers choose their parent DR autonomously, and the parent node, source or DR, does not know the child nodes it parents. That is, RMTP is receiver-reliable, and this affects the design of RMTP error and flow control mechanisms, as below.

The error control mechanism of RMTP is based on the fact that the sender transmits up to a window of data packets and then waits for a period of time which should be long enough to allow all potential receivers out there to report losses. After this period, the source or DR advances the window to the lowest packet sequence which was NACKed, and retransmits reported losses. Hence, in RMTP it is possible that the sender receives retransmission requests for packets that have been "left behind" in the window, and thus have been discarded. RMTP overcomes such problem by requiring the sender and all DRs to store *(cache)* all data, irrespective of the size of the stream being transmitted (i.e., "infinite buffers" abstraction). So, in RMTP all nodes must store all data in their disks in order to achieve *full reliability*. Disk operations may, of course, affect performance. In PRMP there are no such requirements, and communication takes place using the memory which is made available for a parent and its children. Parents in PRMP know their children, so that they do not need to wait for NACKs that might exist before advancing the window; as soon as all children have responded (and a packet becomes fully acked), the parent can safely slide the window.

The Multicast File Transport Protocol, or MFTP ([Miller97]), is designed for uploading files to multiple receivers. The unique aspect of MFTP is the way it organizes a transmission: the file to be transfered is sent through multiple "passes". Before transmitting, the file is logically divided in "blocks", and blocks in "data transfer units" (DTUs). A feedback packet which is sent by a receiver contains a bit vector which refers to all DTUs within a given block. In the first pass, the entire file is sent block after block. At the end of each block, the sender multicasts a "Status Request" message identifying the current pass and block. Like a poll, this request allows receivers to return a response requesting the retransmission of packets within the identified block; however, a receiver *only* sends a response for the block if there were losses. The sender does not wait for such responses, immediately proceeding to the next block. The first pass ends with the transmission of the last block. If one or more responses requesting retransmissions have arrived during the first pass, the sender starts a second pass in which it re-multicasts all packets which have been nacked by one or more receivers. For each block, the sender checks if there were losses reported; if so, it retransmits all nacked packets within the block, and then multicasts a Status Request to allow receivers to negatively acknowledge the retransmissions (in case retransmissions themselves are lost). When the last block has been processed, the sender checks if any retransmission requests have been received; if so, it starts a third pass, and this continues until no response is received. When there is a pass where no response (thus retransmission request) has been received, the sender multicasts a Status Request regarding *all blocks*, and waits on a (user-defined) timer. A receiver which misses some data packets but successfully receives such Status Request message transmits to the sender as many responses as there are blocks with missing packets. When the timer expires, the sender starts a new pass to retransmit missing data. Otherwise, if the timer expires without responses, the sender sends a termination message to all receivers (such message varies according to the group model being employed, see [Miller97] for details).

Comparing with PRMP, MFTP uses "poll-all" messages in order to allow receivers to request retransmissions; unlike PRMP, receivers may remain silent (if they did not experienced any loss). Though this reduces the risk of ACK-implosion, it allows the sender to wrongly infer that there were no losses if no NACK is received. Further, if losses are correlated and a great number of receivers experiences at least one loss within the same block, the sender may still be imploded. MFTP does not include flow control: the sender employs a fixed transmission rate throughout the transfer; if one or more receivers are being overrun by the sender, they will report losses, which will lead to retransmissions, and to waste of network bandwidth and increase in end-to-end latency. There is no congestion control either. If one or more of the flows passing through routers that are or become congested, a large amount of packets may be dropped. Even if receivers report losses to the sender, the sender keeps transmitting at the same pace. Like RMTP, MFTP requires that all I/O devices involved allow random access. Finally, MFTP cannot guarantee full reliability: the sender transmits a Status Request and waits for responses during a given time; if the request or the response is lost, the sender will wrongly assume that *all is well*. This is different than RMTP, in which receivers send feedback periodically so that a long wait delay at the sender may allow multiple feedback packets to be sent, increasing the probability that one or more feedbacks reach the sender. Unlike PRMP, in MFTP there is no retransmission of request or response.

6 Concluding Remarks

This paper presented PRMP. PRMP's unique implosion avoidance mechanism polls receivers at carefully planned timing instants achieving a low and uniformly distributed rate of feedback packets. The sender retains controls of receivers: the main PRMP mechanisms are based on a one-to-many sliding window mechanism, which efficiently and elegantly extends the abstraction from reliable unicasting to reliable multicasting. The error control mechanism of PRMP incorporates the use of NACKs and selective, cumulative acknowledgment of packets; additionally, it can wait and judiciously decide between multicast and selective unicast retransmissions. The flow control mechanism prevents unnecessary losses caused by the overrunning of receivers, despite variations in round-trip times and application speeds. Congestion control reduces the number of losses in case of network congestion and allows network conscious multicast transmission.

The scalability provided by the polling mechanism is further extended by an hierarchic organization to exploit distributed processing and local recovery: receivers are organized according to a tree-structure. Unlike other tree-based protocols, PRMP is "fully-hierarchic": each parent node forwards data via multicast to its children, and retains/explores the control of and knowledge about its children while autonomously applying error, flow and congestion controls in the communication with them.

This paper only provides an informal comparison with similar protocols. Simulation experiments are desired in order to *quantitatively* compare PRMP with other reliable multicast protocols. This is hard, however, because it is necessary to implement all protocols under the same simulation environment; also, because each protocol has protocol-specific input values that have to be tuned for best performance; finally, there is no such thing as "typical" scenario in the Internet, and protocols may perform differently according to the group size and topology. Despite these difficulties, we intend to develop a simulation study to compare these protocols using either the widespread "ns" network simulator ([ns]) or the multicast-oriented network simulation environment described in [Barcellos98c].

Acknowledgments

The research I described in this paper was developed while I was at Newcastle University, UK, and has been funded by CAPES through grant #0345/94. I would like to thank Dr. Paul Ezhilchelvan, for his invaluable help in guiding my research work.

References

[Bagnall97] P. Bagnall, B. Briscoe, A. Poppitt, "Taxonomy of Communications Requirements for Large-scale Multicast Applications", 21 Nov 1997, Internet Draft, http://www.labs.bt.com/people/briscorj/projects/lsma/taxonomy-reqs.txt

[Barcellos98a] M. Barcellos & P. Ezhilchelvan, "A Reliable Multicast Protocol using Polling for Scaleability". In Proc. of IEEE INFOCOM'98 (The Conference on Computer Communications), San Francisco, 29 March-2nd. April 1998.

[Barcellos98b] M. Barcellos & P. Ezhilchelvan, "A Scalebale Polling-based Reliable Multicast Protocol". In Proc. of HIPPARCH'98 (Fourth International Workshop on High Performance Protocol Architectures), UCL, London,15-16 June 1998.

[Barcellos98c] M. Barcellos, "PRMP: A Scaleable Polling-based Reliable Multicast Protocol". Ph.D. Thesis, Department of Computing Science, Newcastle University, Newcastle upon Tyne, October 1998, 200pp.

[Buskens97] R. W.Buskens, M.A. Siddiqui, S. Paul, "Reliable Multicast of Continuous Data Streams", Bell Labs Tech. Journal, Spring 1997, pp.151-174.

[Crowcroft88] J. Crowcroft, K. Paliwoda, "A Multicast Transport Protocol", ACM SIGCOMM'88, Stanford, 16-19 Aug. 1988.

[Deering91] S. Deering, "Multicast Routing in a Datagram Internetwork", PhD Thesis, Stanford University, Dec. 1991.

[Floyd95] S. Floyd, V. Jacobson, S. McCanne, C. Liu, L. Zhang, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing", ACM SIGCOMM'95, Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication. Aug. 28 - Sept. 1st, Cambridge, USA.

[Grossglauser96] M. Grossglauser, "Optimal Deterministic Timeouts for Reliable Scalable Multicast", IEEE INFOCOM'96, San Francisco, California, March 1996.

[Hofmann96] M. Hofmann, “A Generic Concept for Large-Scale Multicast”, Proc. of Intl. Zurich Seminar on Digital Communications, IZS’96, Zurich, Switzerland, Springer Verlag, Feb. 1996.

[Holbrook95] H. Holbrook, S. Singhal, D. Cheriton, “Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation”, ACM SIGCOMM’95, Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication. Aug. 28 - Sept. 1st, Cambridge, USA.

[Hughes94] L. Hughes, M. Thomson, “Implosion-Avoidance Protocols for Reliable Group Communications”, Proc. of 19th Conf. on Local Computer Networks, Minneapolis, Minnesota, October 1994.

[Jacobson88] V. Jacobson, “Congestion Avoidance and Control”, Computer Communication Review, vol.18, no.4, pp.314-329, Aug. 1988.

[Levine97] B.N. Levine, J.J. Garcia-Luna-Aceves, “A Comparison of Reliable Multicast Protocols”, ACM Multimedia Systems Journal, August 1998 (accepted for publication)

[Lin96] J. Lin, S. Paul, “RMTP: A Reliable Multicast Transport Protocol”, IEEE INFOCOM’96, 24-28 March 1996, San Francisco, pp.1414-1424

[Miller97] K. Miller, K. Robertson, A. Tweedly, M. White, “Starbust Multicast File Transfer Protocol (MFTP) Specification”, (expired) Internet Draft, January 1997.

[ns] ns network simulator web site, http://mash.cs.berkeley.edu/ns/ns.html

[Paul97] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharryya, “Reliable Multicast Transport Protocol (RMTP)”, IEEE Journal on Selected Areas in Communications, Vol. 15 No. 3, April 1997, Pages 407-421.

[Pingali94] S. Pingali, D. Towsley, J. Kurose, “A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols”, Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems, Nashville, May 16-20, 1994.

[Sharma98] P. Sharma, D. Estrin, S. Floyd, and L. Zhang, “Scalable Session Messages in SRM”, Technical report, February 1998.

[Stevens94] W. R. Stevens, “TCP/IP Illustrated, Vol. 1: The Protocols”. Chapter 21: TCP Timeout and Retransmission, Addison-Wesley Professional Computing Series, Addison-Wesley, 1994.

[Yajnick96] M. Yajnick, J. Kurose, D.Tosley, “Packet Loss Correlation in the Mbone Multicast Network”, UMCASS CMPSCI Technical Report 96-32.