UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

PRMP: A Scaleable Polling-based Reliable Multicast Protocol

by

Antônio Marinho Pilla Barcellos

NEWCASTLE UNIVERSITY LIBRARY
098 50584 3
Thesis L6216

Ph.D. Thesis

Newcastle upon Tyne,

September 1998

Abstract

Traditional reliable unicast protocols (e.g., TCP), known as *sender-initiated* schemes, do not scale well for one-to-many reliable multicast due mainly to implosion losses caused by excessive rate of feedback packets arriving from receivers. So, recent multicast protocols have been devised following the *receiver-initiated* approach: scalability (in terms of control traffic, protocol state and end-systems processing requirements) is achieved by making the sender *independent from receivers*; the sender does not know the membership of the destination group. However, this comes with a cost: the lack of knowledge about and control of receivers at the sender has negative implications with respect to throughput, network cost (bandwidth required), and degree of reliability offered to applications.

This thesis follows an alternative approach: instead of adopting the receiver-initiated scheme, it *greatly enhances* the scalability of the sender-initiated scheme, by means of *polling-based feedback* and hierarchy. The resulting protocol is named PRMP: Polling-based Reliable Multicast Protocol. Its 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.

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. However, 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, congestion and session controls in the communication with them. Two congestion control mechanisms, one window-based and another rate-based, have been incorporated to PRMP.

As shown through simulation experiments, the resulting protocol achieves high thoughput with cost-effective reliable multicasting. They also show the scalability and effectiveness of PRMP mechanisms. PRMP can achieve reliable multicast with the same kind of reliability guarantees provided by TCP but without incurring prohibitive costs in terms of network cost or recovery latency found in other protocols.

I would like to dedicate this thesis to my wife,

Cláudia,

and to my parents,

Tuca & Marilisa.

Acknowledgments

I would like to express my sincere gratitude to several people who have contributed in various ways to the completion of this thesis. First and foremost, I thank my supervisor, Dr. Paul Ezhilchelvan, who has *significantly* contributed to my research, in particular through the discussions during the frequent meetings we held between January 1997 and June 1998. His expertise in distributed systems and group communication protocols was fundamental for my learning, as well as the development of my research work and thesis.

I am also very much indebted with Dr. Larry Hughes, who acted as my co-supervisor between January and June 1996, while spending a sabbatical in Newcastle. He introduced me to the "implosion problem" and suggested the extension of the polling-based implosion avoidance scheme to hierarchic.

I shall never forget the support and love shown by my wife, Cláudia, who has been with me (and *without me*) since this life project began (1984). Nothing will bring back nor replace the many hours that we were apart because I was busy studying or working. This PhD would not have been possible if I did not have her understanding, help and support. Thanks also to my parents, who have always understood and supported me in my quest.

I am indebted with Martin Beet; the text of the thesis has significantly improved after his careful review. I am also grateful to Cong-yue Liu and Avelino Zorzo, who read parts of the thesis. The research work and the writing of the thesis depended on having strong logistic support. In particular, I thank Jim Wight, who was always promptly able to help. I also thank Tim Smith, whose help *included* preparing a Linux machine in record time (this was essential for the simulation work). Thanks also are due to other staff members, in particular Dr. Mark Little, for the support regarding his C++SIM package, Prof. Santosh Shrivastava, Shirley Craig, Ron Kerr, Savas Parastatidis (and the *leys* cluster) and Andrewena Swainston.

Throughout my time in Newcastle, I have met many friends and colleagues who have made my time here in Newcastle very joyful. Although omitting important names, I shall mention Sérgio/Patrícia Cavalcante, Eduardo Figueiredo, and in particular, Avelino/Mári Zorzo and Martin/Ulrike Beet. Thanks to the footy group, including Barry Hodgson, Nick Cook, Sascha Romanovsky and Dave Hartland.

Many thanks to my employer in Brazil, UNISINOS- Universidade do vale do Rio dos Sinos (and Prof. Aretê Porciúncula deÀvila), which has invested on me and allowed me to be on leave from the Informatics Department for the last four years.

Last but not least, my sincere thanks to the Brazilian Research Agency CAPES (and to the Brazilians who pay its bill), which provided financial support (grant number 547/94-8) in the period 1994-1998. This amounts to approximately £70,000 (U$120,000) among university fees and living expenses.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation: Scaleable Reliable Multicasting

Multicast allows the efficient transmission of packets to potentially large sets of receivers. Packets are carried (routed) to receivers through a multicast (routing) tree which is set up by the network. As shown in Figure 1.1, this tree has the sender (also known as the "source") as the root node and receivers as leaf nodes (as well as network routers as internal nodes). Packets are replicated at each non-leaf node so that a copy of the packet follows each downstream route. Hence, each packet transmitted by the source crosses an edge of the tree only once on its way to receivers. The essential advantage of multicast over the simpler alternative of multiple unicast transmissions is the potential gain in network cost (or bandwidth). To illustrate the point, consider a complete d-ary tree of height h (as in Figure 1.1). 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 $d^h \times h$. In contrast, using multicast, as each edge is only traversed once, the network cost is equal to the number of edges: $\sum_h^{i=1} d^i$. Figure 1.1, for example, shows a ternary tree with 27 receivers, for which unicast costs 81 ($3^3 \times 3$), while multicast costs 39 ($3^1 + 3^2 + 3^3$), a percentage gain of 51%.

This gain of multicasting can be realized in the Internet by the IP multicast architecture ([Deering91]). The popularization of the IP multicast created the potential for the development of new multicast applications. Examples are software distribution, dissemination of web-pages

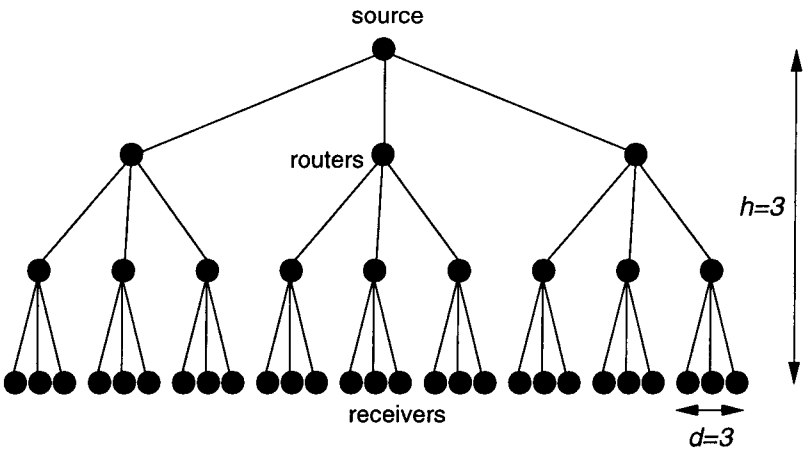


Figure 1.1: Example of multicast tree where tree is complete with degree 3 and height 3.

by WWW-servers, off-line video distribution, live audio/video stream broadcast (e.g., in the Mbone), and multimedia remote conferencing. These applications have different demands regarding reliable multicasting, and require individualized solutions or protocols to tackle them (often referred to as "one size does not fit all").

The subject of reliable multicast is not new. Several reliable multicast protocols, like Isis ([Birman91]) and Newtop ([Macedo94]), to name a few, have been proposed and developed in the past decade. These protocols have focused on high-level services such as "causal" and "total ordering" properties typically required by many-to-many distributed applications, as well as "atomicity" for those requiring fault tolerance. Because of these high-level services they provide, the protocols have high network cost (frequent message passing), substantial processing cost and protocol state. Hence, they do not scale well for larger number of receivers. Furthermore, part of these protocols have been developed having an underlying *broadcast* network in mind, and so the design choices do not directly extend to wide-area networks (e.g., topological issues, long latency delays, congestion control). Other protocols of this category require an underlying transport-level reliable multicast protocol capable of providing efficient and lossless one-to-many delivery (such as the one described in this thesis).

To overcome these limitations, the research emphasis has more recently shifted to the *scalability* of multicast protocols, addressing at network and transport levels the effect of group size on throughput, network cost, and amount of protocol state. One fundamental scalabil-

ity problem related to multicast is that of "ACK-implosion[1]" ([Crowcroft88]): the sender may be overwhelmed by acknowledgments (or "feedback packets") sent by receivers in response to a multicast transmission. As shown through simulations, implosion losses may occur for groups as small as 10 receivers. Further, as these protocols are applied to internets, they need to explore the topology and provide congestion control. Several reliable multicast protocols have been developed to attempt to realize large-scale multicasting in the Internet. Representative examples include LBRM ([Holbrook95]), SRM ([Floyd95]), RMTP ([Paul97]), and TMTP ([Yavatkar95]). RMTP is the only one of these protocols which has some congestion control mechanism, and it is also the closest in purpose to the protocol presented in this thesis.

In order to achieve scalability, most reliable multicast protocols have shifted the responsibility for ensuring reliable delivery from the sender to the receivers. In such scheme, receivers do not return positive acknowledgments ("ACK") when they successfully receive packets, but only negative acknowledgments ("NACK"), when a retransmission is necessary to recover a loss. Further, the sender does not need to maintain group membership information, so that the amount of state kept by the sender is independent of group size, allowing the protocol to cope with dynamically changing groups. The sender, however, is not able to guarantee that all receivers successfully received all packets due to the lack of positive acknowledgments: when no feedback is received the sender cannot distinguish between a persistent fault and successfull delivery. There are different ways of achieving reliable delivery under these "loosely-coupled" circumstances, including the use of probability and forward error correction.

1.2 Problem Definition (goals)

The research described in this thesis aims at developing a scaleable reliable multicast protocol and gain insight in related error control, flow control, congestion control, and session control schemes. The protocol to be described has the following attributes and requirements:

- there is a single "sending application" which disseminates data to a group of "receiving applications";

[1] also known as "feedback implosion".

- receiving applications cannot tolerate the loss of any data;
- the sending application has a list of receiving application addresses, and starts a communication by opening a session with them;
- the sending application wishes to know when any receiving applications departure (spontaneous or not) from the destination set;
- the protocol scales despite the number of receiving applications and their location in the internetwork (clustered or sparse), and harness the hierarchy in the existing topology;
- generic protocol, requiring only usual 1-1 and 1-N transmission;
- protocol takes a byte stream from the sending application and replicates it to each receiving application.

1.3 Outline of the Thesis

Chapter 2 presents essential background information on reliable multicast protocols. It addresses issues that are pertinent to reliable multicasting applications, such as degree of reliability, protocol design issues, and then analyses representative examples of reliable multicast protocols found in recent literature.

The description of the PRMP reliable multicast protocol is divided in two parts: flat and hierarchic. Chapter 3, the core of this thesis, presents the flat PRMP protocol. The chapter describes a novel sliding window mechanism for reliable multicast; error control and flow control mechanisms are developed according to this one-to-many sliding window. To prevent the scalability problems usually associated with multicast window mechanisms, PRMP provides an implosion avoidance scheme where receivers return feedback only when allowed (by a sender) to do so; the sender elicits feedback from receivers by means of polling requests. PRMP aims to deliver the potential cost-effective of reliable multicasting, and for that purpose PRMP includes a loss detection and recovery mechanism which saves network bandwidth when retransmissions are required.

A multi-threaded protocol architecture was designed for flat PRMP, and a prototype implemented. To evaluate the protocol and the architecture, simulation experiments were conducted with the prototype. Chapter 4 describes the protocol architecture, the conditions and assumptions in which experiments were run, and the simulation results obtained. It is shown that PRMP's polling mechanisms is very efficient in preventing implosion losses; its error control mechanism keeps control over the set of receivers and thus can treat receivers individually. The result is cost-effective mechanisms, leading to low network cost and hight throughput.

The scalability of the flat protocol is somewhat limited because of the window mechanism; it cannot support very large group sizes, and in large networks with long delays, it may suffer in terms of performance and network cost. A new, more scalable version of PRMP was designed to harness the inherent hierarchy of large current networks; Chapter 5 describes this extension, from the flat PRMP to the hierarchic version of PRMP, including issues of wide-area networks such as congestion control.

Chapter 6 provides the protocol architecture of hierarchic PRMP, as well as enhanced network simulations of the protocol. It compares PRMP to a sender-initiated, TCP-like reliable multicast protocol, using a network configuration with hosts, routers, and links; it also compares the flat and the hierarchic versions of PRMP, and finally evaluates the effectiveness of the mechanisms which were added to flat PRMP while extending the protocol to the hierarchic version.

Chapter 7 provides a summary of the work presented in this thesis, highlighting its contributions. It also includes some final remarks, and the thesis ends with ideas for future work.

Chapter 2

Background

The contents of this chapter are organized into three main sections. Firstly, it identifies the main *application* issues regarding reliable multicasting:

- communication model;
- reliability & timeliness issues;
- sender v. receiver reliability;

Secondly, it addresses the main *protocol* design aspects related to reliable multicasting:

- packet losses in multicast communication;
- error control: ARQ v. FEC;
- sender-initiated v. receiver-initiated schemes;
- implosion avoidance;
- group organization.

Finally, the chapter addresses related work by presenting an informal analysis of the most representative reliable multicast protocols found in the literature:

- SRM - Scaleable Reliable Multicast ([Floyd95]);

- RMTP - Reliable Multicast Transport Protocol ([Paul97], [Lin96], [Buskens97]);
- TMTP - Tree-based Multicast Transport Protocol ([Yavatkar95], [Yavatkar95b]);
- LBRM - Log-based Reliable Multicast Protocol ([Holbrook95]);
- MFTP - Multicast File Transfer Protocol ([Miller97]).

2.1 Application Issues

2.1.1 Communication model

Models for applications of multicast communication can be categorized based on two characteristics:

- one-to-many v. many-to-many;
- unidirectional v. bidirectional.

The division between one-to-many and many-to-many models depends on which participants of the multicast transmit application-level data. As the name implies, the one-to-many model is characterized by one sender which transmits data to many receivers, whereas in the many-to-many model multiple nodes can send and receive data.

Applications which require one-to-many and many-to-many have different characteristics. For example, one-to-many is typical of "information dissemination", such as software distribution, active distribution of web-pages (with *push* technology as in [Nonnemacher97]) or media broadcast, while many-to-many is typically used in applications like distributed replica management and multi-party conferencing.

One-to-many and many-to-many applications also differ in their organization (as shown later in Section 2.2.5). In one-to-many applications there exists a central role which is played by the sending application; in many-to-many applications, often data and control are distributed among the set of participants (such as *wb*, [Floyd95]).

The second aspect relevant to the application model is the *direction* of the communication. In many-to-many communication, it is possible that the "many" that send are the same "many"

that receive. In one-to-many, the sending application may expect receiving applications to return responses to a given request which is multicast; one example is a replicated service (e.g., a client requesting *dependable* services from a server which is replicated in several different sites). In both these cases, the communication between these nodes is said to be *bidirectional.*

All combinations are possible; this thesis addresses *data dissemination*, which appears to be the most common application of one-to-many, unidirectional multicast.

2.1.2 Reliability v. timely Guarantees

The degree of reliability required in a multicast communication may vary from application to application. There is an associated cost with achieving reliability, and not all applications are willing to pay for it. At the lowest end of the scale, there is no guarantee of delivery. This model of communication is often called "best effort", and corresponds to what the IP network layer provides [Stevens94]. Packets are "dropped" by the network in a silent manner, that is, neither sending nor receiving applications are informed about the packet loss. An application relying on a best-effort service either does not need reliability or implements the reliability itself, so that error control and recovery mechanisms at lower levels (say, transport-level) are not required or perhaps even desirable. Where to place mechanisms to achieve reliability is a fundamental design issue (see the "end-to-end argument" of [Saltzer84]).

A reliable multicast protocol can be added on top of the above best-effort service in order to provide lossless and ordered delivery of data to receiving applications. In such cases, the application at the sending end produces a stream of data which is to be *reproduced* at each receiving end. The protocol attempts to hide problems (see Section 2.2.2 on error control) with the transmission of such streams. In order to hide any data loss by the network, protocol mechanisms incur overhead which may appear as increased end-to-end delays (loss of throughput) and as additional bandwidth/network cost.

The above scheme guarantees that bytes that are taken by the sending end of the protocol are delivered to the receiving ends (i.e., to receiving applications). However, it cannot guarantee that the data will be consumed ("read") by the receiving application (the host or application may crash before this happens), neither can it guarantee that once consumed the receiving

application will have time to process the bytes. Only an application-level message exchange can provide this guarantee to the sending application, and this is called "end-to-end application reliability". One example of such a communication is when a client sends a *request* to a server, which sends back a *response* that works both as an acknowledgment to the receipt of a request as well as the carrier for the results expected from that request. (Note that this is bidirectional communication.) The key aspect is that the receiving application only acknowledges the data *after* it has been safely processed.

In general, any application wishes the data it transmits to be reliably delivered. Some, however, cannot afford to spend time waiting for losses to be recovered; the *timely delivery* of data is more important. Examples include the broadcast of audio and video over the Mbone. These applications have "soft real-time" requirements: they can sacrifice some of the reliability in favor of speedy delivery of data. As a "live" transmission, the usefulness of data at the receiving applications is time-bounded. For example, if a receiving application employs a "playout buffer" to display a live video stream (with a constant small delay with respect to the actual event), it needs to have the packet available at the moment its information is to be displayed. After that, the packet is of no use.

In conclusion, multicast applications may require different degrees of reliability; in general, the more reliable the service is, the higher is the overhead incurred to achieve it. Some applications require timely delivery, and thus cannot accept the overhead associated with reliability. With that in mind, applications of reliable multicast have been broadly divided ([Lin96],[Bagnall97]) into two categories: *timely-delivery* and *fully-reliable* multicast. Fully-reliable multicast applications are those where receivers cannot tolerate any losses (e.g., file transfer). Applications with timely-delivery are those where some degree of loss can be tolerated, but with small end-to-end latency for the data which is successfully delivered. Applications of fully-reliable multicast with timely-delivery (i.e., soft or hard real-time guarantees) are possible, though difficult to realize in the current networks.

This thesis focuses on fully-reliable (non-real-time) dissemination of data.

2.1.3 Sender-reliable v. Receiver-reliable

Another relevant aspect associated with the reliability regards *which end* of the application requires reliability: sending or receiving. In many reliable multicast applications, the sending application is unaware of the membership of the destination set: it cannot determine which, or how many, receiving applications are successfully receiving the data from the transmission. Making an analogy, this is the case of a TV or radio broadcast; the sending application is not interested in knowing how well the transmission succeeds. As in a TV broadcast, the sending application does not care if a segment of the receiving applications becomes "unreachable", and thus fails to receive part of the data. It is up to the receiving applications to accept such (unrecoverable) losses, and possibly "complain" using some other means. Applications like broadcast of IRTF meetings through the Mbone fit this model well.

In other uses of multicast, however, the sending application *knows* which are the receiving applications (it has a list) and wishes to keep control over the group membership, limiting access to the group and being informed of any departures. At any point in the communication (including at the end), the sending application might wish to be informed *which* receiving applications have received all data transmitted so far. In addition, the sending application may wish to know in which point of the data stream those which have not received all data stopped receiving. This model is important whenever the transmission has economic value, i.e., when it is important to bill receiving applications for the data received (possibly proportionally); this kind of agreement might be called *"pay-per-data* events" in the future.

Holbrook [Holbrook95, p.330,p.337] has classified the above two types of applications according to the reliability requirements: "receiver-reliable communication" and "sender-reliable communication". In *receiver-reliable* schemes, each receiving application defines its own reliability requirements. So, it is the receiving-side of an application which requires reliability. The model is typically used in applications which can tolerate some loss (like the broadcasting application), as the sending end of the protocol underneath may not be able to satisfy a request from a receiving end; the receiving application is informed by the receiving end of the protocol about the unrecoverable loss.

The *sender-reliable* communication, in contrast, corresponds to the traditional model where

the sending application determines the degree of reliability of the transmission, and has reliable feedback about an on-going transmission. Sender-reliable schemes are used when the sending application is interested in the termination status of the communication, or when the sending application wishes to consult or be informed about membership changes.

One of the main differentiating aspects between sender and receiver-reliable schemes is whether the sending application is aware or unaware of the group membership. In the sender-reliable model, the sending application knows the group membership, and requests the reliable multicast protocol to deliver data providing as destination *a list of receiving applications.* In contrast, in the receiver-reliable model, the sending application provides *the group identifier,* not knowing the identity of receivers behind the group.

The division between sender- and receiver-reliable is orthogonal to the degree of reliability required by the application (see Section 2.1.2); the degree of reliability is related to the amount of effort put into recovering a loss, while sender- and receiver-reliable classification is related to what happens once a loss could not be recovered (*unrecoverable* data loss).

This thesis aims at a one-to-many, fully-reliable, sender-reliable multicast protocol.

2.2 Protocol Design Issues

2.2.1 Packet loss in multicast

A critical issue for reliable multicast protocols is the manner in which packet losses occur in the network. First consider a simple multicast network model where each of the (say, N) receivers maintains an *independent, individual* channel (c_i) with the sender. Each channel c_i has a given non-nil loss rate (ε_i): the probability that a packet sent by one end reaches corrupted or does not reach at all the other end of the channel is ε_i. When a packet is multicast by the sender, a copy of the packet is sent to each of the channels; any feedback packets that result from receivers go through the same channel from which the original data packet arrived. Figure 2.1 illustrates this arrangement for $N = 6$.

Note that if a single copy of the packet is lost in one of the channels, recovery-related action will be required; for most protocols[1], it means that the sender will have to detect the

[1]this may not be true for forward-error control protocols (see Section 2.2.2).

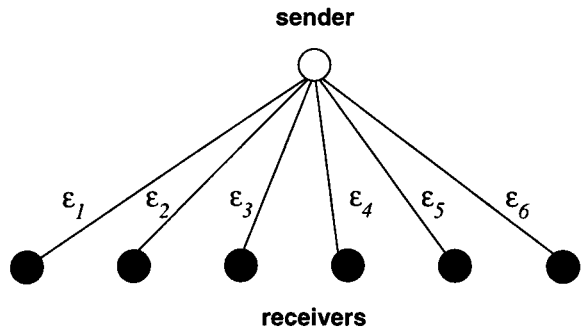


Figure 2.1: Simple network loss abstraction.

loss (e.g., through feedback from receivers) and it will have to retransmit the lost packet. The probability that at least one copy of the packet is lost increases with N. So, if N tends to ∞, the probability a given multicast transmission will require recovery tends to 1. In other words, there will be a group size sufficiently large to cause all multicast packets to require recovery by one or more receivers.

The above observation does not address the issue of *how many receivers* will require recovery of a given data packet. In this simple network model, channels are independent, and so are losses. The number of receivers experiencing a given loss will depend on the loss rates set for individual channels; for example, if two receivers are at the end of very lossy channels then the chance that both receivers miss a given packet is, compared to other receivers, higher.

When multicasting in actual networks, instead of independent channels, packets are propagated from sender to receivers through a multicast routing tree. In the Internet, anecdotal evidence suggests that most losses are caused by congestion, that is, by buffer overflow at (packet-switching) routers, not by packet corruption. Even when there is a low probability of loss in each of the nodes of the tree, or in the physical links that connect such nodes, the cumulative loss probability seen by the source at the root of the tree may be quite high ([Bhagwat94]).

Note that a packet which is lost at a given node of the tree will not arrive at *any receiver* that is downstream of the point of loss. In the worst case, a packet is dropped at the root itself (before being transmitted) and is therefore missed by *all* receivers. Hence, the higher the multicast tree is, the more overlap exists between paths, and so higher is the probability that a given loss will be *correlated* among receivers. This loss correlation is *spatial*: if a given receiver

experienced a loss, it is likely that nearby nodes (siblings and downstream nodes in the tree) will experience the same loss. Two examples of losses in a multicast transmission are shown in Figure 2.2; routers are represented as squares, and receivers as circles; the receivers affected by a loss are marked with an external dashed line. The loss on the left is higher than in the tree on the right, and affects 3 receivers; the one on the right affects a single receiver.

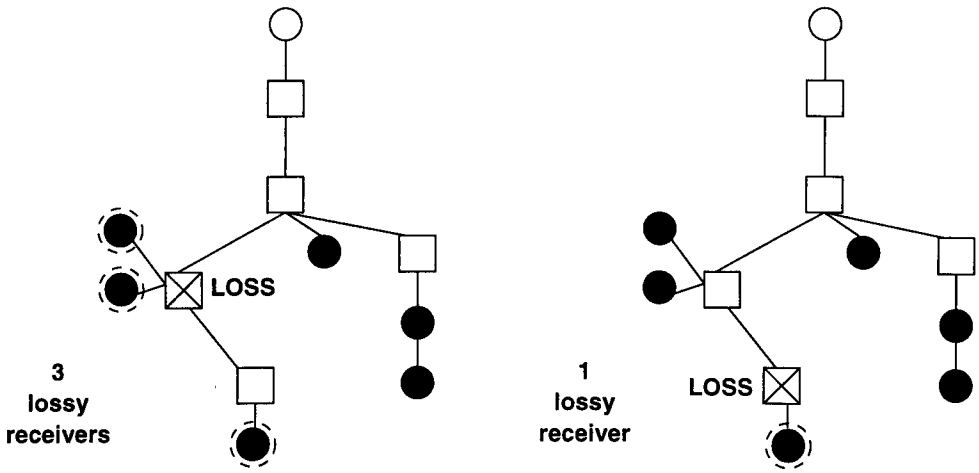


Figure 2.2: Two losses with different "degrees" of spatial correlation.

There is a second kind of loss correlation, which is directly related to network congestion: *temporal loss correlation.* When a router receives more packets than it can forward, it eventually starts dropping packets due to buffer overflow. When this situation arises, it tends to cause several losses: a congested router is likely to drop several packets until the congestion control in the protocols with flows passing through the router react and slow down. Therefore, if a given packet is missed by a receiver, the chance that one or more of the next few packets will also be missed is greater than the original loss probability.

The experimental study performed in the Mbone and presented in [Yajnick96] confirms the statements above. It reports that: (a) 47% of multicast transmissions required recovery; (b) most receivers experienced very low loss rates throughout the transmission; and (c) there were *loss bursts*: a large number of consecutive packets were lost by a receiver. However, the study also found that spatial correlation was generally small apart from losses near the root. This is only due to the topology (the Mbone) employed in the experiment, as the authors found that links connecting leaf nodes to the tree ("tail circuits") had higher loss rates than those in

the Mbone backbone. According to [Holbrook95], LAN bandwidth and resources are likely to stay cheap and plentiful, while tail circuits linking LANs (local-area networks) to WANs (wide-area networks) are likely to become more congested in the future. So, more losses are to be experienced either between the root and the backbone (all receivers miss packet) or between the backbone and a receiver (only one or two receivers miss packet). Therefore, generally, a designer of a reliable multicast protocol may expect to find strong *temporal* loss correlation and possibly also strong *spatial* loss correlation.

2.2.2 Error control (ARQ v. FEC)

Depending on line and network characteristics, transmitted data may be reordered, distorted, or deleted, and occasionally, duplicated. Transmission errors may be divided in two categories:

- *data corruption*: distortion or insertion of message contents, and
- *data sequencing problems*: deletion, duplication, and reordering of messages.

If data can be corrupted, then it is necessary to detect these errors by adding some kind of redundancy to messages, whose consistency is checked at receiver side. Assuming that a corrupted packet will be detected and discarded by the underlying layer, there are two basic methods of error control:

- *forward error control (or forward error correction*, FEC): the sender generates redundant codes from the data transmitted, and sends these codes through additional packets (typically at the end). The receiver detects the losses and may be able to reconstruct missing data using the coded data in conjunction with part of the original data which was successfully received.
- *feedback error control*: the sender detects losses through feedback sent by receivers (NACKs), or the absence of it (timeout for ACKs), and reacts by retransmitting missing data.

Most reliable multicast protocols follow the feedback error control approach; in this thesis, only the feedback-based techniques are considered. It is assumed that the detection of corrupted

packets (e.g., through cyclic redundancy checks, or CRC) may be easily executed at any layer equal to or below the error control layer in question, transforming distortion errors into deletion errors by simply discarding corrupted packets. As only feedback error control techniques are considered, deletion (loss), duplication and reordering of packets have to be detected and then recovered by means of retransmissions. This technique is known as automatic repeat request, or ARQ. There are three classes of ARQ protocols:

- *Stop-and-wait*: the sender waits on a timer for an ACK from the receiver for every message transmitted. Not only the ACK confirms the receipt of the message by the receiver, but also entitles the sender to transmit another message (thus, error control and flow control become mixed). If the timer expires, the sender assumes the packet or the ACK was lost, and retransmits the packet and resets the timer. To calculate the timeout, the sender uses the average *round-trip time* (RTT) between sender and receiver. A 1-bit sequence number in the packet and ACK is used to uniquely identify packets. ([Holzmann91, p.83], [Peterson96, p.111])

- *Go-back-N:* the sender can transmit several packets in a sequence, keeping outstanding packets in a sliding window. The receiver too is modified so that it can store (in a window) packets received out-of-order. A receiver returns an ACK which acknowledges the receipt of the highest-numbered packet received in order and implicitly all previous ones. When the sender detects a loss, it "backtracks" to the lost packet, and retransmits this packet and all subsequent ones that had been transmitted (potentially an entire window) ([Peterson96, p.155]).

- *Selective retransmission*: the receiver acknowledges *exactly* those packets which it has received, and the sender *selectively retransmits* only those packets whose ACK has not arrived, or which have been negatively acknowledged by the receiver.

It is possible to develop *hybrid reliable multicast schemes* by mixing ARQ and FEC techniques. Examples are the schemes in [Nonnemacher97b] and Protocol *D2* in [Nonnemacher97c, p.974]. These FEC schemes increase the scalability of reliable multicast protocols by reducing the overall number of feedback packets in a transmission. More precisely, by allowing receivers

to reconstruct data from packets which have not been lost, less NACK packets are required. By reducing the volume of NACK packets, it becomes less likely that "NACK-implosion" will occur. This phenomenon occurs when the same loss is experienced by many receivers (e.g., when a packet is dropped near the root of a multicast propagation tree): a large number of receivers send NACK packets to the sender, causing NACK-implosion. However, FEC schemes have their own limitations. One is the processing burden involved in coding and decoding data. Another relates to *loss correlation*: if losses occur in long bursts (as discussed in Section 2.2.1), the receiver may be unable to reconstruct the lost data. A third limitation is that the feedback packets in ARQ protocols may serve other functions beyond error control, like flow and congestion control. With regards to flow control, the feedback may help the packet buffer management (through a sliding window), and prevent any receiver being overrun by the sender. Feedback packets (or their unexpected absence) may also be required in order to identify potential congested routers and act conversely by reducing the load injected into the network. Nevertheless, solutions are being investigated, such as the use of a layered scheme for congestion control in FEC-based protocols [Vicisano98].

2.2.3 Sender-initiated v. Receiver-initiated

[Pingali94] has classified protocols according to the way they achieve reliable delivery. The classification divides protocols in either "sender-initiated" or "receiver-initiated", and it closely resembles the one between sender and receiver-reliable presented in Section 2.1.3. The difference between the two classifications is that the latter refers to *what kind* of reliability guarantees are offered to the application, whereas the former refers to *how* such reliability requirements are implemented. Both classifications are applicable to one-to-one, one-to-many and many-to-many.

Traditional reliable protocols such as TCP correspond to the *sender-initiated* approach. The responsibility of ensuring reliable delivery lies with the sender, which maintains state information regarding all receivers that it is multicasting data to. This state includes the group membership (who are the receivers) and which of them have acknowledged which data. According to [Pingali94], in sender-initiated protocols each packet received is positively acknowledged

with an ACK packet, transmitted from a receiver to the sender, and losses are detected (only) at the sender through the absence of ACKs (triggering a timeout).

In contrast, *receiver-initiated* schemes shift most of the responsibility for reliable data delivery to the receivers. The sender transmits packets to a group address whose membership is unknown and may dynamically change. The amount of status the sender maintains is thus independent of group size. Each receiver is responsible for detecting losses and informing the sender via NACKs when it requires a retransmission of a packet. To allow receivers to detect losses, the sender includes a sequence number in the packets (re)transmitted; receivers look for sequence gaps in the packets received. Though sometimes it may indicate packet reordering, a gap is a good indicative of a packet loss. When a gap occurs, the receiver sends a retransmission request (NACK) to the sender. NACKs can be transmitted either via unicast or multicast, depending on the protocol. The sender (or other receiver in case of NACK multicast) responds by taking a packet from the buffers and retransmitting it.

Because there is no positive confirmation of receipt in receiver-initiated schemes, the sender cannot guarantee which packets have been received and which will require retransmission. When packets have a given fixed useful lifetime (e.g., multimedia broadcast), sender and receivers can automatically discard packets based on real-time; *buffer management* in this case is simple and timer-based. Otherwise, the sender is unable to tell when a stored packet can be safely released from the buffers.

In order to provide *full reliability* in receiver-initiated schemes, the sender must be prepared to retransmit *any* of the packets which have been already sent, for the entire transmission and for an arbitrarily long time after the last packet has been sent. To be able to retransmit any packet at any time, the sender needs an "infinite buffer", which can be accomplished through an external caching mechanism (such as in RMTP [Paul97]) or by "logging" all packets transmitted to a disk (such as LBRM [Holbrook95]). Use of disk or external cache for infinite buffering has negative impact in terms of performance, protocol complexity/generality, and scalability.

An alternative to the infinite buffer approach is the use of *probability*: after transmitting a set of packets (e.g., a window), the sender waits for an *arbitrarily long time* in order to allow (with high probability) all receivers to negatively acknowledge any of the packets transmitted.

The sender then retransmits any of the reportedly missed packets. The time for which the sender remains idle must be equal to or larger than the highest current RTT estimate between sender and all receivers. The sender has to be *very conservative* in choosing this waiting time, since it is unable to estimate what is the maximum RTT among its receivers, as it does not know how many or which are its receivers.

Another consequence of not knowing the receiver group membership is that the sender cannot be sure of how much data receivers are receiving. Consequently, receiver-initiated schemes are limited to receiver-reliability (see Section 2.1.3) only. The sender-initiated approach can provide higher degrees of reliability (than receiver-initiated) without having to resort to infinite buffers. It has also the potential to achieve higher throughput, since the sender has to wait for feedback only from a known set of receivers (in the receiver-initiated the sender must be *conservative* in regards to group size and RTTs between itself and receivers). On the other hand, the sender-initiated approach scales poorly, due to the following factors:

- amount of state kept by the sender: dependent on the group size;
- volume of ACK packets returned to the sender, causing feedback implosion.

To overcome these limitations, a *hybrid* scheme can be designed by adding to the sender-initiated approach characteristics of the receiver-initiated one. There are two major modifications required: the first one is to include negative acknowledgments, allowing receivers to detect losses and report such losses to the sender. This may reduce latency significantly, since potentially the sender is able to detect a loss and retransmit a packet much sooner than by timeout alone. The second change regards scalability: to avoid the feedback implosion, the volume of feedback packets returned by receivers (now both ACKs and NACKs) must be reduced, and preferably their arrival should be spread in time. The resulting hybrid schemes differ in how frequently feedback packets are returned by receivers, and how much status each feedback packet contains. The next section describes several of such hybrid feedback schemes, which are designed to reduce the amount of feedback packets and reduce the risk of implosion losses.

2.2.4 Implosion avoidance

Different strategies have been used in order to overcome the implosion problem. They can be roughly divided into four categories (which can be used together):

- *tree-based*
- *period-based*
- *delay-based*
- *polling-based*

In the *tree-based* schemes, receivers are organized according to a tree structure. The amount of feedback sent to the source is reduced because receivers send feedback to their parent node only. The actual rate of feedback packets arriving at a parent node depends on three factors: (a) the degree of the node in the tree, i.e., how many receivers are sending feedback to the parent; (b) the rate in which data arrives at these (child) receivers; and (c) the kind of protocol used in the interaction between sender and receivers (e.g., sender- or receiver-initiated, etc.). Examples of schemes which fit this tree-based implosion avoidance include [Paul97], [Yavatkar95], and [Hofmann96].

In the *period-based* scheme receivers send fewer feedback packets, but each feeback has more content (and is thus larger). The scheme appears in two forms: *block acknowledgment* or *periodic acknowledgment.* In the former case, the sender divides the transmission in blocks, and at the end of each block receivers send a "block ACK" containing several positive and negative ACKs referring to all packets in the block (as in [Bhagwat94, Section 2.1] and [Miller97]). In the latter case, each receiver *periodically* sends (with fixed time period) a feedback packet containing a bit vector identifying which packets require retransmission. In both [Paul97] and [Yavatkar95], a receiver generates a feedback packet to its parent every "T_{ack}".

The use of probabilistic timers is the main characteristic of the *delay-based* schemes. Feedback packets are multicast to the group, and so are the resulting retransmissions. When a receiver is given a NACK (for a data packet it possesses), it may multicast a retransmission. When a given loss is experienced by more than one receiver, it is possible that multiple receivers

multicast NACK packets. Delays are part of a feedback suppressing mechanism which reduces the number of redundant NACKs and retransmissions, preventing implosion (and congestion). Examples are [Floyd95] and [Grossglauser96].

The last category is the *polling-based* implosion avoidance. It can be probabilistic or deterministic. One example of the former case is [Turletti94]: to avoid congestion and overrunning of receivers, the sender requests (through polling) feedback packets from a random *subset* of receivers, and uses such feedback in order to adjust its sending rate; asking all receivers to return such feedback would cause implosion. In the second case, as in [Hughes94], the sender achieves reliable delivery and flow control with a sender-initiated ARQ scheme; the sender uses polling to select a subset of receivers to send an ACK packet, so that the volume of ACK packets is sender-controlled and does not cause implosion.

2.2.5 Organization

Most current reliable multicast protocols use one IP multicast group to propagate packets from the sender to receivers[2]. Packets flow through a *multicast routing tree* which is built by the network according to packet routing (e.g., the Distance-Vector Multicast Routing Protocol, DVMRP) mechanics and a membership protocol (e.g., Internet Group Management Protocol, IGMP).

Most protocols deliver packets to receivers using the above scheme, they however differ in how feedback is propagated from receivers to the sender; there are three general organizations (see Figure 2.3):

- *flat or centralized model;*
- *hierarchic or tree-based;*
- *symmetrically distributed.*

This organization permeates the protocol design, affecting error control, implosion control, flow control, and congestion control.

[2]although there are approaches involving multiple groups, most notably *destination-set splitting* ([Ammar92], [Cheung95]) and *layering* ([McCanne96]).

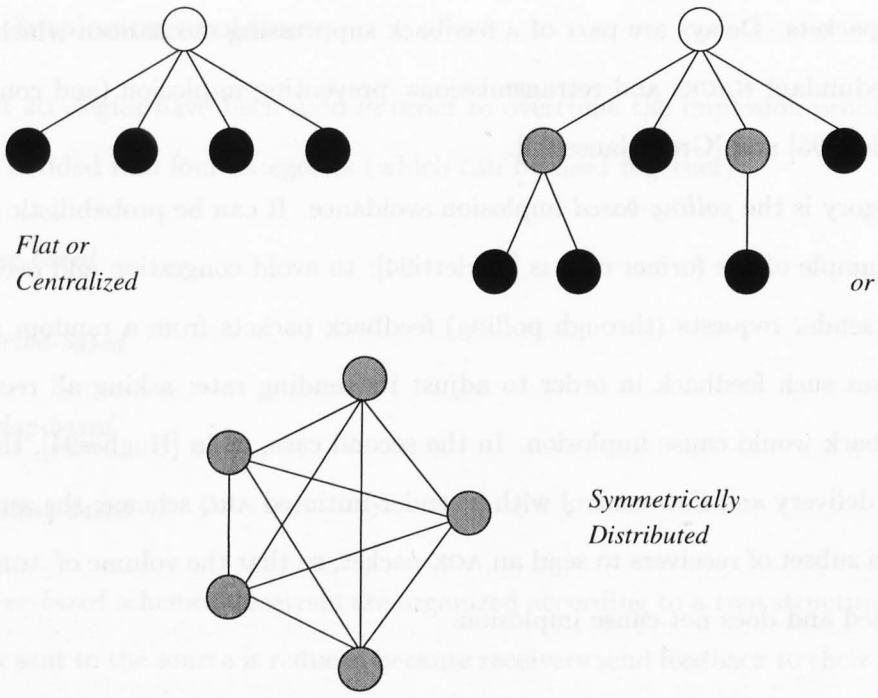


Figure 2.3: Organization of reliable multicast protocols.

In the simplest organization, the *flat or centralized model*, each receiver *directly interacts* with the sender. That is, receivers transmit feedback to the sender, which examines the feedback and carries out recovery by retransmission. One potential benefit of such a model is that the sender has information about all receivers and is able to make "global" decisions regarding the entire group. For example, when retransmissions are required, the sender is able to judiciously decide between multiple unicast or multicast retransmissions according to the number of receivers that experienced a given loss. With the loss scenarios of Section 2.2.1 in mind, using global multicast or multiple unicasts to recover losses may have a strong impact on network cost associated with the protocol. On the other hand, this flat model presents scalability problems. As the sender handles feedback packets from all receivers, the volume of feedback packets may exceed the capacity of the host and its surrounding network, leading to implosion losses. Further, in wide-area networks, where the RTT between sender and receiver may be substantial, a feedback packet and the resulting retransmission has to propagate all the way from receivers to the sender and then back; this is likely to increase end-to-end latency.

The other two models depend on the interaction among receivers. In the *hierarchic or*

tree-based model, receivers are organized according to a *tree*, with the sender at the root. Every receiver is a child node which sends feedback to its parent node. This increases the scalability of the protocol: since the sender only receives feedback packets from its child nodes, the problem of implosion is somewhat alleviated. In the tree-based model, there are two ways of organizing the recovering of packets:

- *collated feedback with centralized retransmission*: all acknowledgments are collated through the tree like in *concast* communication ([Rajagopalan93]), and the sender does all retransmissions according to the collated ACKs;
- *hierarchical recovery*: each parent node in the tree keeps packets received in buffers and performs retransmissions requested by its children; if a parent does not have a requested packet, it asks its own parent.

Though there are difficulties associated with both schemes (which will not be addressed here), *hierarchical recovery* seems to be more common (e.g., RMTP [Paul97], TMTP [Yavatkar95], and LGC [Hofmann96]). With *collated feedback with centralized retransmission*, packets have to be NACK*ed* all the way to the source, and then retransmitted, an operation which may involve substantial delays depending on the network distance between the two. The gain in comparison to the above *flat* scheme is that feedback is collated through the tree, reducing the risk of implosion, and reducing the network cost (feedback packets go only to the parent). Hierarchic recovery allows *localized recovery* to take place: a given loss may be recovered by a nearby node (often called "representative" or "group leader"), with low latency and without affecting the rest of the group.

In the *symmetrically distributed model, any* receiver which has received the packet (or the sender) is capable of retransmitting it. All retransmission requests are multicast to the entire group, so that all receivers learn about it. One or more receivers which received the packet will retransmit, again via global multicast. This symmetrical model has high network cost: suppose one of the N receivers experiences a loss, and multicasts a NACK packet; as a result, the $N-1$ nodes that received the packet retransmit via multicast. This corresponds to $(N-1) \times N$ packets being exchanged, a very high number for a single loss.

To overcome the network cost problem, a "feedback and recovery suppression mechanism" can be used. As in SRM [Floyd95], random timer values are used to delay receivers to attempt to reduce the amount of redundant NACKs and retransmission packets. To calculate a good random suppression timer for a given loss, each receiver needs to estimate the distance between itself and the sender of the packet. If all participants can send, each of the receivers need to keep RTT estimates between itself and all others. To estimate RTT between itself and all other receivers, each receiver needs to exchange periodic messages with all other receivers. The symmetric model better suits many-to-many applications, where the overhead may be reduced because group members (frequently) exchange (data) messages.

2.3 Related Work

This section analyzes the protocols which are most relevant to the thesis research, using the taxonomy presented in Sections 2.1 and 2.2. All protocols discussed employ feedback-based error control, and apart from SRM, all aim at fully-reliable one-to-many dissemination.

2.3.1 The Full Feedback Protocol

In order to realize reliable multicast in the Internet, there are several reasons for extending the TCP protocol to allow for reliable multicasting (e.g., SCE, [Talpade95]). Firstly, although TCP is in many aspects obsolete, it is well-understood and mature network technology. Secondly, it seems appropriate to develop a multicast extension of TCP which is capable of coexisting fairly with TCP flows in the Internet. Thirdly, a TCP-like application program interface (API) for multicast may take advantage of a huge software base (libraries, applications, etc.) and knowledge base (trained people) in the development of network applications. However, as indicated by [Talpade95] and demonstrated by [Pingali94], the TCP-multicast approach does not scale well because it is subject to implosion.

[Pingali94] describes a generic sender-initiated reliable multicast protocol which is similar to TCP, and compares it with two receiver-initiated schemes. Below an implementation of such a protocol is given, and termed "Full Feedback Protocol" (or simply "FF").

FF is a fully-reliable one-to-many protocol. The source interacts with all receivers (flat

organization) and the communication is driven by a sliding window mechanism. The data to be transmitted is separated in (data) units of same size; each unit fits into a single network layer packet, and is uniquely identified by a sequence number. The sender multicasts packets to a group of receivers and expects to receive a positive acknowledgment from each of them for every packet sent. Through the sliding window, the sender keeps track of which receivers have acknowledged which packets.

To limit indefinite waiting for ACKs, the sender employs a retransmission timeout ("RTO"); when the RTO expires, if one or more receivers have *not* acknowledged the packet, the packet is retransmitted via multicast ("global retransmission"). The timeout calculation by the sender is based on the original TCP scheme and uses a smoothed estimate of the round-trip time, which is calculated as:

$$RTT_{estimate} \leftarrow RTT_{estimate} \times \alpha + (1 - \alpha) \times RTT_{measurement}$$

where $RTT_{estimate}$ is the variable containing the RTT estimate, $RTT_{measurement}$ is the new fresh measurement, and α the smoother (e.g., set to 0.8). Since the sender may be at a different distance from each receiver, the sender keeps one RTT estimate for each receiver. The highest of all RTT estimates (i.e., the worst case) among receivers is used for the proper RTO determination, as below:

$$RTO \leftarrow \max\{RTT_{estimate}\} \times \beta$$

where β is an "error factor" (e.g., set to 2). Unlike TCP, the FF protocol measures the RTT by including a timestamp in the packet sent, i.e., the current value of the clock; when the receiver transmits the ACK packet, it returns a copy of the timestamp received. When the sender receives the ACK, it simply subtracts the current time from the timestamp to determine the new $RTT_{measurement}$.

Selective retransmission is used (instead of Go-Back-N): only packets which are detected to be lost (upon RTO expiration) are retransmitted, and a new RTO for the packet is defined.

It is possible to optimize this protocol in some ways. One main improvement is to change the loss detection scheme to include negative acknowledgments (e.g., send a "NACK *seq*" when packet $seq+1$ is received for the first time without having received *seq*). This adds complexity to receiver and sender, but might prevent in some cases the latter waiting for a timeout before

retransmitting.

2.3.2 Scalable Reliable Multicast (SRM)

The Scalable Reliable Multicast Protocol (SRM [Floyd95]) is aimed at many-to-many multicast applications. Designed with the Application Level Framing (ALF) philosophy in mind, SRM is implemented *as part of the application* (as opposed to an underlying layer). The application itself provides the unique identification for packets (e.g., sequence number). As pointed out in Section 2.2.5, SRM has a *symmetric* organization, and this fact influences the design of its main mechanisms, as explained below.

Firstly, SRM employs a "decentralized error recovery". A node that detects a packet loss (via a gap in the packet sequence) multicasts a NACK for that packet, and any node that has the data packet can multicast to recover that loss. The data packets which are re-multicast can speed up recovery in nodes that experience the same loss but have not yet detected it. To avoid simultaneous multicasting of NACKs or retransmissions for a given loss, a node waits on a timer to ascertain the need to multicast a NACK/retransmission. In [Floyd95] the effective way to choose these timers is discussed for some basic network topologies.

Since *any* node that successfully received the packet may provide a retransmission, recovery in SRM has the potential to be the fastest possible among reliable multicast protocols. On the other hand, the recovery mechanism of SRM introduces delays: a receiver waits on a timer before it reports a loss to other receivers and also before multicasting the packet lost by some other receiver. The recovery time is increased by the delays applied before multicasting a NACK (a REQUEST packet) or a retransmission (a REPAIR packet). To minimize recovery time, SRM proposes an adaptive mechanism to adjust the timer values dynamically; the price is an increase in the computational complexity of protocol and the state information held by a receiver.

The error control mechanism of SRM does not require a node to maintain the group membership. However, to work effectively, the suppression mechanism depends on each node maintaining RTT estimates between itself and other nodes. So, in practice, *every* SRM node needs to know explicitly *all other sending nodes.* The fact that each node has to periodically estimate the RTT between itself and other nodes (through a packet-pair) may pose a scalability problem.

The very low recovery times SRM can potentially achieve come with a price: potentially high network cost. Even if the suppressing mechanisms achieves *perfect* random delays, and there is no loss of feedback packets, the *best* it can achieve in terms of packet exchange is 2 multicast operations (1 NACK and 1 retransmission) per recovery. Recall the multicast loss patterns discussed in Section 2.2.1: the larger the group, the higher the probability a given packet will require retransmission. So, as in [Yajnick96], 47% of data packets multicast require recovery (say, by 1 receiver), the *best* the SRM mechanism can achieve is dominated by the fact that in nearly half of the transmissions...

- ...all nodes (including the sender) will be sent a feedback packet, and will start timers as part of the *probably-to-be-suppressed* recovery mechanism;
- ...the data packet will be re-multicast, so that in almost half the transmissions all receivers but those which experienced the loss (a small number on average [Yajnick96]) will receive unwanted retransmissions.

These shortcomings are being addressed in SRM by adding hierarchy to its symmetric structure. By dividing the receiver group in local domains or regions, the mechanism can work effectively in small scale within each domain; each domain, on its turn, can use a "proxy server" to communicate with other domains ([Sharma98]).

2.3.3 Reliable Multicast Transport Protocol (RMTP)

The Reliable Multicast Transport Protocol, RMTP ([Lin96],[Paul97]), is a one-to-many, tree-based protocol which provides fully-reliable file transmission. It is an evolution from the study on hierarchic multicast protocols presented in [Paul94]. RMTP is receiver-reliable (as defined in Section 2.1.3): the protocol is designed with the IP multicast model in mind, so that the sender does not need to know the membership of the receiver group. For this and other reasons, RMTP appears to scale very well. The protocol has been shown to work in actual networks through experiments using an intercontinental network configuration (with around 10 receivers). RMTP+, a variation of the protocol which is capable of transmitting continuous streams, has been implemented and deployed by AT&T for dissemination of call-billing related information. RMTP

has been recently re-implemented by GlobalCast Communications for commercial utilization.

RMTP is organized as a *two-level logical tree*: receivers are grouped into "local regions" or domains. As illustrated in Figure 2.4, each local region has a special receiver, the "Designated

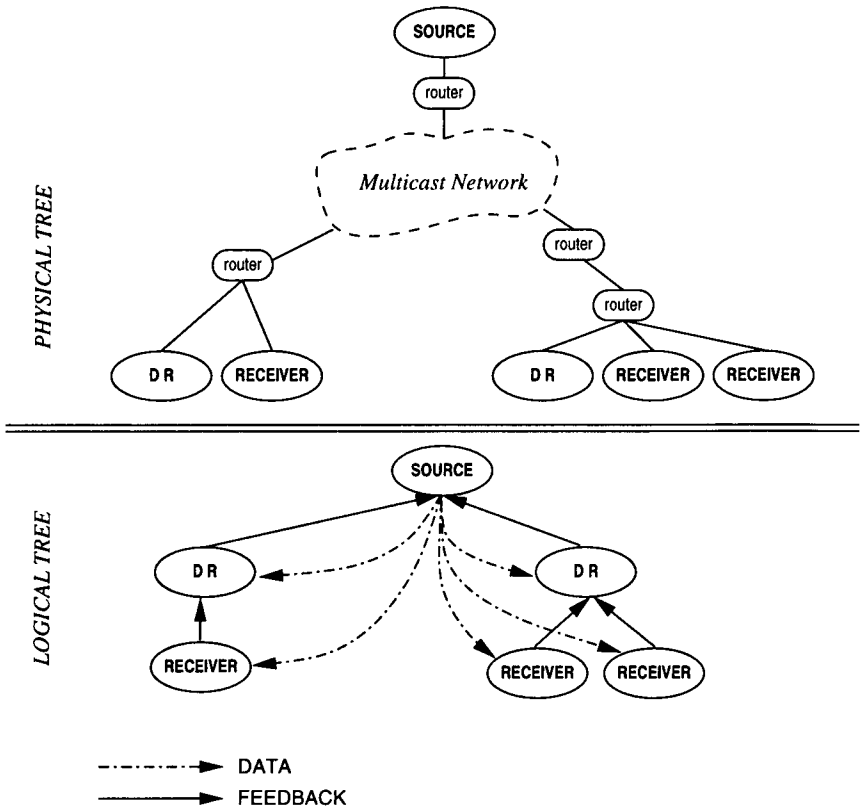


Figure 2.4: RMTP tree-based structure.

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. To reduce the volume of feedback arriving at any node, a receiver or DR sends feedback only to its parent. Additionally, a child node only returns feedback packets to its parent periodically. The logical tree is built so to explore the physical network topology; the physical agglomeration of receivers in subnetworks is "mapped" into domains of the logical tree. The logical tree is built in a distributed way: receivers choose their parent DR autonomously, and the parent node, source or DR, does not know the child nodes it parents. This decision affects the design of RMTP error and flow control mechanisms.

Though RMTP is tree-based, a DR *does not* collate status as in *concast* communication ([Rajagopalan93]). Instead, error control is applied *independently* in each level of the tree: the

feedback sent to the source by DRs regards only the reception of packets at DRs themselves, not at the source's grand-children. In other words, the source only "sees" the set of DRs, while each DR only "sees" its children. Error and flow control are based on a sliding window representing a set of fixed-size data packets. A window is comprised of a bit vector (v) and a sequence number which represents the left edge of the window (lbe, for *lower bound edge*). Each parent keeps a send window (SW) of size W_s; however, the only parent to multicast new data is the source. Each child keeps a receive window (RW) of size W_r that represents the reception of packets. A gap in RW indicates that a packet has not been received; a gap in SW indicates that one or more receivers have requested a retransmission of the packet (see below). RW slides forward according to the reception of packets (independently of consumption of data by the application), whereas SW slides forward according to the minimum left edge recorded ($\min\{lbe\}$) in feedback packets received from the children.

RMTP is timer-based. On the receiver side, each child periodically sends feedback packets to its parent at every T_{ack}. A feedback packet contains a copy of RW; gaps in the sequence represent retransmission requests (NACKs) to be served by the parent. Selective retransmission is used so that only NACK*ed* packets are retransmitted. A parent decides between multicast and multiple unicast retransmission according to the number of receivers requesting the retransmission of a given packet, in order to save network bandwidth. The timer T_{ack} is set independently by each child according to the estimated RTT between itself and its parent, plus 1/3 of the value chosen as period for retransmissions (denoted as T_{retx}, see below).

The processing of such feedback packets at a parent is event-based, triggered when feedback packets arrive. The source both transmits and retransmits data periodically, while DRs retransmit periodically (only the source transmits original data). Transmission and retransmission operations happen at every T_{send} and T_{retx} time, respectively, and independently of each other. That is, T_{send} and T_{retx} "overlap" in time. The scheme is illustrated in Figure 2.5, using the source, a T_{send} which is equal to the T_{retx}, and a T_{retx} which is offset in $\frac{T_{send}}{2}$. Apart from being idle, the sending role of a parent may be in one of three activities: sending new data (at T_{send} times), retransmitting (at T_{retx} times) or waiting/processing feedback. The workings of the protocol can be summarized as follows:

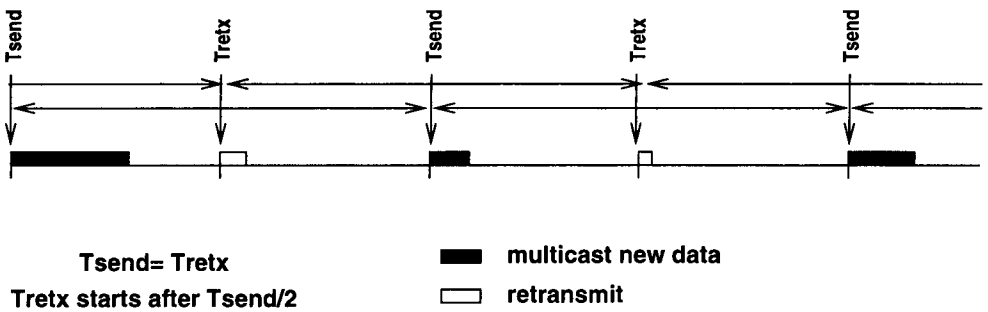


Figure 2.5: Periodic transmission of packets by the source in RMTP.

1. At every T_{send}, the source transmits a burst of data packets, up to a window size of packets. After the burst, the source does not transmit *new* packets until the next T_{send} time arrives.

2. Meanwhile, the source receives feedback packets which are periodically sent by the DRs (as DRs receive from their receivers), and accumulates all NACKs in a retransmission queue which contains one entry per packet sequence requested, each entry with a list of receivers' network addresses.

3. When T_{retx} time arrives, the retransmission queue is consumed: each entry in the queue triggers recovery for the packet referred by the entry. The source opts between multiple unicasts or a multicast retransmission according to the number of receivers which have NACKed the packet. If the queue is empty, no packet is retransmitted; with or without retransmissions, new data packets can only be transmitted at the next T_{send}.

4. When the new T_{send} comes, a parent advances *lbe* of SW to the smallest *lbe* reported since the last T_{send}. The amount of new packets that can be now multicast depends on the feedback received since the previous T_{send}: it is equal to the number of packets that the *lbe* just advanced, i.e., at worst, 0 (no new transmissions), and at best, a full window.

This transmission cycle is illustrated with an example (see Figure 2.6), with settings: $T_{send} = T_{retx} = 600$ms, and $W_s = 15$ packets. Suppose T_{send} happens at time 2,000, when the source has an available window of W_s packets (15). The source multicasts 15 packets, sequences #101-#115, in a row (①). A feedback packet ($lbe = 103, v = 011...1$) arrives at the source (②), NACK*ing* #103, at time 2,250, and an entry is added to the retransmission queue. T_{retx} comes

at 2,300, and #103 is retransmitted (③). Nothing else happens until a new T_{send} comes at 2,600, when the source advances *lbe* to #103, the minimum left edge verified since time 2,000; the source uses up the current available window, and transmits two new packets (④), #116 and #117. At 2,700, a new feedback arrives (lbe = 115,v = 00...0), acknowledging all packets up to #114 (⑤). At the next T_{retx}, at 2,900, the retransmission queue is found empty, and no packet is retransmitted (⑥). The source remains idle until the next T_{send}, at 3,200, when it advances *lbe* to #115, and transmits #118-#131 (⑦).

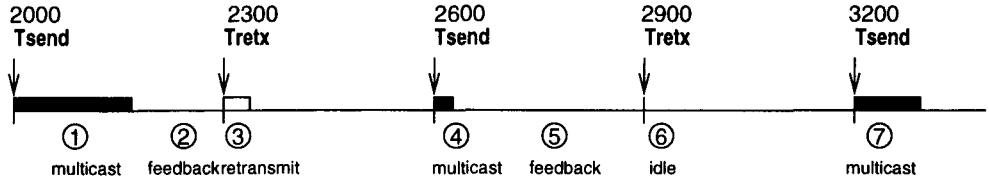


Figure 2.6: Example of transmission cycle at sender in RMTP.

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. Each parent node waits for a time equal to T_{send} for feedback to come from all receivers/DRs before sliding the window forward. The following example will demonstrate why this can be a major limiting factor:

- if the time needed to transmit a packet equals T_{tx}, then it takes $T_{tx} \times W_s$ time to send a window;
- if the RTT to the farthest child node is some RTT_{max}, *seq* is the last packet of a window and its transmission occurs at time $T_{tx} \times W_s$, then packet *seq* will reach the farthest child at time $T_{tx} \times W_s + \frac{RTT_{max}}{2}$;
- the first instance of T_{ack} *after* the arrival of the packet *seq* allows the receiver to send a feedback packet; assume that this happens with some delay d after time $T_{tx} \times W_s + \frac{RTT_{max}}{2}$;
- after being sent, the feedback requires another half-RTT and only reaches the sender at time $T_{tx} \times W_s + RTT_{max} + d$ (the time to transmit the feedback has been left out of this analysis);

- at time $T_{tx} \times W_s + RTT_{max} + d$, when the feedback arrives, the next T_{retx} should not have been triggered, or else the packets which are NACK*ed* by this feedback will be only retransmitted in the next T_{retx}.

- further, when the feedback arrives, the next T_{send} must not have been triggered; if it has, (i.e., $T_{tx} \times W_s + RTT_{max} + d > 2 \times T_{send}$), then the window was advanced without the knowledge of this feedback packet, which may have brought NACKs to packets which were left behind in the window.

In other words, if T_{send} is not sufficiently large, or RTTs grow unexpectedly larger, or if a feedback is lost, it is possible that the sender does not see a "NACK *seq*" and at T_{send} advances *lbe* beyond *seq*. If this happens, the sender will transmit the next burst of packets as if there was no such loss. The receiver which is missing *seq* has to discard part of these packets (all packets in the worst case), since there is a gap and data cannot be delivered unordered to the application. When the sender receives the next feedback from the same receiver, it learns about the loss of *seq*, $seq < lbe$. So, in RMTP it is possible that the sender receives a retransmission request for packets that have been left behind in the window, and thus have been discarded. RMTP overcomes this problem by requiring the sender and all designated receivers to *cache* all data, irrespective of the size of the stream being transmitted (this is the "infinite buffers" abstraction). When a packet to the left of the window is requested ($seq < lbe$), the copy of *seq* is fetched using a two-level caching mechanism (main memory and disk). Hence, in RMTP storing all data on disk in all nodes is required to guarantee *full reliability* (see Section 2.1.2). This is the price paid by RMTP for being scalable [Buskens97] (recall that in RMTP the sender or any parent does not know the receiver group membership).

One limitation of RMTP is that its timer-based approach depends on timers which are not *adaptive*. To work efficiently, the protocol depends on the *careful selection* of values for its timers. For example, T_{send} must be sufficiently large (as indicated above), otherwise a parent will have to frequently resort to its cache mechanism in order to serve "late" retransmission requests. Part of the cache mechanism is on disk, whose access delays are orders of magnitude larger than a main memory access, affecting performance. The values of T_{send} and T_{retx} must be provided (by the user) before the session starts. The user may not know what the best

values are, and even if the input values are correct at the start of the communication, they may become "bad" if network conditions change later (as RTTs may fluctuate).

RMTP requires an external entity, a session manager, to prepare a connection, and give all participants proper connection values. It assumes that this session manager is responsible for detecting receivers voluntarily or involuntarily leaving the multicast group and "taking necessary actions". This limitation, as well as the infinite buffer requirement, derive from the design decision of *not having a parent know the id of its children.* The source does not know who its children are and has to adopt a conservative behavior: wait an arbitrarily long time before declaring packets successfully received, and transmitting new packets. This has substantial implications on throughput and network cost. It might be straightforward to change RMTP to add the knowledge of the membership to a parent, but as RMTP is designed, it cannot take advantage of such knowledge.

2.3.4 Tree-based Multicast Transport Protocol (TMTP)

The Tree-based Multicast Transport Protocol (TMTP, [Yavatkar95], [Yavatkar95b]) has several similarities with RMTP. They are both tree-based, employing a control tree for scaleable error and flow control. However, unlike RMTP, the control tree of TMTP is made up by protocol agents, which are called "Domain Managers" (DM). Each DM controls a *domain* and may have an arbitrary number of receivers as its children. Another difference to RMTP is that TMTP employs a multi-level tree, which is built dynamically according to group members which leave and join the group during a multicast session. To build the tree, it uses an *expanding ring search* which is implemented with successive multicasts, increasing each time the value of the TTL (*time-to-live*) field of IP packets. Both RMTP and TMTP employ local recovery: feedback packets are sent to the parent, which locally retransmits packets. One important difference to RMTP is that in TMTP *all* retransmissions are multicast (though with restricted TTL scope).

The error and flow control of TMTP is applied independently in each level of the tree, as in RMTP. However, TMTP parent nodes know the membership, that is, keep track of the group of children under their control. Like a sender-initiated scheme, each parent node expects positive acknowledgments from its children, and uses timeouts to limit waiting for such positive

ACKs. Positive ACKs are sent periodically by each child according to a *common* T_{ack} (unlike RMTP, where each child uses its own T_{ack}). T_{ack} is set by the source at the beginning of the communication as the RTT between itself and the farthest receiver in the tree; it remains fixed during the transmission. The error control adds "negative acknowledgments with NACK suppression" to the periodic positive ACKs: when a receiver detects a loss[3], it starts a random delay timer and, at the end of this timer, multicasts a NACK *seq* with sufficient scope to reach its parent and siblings. If the receiver receives a NACK packet requesting the retransmission of *seq* before the timer expires, the receiver suppresses its own NACK. Upon receipt of NACK *seq*, the parent *multicasts seq* (provided that the parent itself is in possession of the packet).

The source transmits a window of data within a period of time equal to $T_{retrans}$, using a fixed (maximum) transmission rate, Tx_{rate}. The value of $T_{retrans}$ is set to be a multiple of T_{ack},

$$T_{retrans} = T_{ack} \times n$$

, with n an integer and $n \geq 2$. The window size, Ws, and hence buffer requirements, are derived according to the size of $T_{retrans}$, and the value of Tx_{rate}, as

$$Ws = T_{retrans} \times Tx_{rate}$$

packets. The source "splits" the sliding window in n *segments* of $\frac{Ws}{n}$ packets. It attempts to slide the window forward one entire segment at every T_{ack}; if it succeeds, it slides the window in $\frac{Ws}{n}$ packets. To succeed, all $\frac{Ws}{n}$ packets of the *first* segment must have been "fully acknowledged" (i.e., acknowledged by all receivers). If they are not, then the left edge of the window stays in the first segment, and retransmissions are executed until all packets of the segment have become fully acknowledged.

The scheme is illustrated in Figure 2.7, which portrays the transmission (at the source side) from the beginning of the communication. The source starts transmitting data packets with rate Tx_{rate} for 3 T_{acks} ($T_{retrans}$ duration). When the 3rd T_{ack} finishes, the source checks which packets regarding the 1st segment (transmitted in the 1st T_{ack}) have not been ACK*ed* by all receivers. In the example, some packets have not been fully ACK*ed* and are retransmitted during the 4th T_{ack}; later in the same T_{ack}, part of the retransmitted packets are NACK*ed* by one or more receivers, and re-retransmitted. Then all packets of the 1st segment which have

[3]assuming that in TMTP receivers detect losses through gaps in the packet sequence.

been retransmitted within the T_{ack} become ACK*ed*, and the window is advanced $\frac{Ws}{n}$ packets. During the 5th T_{ack}, the fourth segment is transmitted; at the end of this, the source finds out that some packets of the 2nd. segment require retransmission, and does so during the 6th T_{ack}.

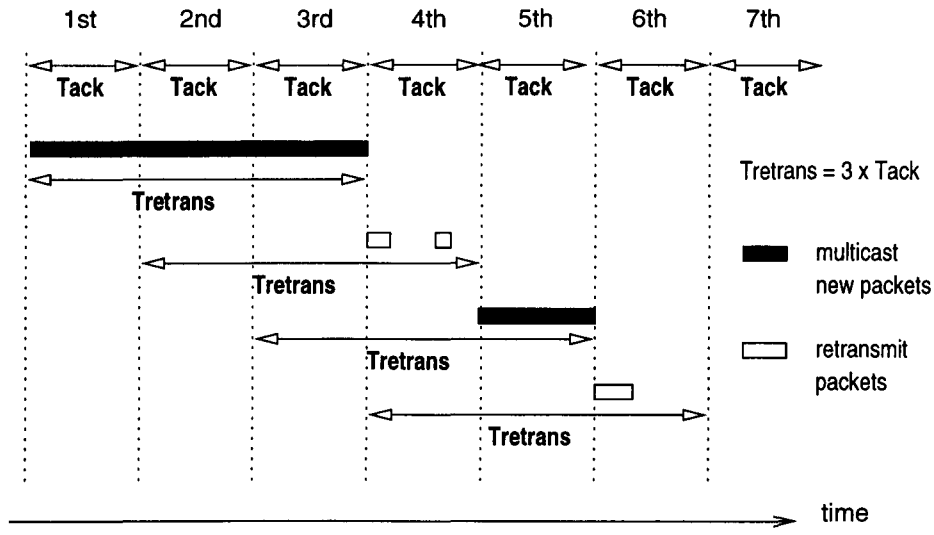


Figure 2.7: Example of transmission cycle in TMTP.

The parent or other source deals only with the first segment of the current window in order to prevent unnecessary retransmissions (arising from expired timeouts or NACKs), as well as to delay the flow control backpressure applied towards the source.

TMTP provides reliable delivery to *current* set of children attached to a DM. A parent will expect positive acknowledgments, insist on the retransmission of lost packets, and hold the sliding window until all packets have been fully acknowledged. In this sense, TMTP is fully-reliable with sender-reliability. However, it is *not* fully-reliable because the TMTP model allows a receiver which joins late to receive only *part* of the data transmitted; in contrast, in RMTP, the original transmission is delayed until such a receiver gets *all* the data transmitted via unicast. TMTP is not sender-reliable either, considering that each parent node knows the membership of its *immediate children* only (DMs and receivers). So, if a child of a DM (spontaneously) leaves the group, or a new child joins in, the source knows only if the new member is an immediate child. It is not clear (in [Yavatkar95]) what happens if a child becomes unreachable and fails to send positive acknowledgments through several timeouts (i.e., T_{acks}); if nothing is done to

remove the child from the group, the protocol will not terminate.

The problem in determining proper values for the timers appears to be another thing in common with RMTP. The values for timers in TMTP are defined at the beginning of the transmission and cannot adapt to changes in network conditions. In TMTP, the value of T_{ack} is set to be the maximum RTT between the source and any receiver; measured by the source at the beginning of the communication, at a moment that the path between the parent and children does not have yet the multicast flow (i.e., possibly an underestimation of maximum RTT). The determination of T_{ack} is important to the transmission cycle, as $T_{retrans}$ derives from this T_{ack}, and buffer size from $T_{retrans}$.

2.3.5 Log-Based Receiver-reliable Multicast (LBRM)

LBRM [Holbrook95] is a reliable-multicast protocol aimed at large-scale military distributed interactive simulations (known as DIS applications). This application is real-time and of a particular nature: the sender usually transmits packets (state updates) with low rate, but they need to be delivered to receivers with low latency. The sender does not need confirmation of delivery, and recovery latency counts on the receiver-side only.

There are two mechanisms for *loss detection* at receivers: (a) gap in packet sequence and (b) lack of packet arriving within a given time threshold (the sender "guarantees" a given transmission rate and transmits an empty packet if there is no status update to send). Losses can be immediately detected at receivers through sequence gaps when the sender keeps transmitting and just one packet in the sequence is lost. Now suppose the sender is idle, sends a packet and becomes idle again; if this packet does not reach a receiver, the receiver will not see the gap until the next packet is transmitted. The "variable heartbeat loss detection scheme" prevents these cases by limiting the interval between two packet arrivals at a receiver; when the timer expires, the receiver infers that a packet has been lost and negatively acknowledges the next packet expected in the sequence. To avoid flooding the network with empty packets, the time interval used by receivers (and by the sender when transmitting) fluctuates within a range: starting at a minimum, it doubles for each empty packet received until the maximum is reached, and is reset to the minimum whenever a non-empty packet arrives.

Reliability is achieved through a two-level hierarchy of logging servers, which store (all) data packets sent/received and retransmit packets which are NACK*ed* by receivers. Like other tree-based schemes, there is a *primary* server (with the source) and a set of *secondary* servers, which act like "representatives". Receivers in a local region request retransmissions from their local server; if the secondary server has not received the packet, it requests the packet from the primary server. The source only "sees" the primary server, discarding a packet as soon as it has been ACK*ed* by the server (which happens as soon as the packet has been logged). To name a secondary server among a set of receivers, a scoped expanding ring scheme is used (as in TMTP).

Like RMTP [Lin96], secondary servers opt between multiple unicast and multicast retransmission depending on the feedback from receivers. As in TMTP ([Yavatkar95]), retransmissions within local domains are multicast with restricted TTL scope. According to the authors, in their configuration NACK-implosion in secondary servers can be ignored because they expect a small number of receivers to be logically attached to it. For the primary server, the selection between multiple unicasts and global multicast retransmission is made on a statistical basis: the source selects a small random subset of secondary servers which are expected to send a positive acknowledgment for every packet received; depending on the number of ACKs received, the primary server uses multicast or not. The set of secondary servers expected to positively ACK is periodically re-selected, in the following manner. The source wishes to select a given number of servers; from the estimated number of secondary servers (see below how this is estimated) it computes a probability p_{ack}, and multicasts a packet with p_{ack}. Each secondary server responds with probability p_{ack}; the source waits a "long time" and counts the number of servers that responded. The primary server waits on a timeout for this set of secondary servers to send ACKs for every packet received, and when the timer for a given packet expires the server decides the kind of retransmission based on the number of ACKs (or their absence). If ACKs from all the secondary servers included in the small randomly selected group have been received, the primary assumes there were no global losses and will treat further requests (originating at other secondary servers) via unicast.

To estimate the number of secondary servers, the primary has to go through a round of

probabilistic pollings to avoid causing implosion. The sender starts multicasting with very low probability p_{ack}, and waits for responses. Each server responds with probability p_{ack}. It then repeats the cycle with increased p_{ack} until a sufficiently large number of responses have been received. The total number of servers can be estimated as $servers = \frac{responses}{p_{ack}}$.

LBRM is aimed at receiver-reliable applications which broadcast information to a large-scale group. In DIS, a sender transmits sporadically, but when it does, the data is time-sensitive. So, LBRM loss detection mechanism allows quick detection of losses by receivers even under sporadic traffic flows, but this comes at a price: it adds overhead to the network (a heartbeat packet must be sent when there is no data to transmit). Reliability with scalability is achieved with the infinite buffers abstraction, by employing a hierarchy of logging servers which store all the data packets transmitted. Hence, there is neither packet buffer management nor flow control mechanisms in LBRM (no congestion control either). Overhead in terms of latency and network cost is incurred by the protocol because the primary server does not know the identity of the secondary logging servers. For example, the source has to periodically select a subset of secondary servers to send positive acknowledgments, as well as to estimate the number of secondary servers through multiple polling rounds.

2.3.6 Multicast File Transport Protocol (MFTP)

The Multicast File Transport Protocol, or MFTP ([Miller97]), is designed for reliable dissemination of files. The unique aspect of MFTP is the way it organizes a transmission: the file to be transfered is sent through multiple "passes", as explained below.

Before transmitting, the file is logically divided in "blocks", and blocks in "data transfer units" (DTUs). A feedback packet (a "response") 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. This request allows receivers to return a response requesting the retransmission of packets within the identified block; a receiver *only* sends a response for the block if there were losses. The sender does not wait for 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 been received during the first pass, the sender starts a second pass in which it re-multicasts all packets which have been NACK*ed* by one or more receivers. For each block, the sender checks if there were losses reported; if so, it retransmits all NACK*ed* 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 request has 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 (i.e., 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 a 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 a message varies according to the group model being employed, see [Miller97] for details).

The sender transmits data at a user-defined, fixed rate. The block size is determined by the protocol (which can be overridden) so that the amount of DTUs which can be NACK*ed* through a single response packet is maximized (recall that a response refers to a single block of DTUs).

MFTP does not employ 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 routers through which the flow is passing has been or becomes congested, a large amount of packets may be dropped. Even if receivers report losses to the sender, it keeps transmitting at the same pace.

As stated in [Miller97], MFTP relies on the fact that a file is being transmitted, not a stream. The error control mechanism of MFTP requires that all I/O devices involved, that is, the device *from* which the file is read, as well as the devices *to* which the received file is written, allow random access. The multiple pass transmission is equivalent to the "infinite buffer" abstraction used by RMTP and some other reliable multicast protocols. Note that random access may not

be available in many computer systems, for example when tapes are used.

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.* There is no retransmission of request or response. If compared with RMTP, in the latter 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. To achieve full reliability, MFTP requires the "closed group model" to be used, in which all receivers send positive acknowledgments together at the beginning and at the end of the transmission (limiting its scalability as the feedback packets implode the sender).

2.4 Conclusions

Scalability problems arise when the sender maintains protocol state which is proportional to the number of receivers in the destination set. To keep the protocol state updated, the sender has to exchange messages with *all* receivers; if *all* is "large", the sender may be swamped by feedback packets (ACKs, NACKs, responses, and the like) returned by *all* receivers. If the sender is able to sustain multicasting to up to *"x"* receivers, packet losses will occur proportionally to the excess "*all-x*" receivers (implosion losses).

Table 2.1 summarizes the characteristics of the protocols presented. To increase their scalability, protocols need to prevent implosion losses; most protocols in the literature attempt to avoid implosion by *making the sender independent of receivers.* They are designed so that the sender does not require to know the group membership, and exploit the receiver-oriented nature of IP multicast, leaving all membership operations to be (transparently) carried out the by the underlying network layer. The resulting protocols (receiver-initiated) are radically different from the traditional, sender-initiated protocols; particularly, reliable delivery is achieved by making the sender "passive" and allowing receivers to "command" the loss detection and recovery process. Examples of these receiver-initiated protocols include LBRM ([Holbrook95]), SRM ([Floyd95]), and RMTP ([Lin96]).

However, the lack of knowledge about receivers in receiver-initiated schemes has negative

Protocol	Model	Reliability	Reliable at	Organization	Membership?	Requires
SRM	N-N	unordered	receiver	symmetric	only for RTT estim.	periodic session messages
RMTP	1-N	fully-reliable	receiver	2-level tree	no	inf. buffer (caching), net. support
TMTP	1-N	partial data, fully-reliable	receiver	multi-level tree	no	TTL-scoped ring search
LBRM	1-N	soft real-time	receiver	2-level tree	no	infinite buffer (disk log)
MFTP	1-N	fully-reliable	sender	flat	both possible	inf. buffer (multiple step transmission)

Table 2.1: Main attributes of related reliable multicast protocols

implications with respect to error control (cannot guarantee fully reliable delivery), flow control (cannot prevent overrunning slow receivers), implosion avoidance (cannot prevent NACK-implosion), etc. As there is no confirmation from receivers, protocols may need to rely on probabilities and heuristics (such as maximum delay after transmitting the last data packet). They depend on expert user-input (to work properly), lack generality, and have to employ conservative values (e.g., assume large RTTs, large number of receivers, etc). Therefore, receiver-initiated protocols avoid the problem of having the excess of *"all-x"* receivers which would overwhelm the sender, but the price is paid in terms of reliability, throughput, and network cost.

The next chapter describes a protocol which follows an *alternative approach* to circumvent the scalability problem posed by implosion: the sender *controls* the amount of feedback generated by receivers, undergoing multicast communication as if there were "x" receivers only. The sender retains the membership and avoids implosion, thus achieving scalability, through a *polling feedback mechanism.* This protocol, flat PRMP, is aimed at the efficient fully-reliable dissemination of data from one sender to a static set of receivers.

Chapter 3

Flat PRMP: Polling Feedback Protocol

This chapter presents the flat version of the PRMP protocol. It starts with an overview of the protocol, in Section 3.1, followed by the description of the sliding window mechanism in Section 3.2. Both error control and flow control are implemented through the sliding window mechanism. Implosion avoidance is addressed with the poll planning mechanism in Section 3.4. Flow control is detailed in Section 3.3, while error control is divided in two parts, in Sections 3.5 for loss of control packets, and 3.6 for loss of data packets.

3.1 Overview

The PRMP protocol solves the problem of a sender having to reliably transmit an arbitrary amount of bytes to a destination set containing GS receivers (the destination set is denoted as $\{*\}$). Data is provided by the sending application as a byte stream through `write()` operations. PRMP aims to deliver a copy of the byte stream "as provided" to each one of the receiving applications. Each receiving application requests data to the protocol through `read()` operations and, when such data becomes available, "consumes" it. To allow transmission of the byte stream using a sequence of limited-size packets, the data is separated into *fixed-size data*

units. Each data unit is uniquely identified by a sequence number seq[1],[2]. The transmission scheme is depicted in Figure 3.1. The protocol does not require in advance the knowledge of the total amount of bytes to be transmitted; however, for illustration purposes, in this thesis it is assumed that there are DP units of data (thus seq will range from 1 to DP).

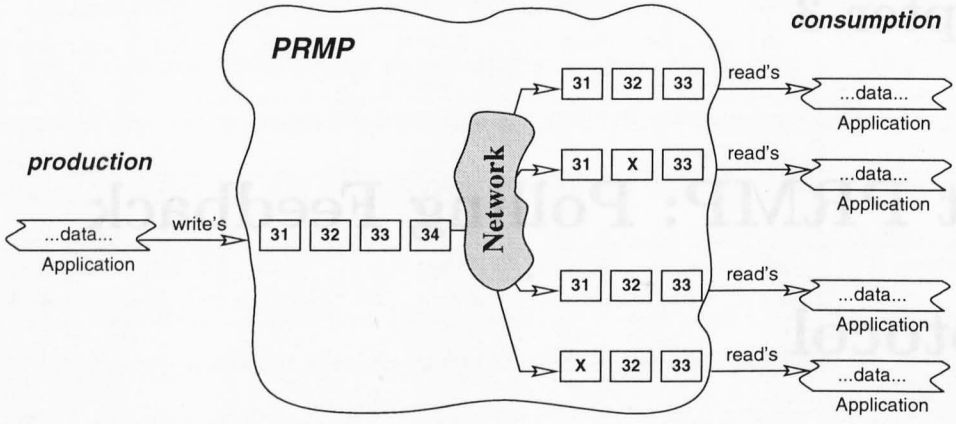


Figure 3.1: Schematic illustration of transmission process.

Failures during transmission result in packets being lost or corrupted and discarded by the network. PRMP employs feedback error control (see Section 2.2.2): the sender detects losses through feedback packets (or lack of them) sent by receivers. After a loss is detected by the sender, it is recovered by retransmission. For that purpose the sender keeps a copy of each transmitted data packet in its buffer until an acknowledgment for that packet is obtained from all GS receivers (i.e., packet becomes fully acknowledged).

The core of the PRMP is a novel one-to-many, polling-feedback sliding-window mechanism. Each receiver maintains one "receiving window" (rw) of length L data units (or "packets", for simplicity). A receiver records packet reception status in its rw. The sender keeps an "aggregate sending window" (sw), which is comprised of GS sending windows (sw_i), one per receiver. This aggregated window permeates PRMP design, and is present in mechanisms for error control, flow control, and congestion control.

All sending windows have the same length as the receiving windows, i.e., L packets. The

[1]it is assumed that seq is large enough to avoid problems that arise when sequence numbers are wrapped around and reused.

[2]in the text, seq is used interchangeably to denote the "sequence number seq" and "the packet whose sequence number is seq".

value of L is negotiated at connection time[3], and remains fixed during the transmission. The sender transmits data packets to receivers; receivers transmit feedback packets to the sender. Feedback is used to update the sending window, which "slides forward" with the progress of the transmission. The scheme is illustrated in Figure 3.2, for $GS = 4$. The sw encompasses the 4 sw_i's; data packets are multicast; when received, they lead to the update of rw_i. Receivers transmit feedback to the sender, and the sender updates the sw_i (and thus also sw) corresponding to the receiver which sent the feedback.

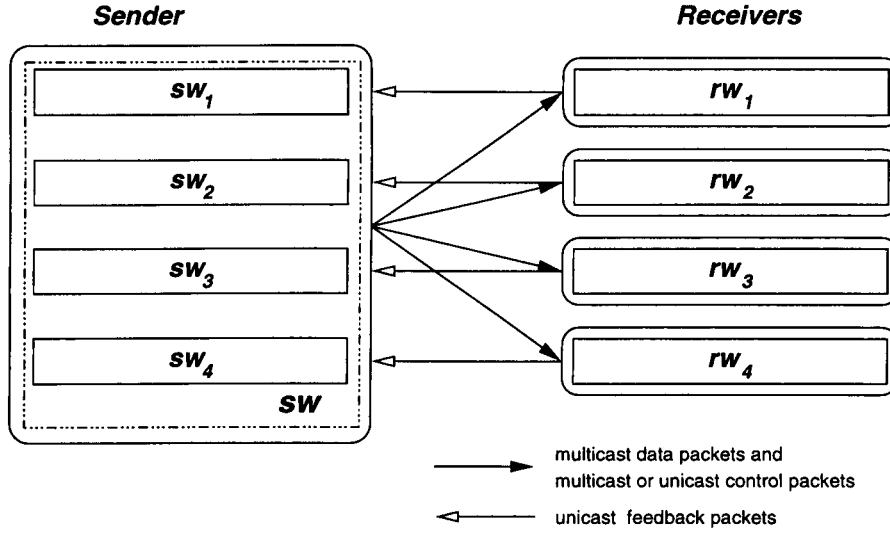


Figure 3.2: Scheme with sending and receiving windows.

To avoid implosion, it is not feasible to allow each receiver to return a feedback packet to (positively) acknowledge every received data packet, as in typical sender-initiated window-based protocols. In PRMP, a *polling mechanism* is used to reduce the amount of feedback to desired levels: a receiver indicates the receipt or non-receipt of packets only upon being explicitly requested to do so. The sender *plans* the polling of receivers, and when the plan is "due" it transmits a "polling request" to elicit feedback from one or more receivers. A polling request nominates a subset of receivers to send feedback and may be transmitted in a control packet or piggybacked onto a data packet. Upon being nominated, a receiver unicasts a response packet which positively acknowledges the packets it has received so far and negatively acknowledges the packets which appear to be missing. As feedback packets are sent only periodically, each

[3]connection set-up and tear-down phases precede and succeed the data transfer phase, respectively.

one will contain more status, i.e., several acknowledgments (like the *block* ACK discussed in Section 2.2.4). A poll response sent by a given receiver R_i will contain (as status) a copy of its rw_i.

Table 3.1 summarizes the four packet types that can be exchanged between sender and receivers.

Type	Originates at...	Contains...
DATA	sender	only data
POLL	sender	a polling request to one or more receivers
DATAPOLL	sender	both data and a polling request
RESP	receiver	feedback status from receiver

Table 3.1: List of packet types.

Of the packets transmitted by the sender, those which contain data (i.e., types DATA and DATAPOLL) carry a sequence number "*seq*" which uniquely identifies the *data unit* in the packet. Packets which carry a polling request (i.e., types POLL and DATAPOLL) contain a copy of the highest sequence number "sent" so far, denoted as "*hs*"; *hs* is employed by the error control mechanism in order to allow receivers to detect packet losses (as explained in the next sections). A DATAPOLL packet will contain both *seq* and *hs* values. There is no sequencing number which uniquely identifies a packet transmission.

As pointed out in Section 2.2.1, a single packet loss may be experienced by many receivers (specially if the loss occurred near the root of the multicast tree used to propagate the packet). The retransmission of this packet via multicast would be beneficial since a single multicast operation may target all receivers missing the packet. However, it is possible that in most original[4] multicast transmissions the number of receivers which will need recovery is small; in this case, global, multicast retransmission is prohibitive and should be avoided, so as to prevent flooding the entire network with undesired retransmissions. Selective unicast retransmissions, in this case, can *isolate* the loss from the rest of the network. *Cost-effective recovery* is achieved by PRMP by collecting retransmission requests in order to judiciously decide between multiple unicasts and multicast retransmission depending on the percentage of receivers experiencing the same loss. The mechanism which accomplishes this is described in Section 3.6.

[4] "original transmission" is used in the text to stress the difference between transmissions and retransmissions.

3.2 Sliding Windows Mechanism

This section introduces PRMP's one-to-many sliding window mechanism for reliable multicasting. The window mechanism cannot be fully described, however, until related issues of error and flow control have been addressed in Sections 3.3 and 3.6, which complete the definition of the window mechanism.

3.2.1 The receiving window

The set of GS receivers is represented as $\{R_1, R_2, R_3, \ldots, R_{GS}\}$ or $\{*\}$. Each receiver R_i maintains a receiving window rw_i, whose length is L packets. A rw_i is characterized by the following attributes:

v	a *bit vector*;
le	a *left edge*;
ned	the sequence number of the *next expected data packet*;
hr	the *highest sequence number recorded* from the sender (contains the highest *seq* or *hs* value in data or polling packets received so far).

The *right edge* (re) is an additional, abstract attribute which is derived from the length and the left edge ($re \leftarrow le + L - 1$). The window scheme is illustrated in Figure 3.3. The attributes of rw are explained below.

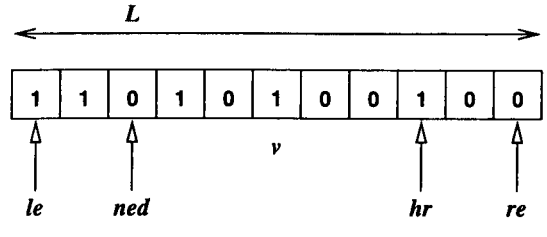


Figure 3.3: Schematic view of a receiving window rw_i.

The window rw_i represents the occupation of buffers by packets in R_i; thus, it also represents the reception of packets by R_i and the delivery of the data that the packets contain to the receiving application. The bitvector v is a vector of size L whose index ranges from 1 to DP:

$v[seq]$, $1 \leq seq \leq DP$. The left and right edges of rw_i advance in synchrony (as $re \leftarrow le+L-1$) to indicate the set of L packets to which there exists a buffer space available in R_i, as follows. All packets seq which "precede" the left edge (i.e., $seq < le$) have been received by R_i, delivered to (i.e., consumed by) the receiving application, and removed from R_i's buffers. All packets of sequence seq which "succeed" the right edge (i.e., $seq > re$) do not have buffers allocated and therefore cannot be received by R_i. The remaining packets (that is, any seq such that $le \leq seq \leq re$) have a buffer allocated and are directly represented in the window (by v); for each of these packets, $v[seq] = 1$ if the packet seq is stored in R_i's buffers and 0 otherwise.

The next in-sequence packet *expected* by R_i is indicated by ned. As ned always indicates a packet yet to be received, it never points to an occupied buffer; therefore, always $rw[ned] = 0$. If the ned indicates a packet which cannot be received, i.e., beyond the right edge ($ned = re+1$), then all buffers in R_i are being occupied. In this case, the receiving window is said to be "full of 1s" ($\forall seq,\ le \leq seq \leq re : v[seq] = 1$).

The attribute hr keeps the highest sequence number recorded so far; sequence numbers considered are DATA(POLL).seq and (DATA)POLL.hs. When a DATA packet arrives, the receiver updates hr if DATA.seq is greater than the current hr: if DATA.$seq > hr$ then $hr \leftarrow$DATA.seq. When a packet of type POLL or DATAPOLL arrives, the receivers record a new hr using hs instead: if (DATA)POLL. $hs > hr$ then $hr \leftarrow$(DATA)POLL.hs; note that DATAPOLL packets contain both seq and hs, and $hs \geq seq$[5].

The next packet to be taken from the buffers and consumed by the receiving application is the one with sequence number le. If the left edge points to an available packet ($v[le] = 1$), the application is busy doing something else (such as processing packet $le-1$) and has not yet been able to consume le. All in-sequence packets which have been received but not yet consumed (i.e., any seq such that $le \leq seq < ned$) are said to be "consumable". If, however, $v[le] = 0$, then seq has not been received and thus connot be consumed. Further, all succeeding packets with sequence seq that were received (i.e., $seq > le \wedge v[seq] = 1$) cannot be consumed either; this is because packets are consumed in sequence by the application (if seq_1 and seq_2 are any two packets, then seq_2 cannot be consumed before seq_1 if $seq_1 < seq_2$). The packets which

[5] DATAPOLL.hs >DATAPOLL.seq when a packet seq, with $seq < hs$ is retransmitted.

have been received but cannot be consumed are said to "unconsumable". A consumable packet of sequence *seq* will be eventually consumed by the application, and as a result, rw_i will slide forward one position (that is, $le = seq$; $le \leftarrow le + 1$; then $le = seq + 1$).

If the left edge indicates a *sequence number* yet to be recorded (i.e., $le = hr + 1$), all buffers at R_i are empty and there is no packet to be consumed. In this case, the receiver is idle. Otherwise, if the left edge corresponds to a sequence number which has been recorded (i.e., $le \leq hr$), there may be up to $hr + 1 - le$ buffers which are occupied. Note that it is possible that all buffers are empty even if $le \leq hr$; this is so because $v[hr] = 0$ is possible, if for example, a receiver misses a packet DATA *seq* and then receives a POLL packet with POLL.$hs = seq$. This case is discussed later in Section 3.6.1 (case depicted by Figure 3.11).

Figure 3.4 uses the same window in Figure 3.3 to illustrate the relation between the rw attributes and the buffer occupation as well as consumption of packets.

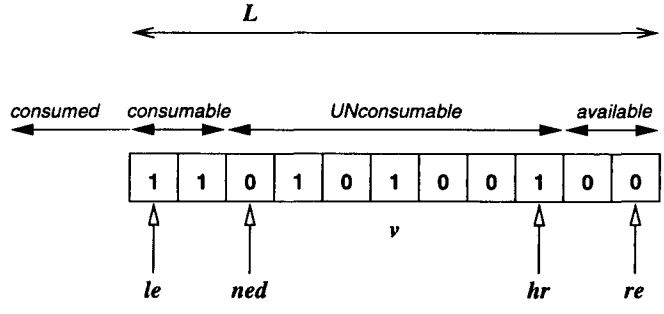


Figure 3.4: Schematic view of rw and consumption of packets.

Besides the fact that $L = 11$, the following observations can be made based on the window in Figure 3.4:

- there are 2 "consumable" packets (le, $le+1$) and 3 "unconsumable" ones ($ned+1$, $ned+3$, hr);
- there are 4 missing packets (besides ned, they are $ned + 2$, $ned + 4$, $ned + 5$);
- there are 6 empty buffers, but only 2 are free and available to be used by new data packets ($re - 1$, re).

3.2.2 The sending window

The sender maintains a set of GS sending windows $\{sw_1, sw_2, \ldots, sw_{GS}\}$ containing status about the receivers. The sending window sw_i contains the (latest) status about R_i. Each sw_i has the following attributes in common with rw_i:

v a bit vector;

le the *left edge*;

hr the *highest* sequence number *recorded* by the receiver.

The "next expected data" (*ned*) appears in the sending window as the "next expected acknowledgment" (*nea*). Each of these fields in sw_i represent the knowledge of the sender about the corresponding fields in rw_i; a poll response from a given R_i brings in RESP.rw a copy of rw_i, which is used to update sw_i attributes (note that RESP.rw works as a "cumulative acknowledgment", and positively or negatively acknowledges all packets up to the point of transmission of the polling request which generated the RESP). For example, $sw_i.le$ is the highest le received in a RESP.$rw.le$ from R_i. Therefore, sw_i indicates which packets the sender knows that R_i has received and/or consumed. All entries of the bitvector $sw_i.v$ are initially set to 0, and changed to 1 whenever the sender obtains from R_i confirmation of receipt of the corresponding data.

As mentioned earlier, the set of sw_i's is abstracted as the aggregate window, sw. sw inherits all the attributes of sw_i apart from v, and adds hs, the (already-mentioned) highest sequence number multicast so far. In other words, $sw.hs$ records which was the last sequence number used in an "original" transmission.

As an abstraction, apart from hs, all sw attributes do not have associated state[6]. Instead, the attributes of sw are *computed* from the set of sw_i's upon demand. There are different ways of "aggregating" a given attribute "a" of sw_i's into the counterpart "$sw.a$", such as using $\min\{sw_i.a \mid R_i \in \{*\}\}$, $\max\{sw_i.a \mid R_i \in \{*\}\}$ or $\text{avg}\{sw_i.a \mid R_i \in \{*\}\}$. The aggregating function employed varies with the semantics of the attribute.

[6] the actual implementation employs state for these variables, working as a caching mechanism which prevents unnecessary computations.

3.2.3 Obtaining feedback

In order to update the sending window, the sender requires feedback from receivers; the sender obtains this *feedback* by transmitting *polling requests*. When the sender transmits a polling request within a POLL or DATAPOLL packet, it includes the following information:

hs a copy of the value of *sw.hs* at the time of transmission;

polled a receiver set indicating receivers which should send back a response;[7]

ts a timestamp, the value of the sender's system clock at transmission time.[8]

When R_i receives a polling request (DATA)POLL, it checks whether its own id i is part of (DATA)POLL.*polled*. If so, R_i responds by transmitting a RESP packet to the sender containing:

rw the copy of its receiving window, rw_i;

ts a copy of the *ts* value which was received within the (DATA)POLL packet, that is, POLL.*ts*.

When the sender receives a RESP packet from R_i, it updates the RTT estimate between itself and R_i using the "fresh" RTT measurement brought by the poll response. An example of basic dialog between the sender and receivers is illustrated in Figure 3.5.

- At time 100, before transmitting the next data unit, the sender increments the value of *sw.hs* (that becomes 31), and transmits the packet of type DATA with DATA.*seq* = 31.
- At 102 *sw.hs* is incremented to 32, and the sender transmits the next data unit, a DATA packet with DATA.*seq* = 32.
- At time 104, for some reason, data units cannot be sent, and a polling request is transmitted (Section 3.4 explains the transmission of polling requests). The packet of type POLL contains POLL.*hs* = 32, to indicate that *sw.hs* was 32 at transmission time, POLL.*polled* = $\{R_2\}$, requesting R_2 to respond to the sender, and POLL.*ts* = 104, which is included in the response: RESP.*ts* = 104.

[7] *polled* is a bit vector of size *GS*, but can be alternatively implemented as a list of receivers' id's.
[8] this *ts* cannot be a simple counter because it is also used for RTT estimation.

- Some time later, when the POLL arrives at the R_2, the receiver sends back a RESP packet with a copy of the current rw and the ts received with the POLL.

- The RESP arrives at the sender at time 120, and the new RTT measurement is calculated: $120 - 104 = 16$ (the round-trip time as measured with the POLL/RESP pair was 16).

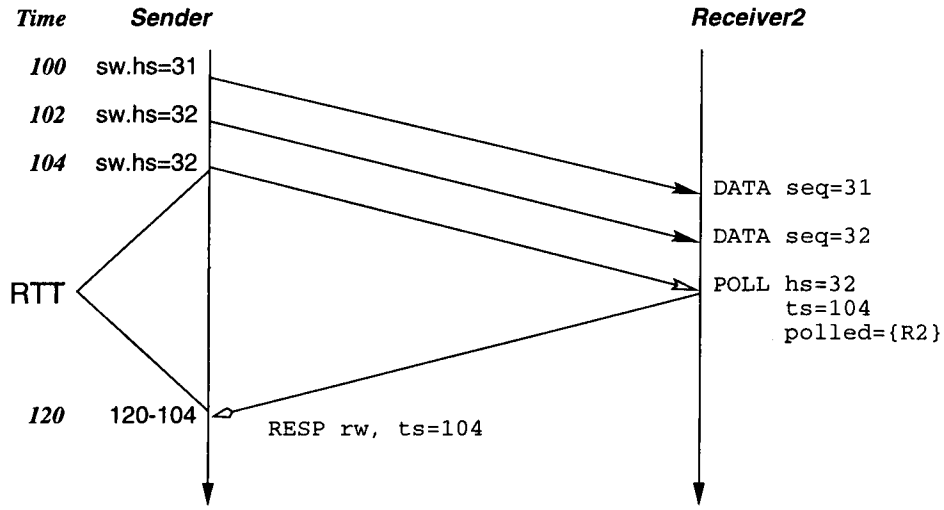


Figure 3.5: Example of basic dialog between sender and receiver.

It is possible to send more than one polling request without advancing $sw.hs$; this occurs when no new data packets are transmitted between successive polling requests. The timestamp RESP.ts enables the sender to uniquely identify pairs of POLLs and RESPs[9], and the order in which they occur.

3.2.4 Updating the sending window

The feedback provided by receivers allows the sending window to be updated. The receipt of a poll response from a receiver given R_i not necessarily leads to the update of sw_i. Upon receipt of a RESP packet, the sending window for R_i will *not* be updated if:

- *reordering*: either polling requests or responses are reordered by the network, so that a response can bring outdated state in comparison to the one currently stored in sw, or

- *unchanged*: the status at R_i did not change between the transmission of two responses.

[9]the protocol cannot transmit two or more requests at the same time.

Some reordered responses may be detected and discarded by the sender without further processing by comparing the current left edge with the one in the response: if $sw_i.le >$RESP.$rw.le$ then discard. More commonly, though, a RESP packet from R_i brings a rw such that RESP.$rw.le \geq sw_i.le$, and in such cases sw_i will be updated in the following manner:

1. update $sw_i.le$ with the le received with the response: $sw_i.le \leftarrow$RESP.$rw.le$.

2. update $sw_i.hr$ if it is smaller than the hr received with the response: if RESP.$rw.hr > sw_i.hr$ then $sw_i.hr \leftarrow$RESP.$rw.hr$.

3. for all packets of sequence seq directly represented by rw in RESP (that is, $\forall seq \mid$ RESP.$rw.le \leq seq \leq$RESP.$rw.hr$), update $sw_i.v$:

   (a) if $sw_i.v[seq] = 1$ (seq is ACK*ed*), then the new status in the response is ignored (note that a packet cannot be NACK*ed* after being ACK*ed*);

   (b) else, if $sw_i.v[seq] = 0$ (seq is *non*-ACK*ed*), and the new status in RESP is RESP.$rw.v[seq] = 1$ (ACK*ed*), then change state to ACK*ed* in sw_i ($sw_i.v[seq] \leftarrow 1$).

So, basically, $\forall seq$, RESP.$rw.le \leq seq \leq$RESP.$rw.hr : sw_i.v[seq] \leftarrow sw_i.v[seq] \vee$ RESP.$rw.v[seq]$. From the resulting sw_i, the sender can infer that R_i has received and consumed all data packets with seq such that $seq < sw_i.le$, received all packets with seq such that $le \leq seq \leq re \wedge sw_i.v[seq] = 1$, and not yet received any of the packets with seq such that $sw_i.nea \leq seq \leq sw_i.re \wedge sw_i.v[seq] = 0$. Not all of these "non-received" packets are regarded as NACKs by the sender, as explained below.

When processing RESP packets, the sender must observe the order in which packets were sent. The sender infers that all packets with sequence seq such that RESP.$rw.hr < seq \leq sw.hs$ were transmitted *after* (the transmission of) the polling request that generated the response, and so were not "included" in the RESP. Figure 3.6 illustrates this case through an example: when RESP arrives at t_4, it regards all packets up to packet $seq = 32$ (RESP.$rw.hr = 32$); packets $seq = 33$ and $seq = 34$ have been transmitted by the sender at t_2 and t_3, but the receiver has not had an opportunity to acknowledge them (the RESP packet that arrives at

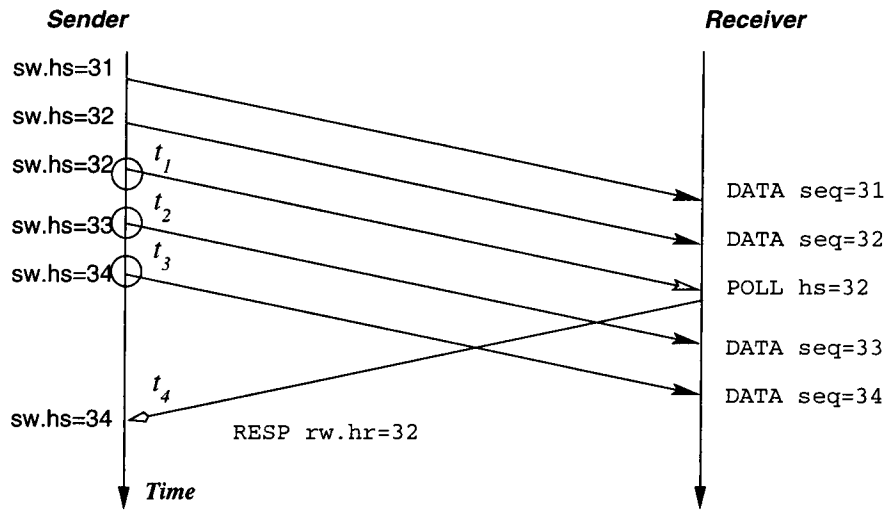


Figure 3.6: Example of POLL/RESP pair exchange.

t_4 *causally* precedes the DATA packets transmitted at t_2 and t_3). The sender relies on these inferences to avoid misinterpreting certain elements of RESP.v as packet losses.

Figure 3.7 presents an example of the window mechanism in action through a snapshot of sending and receiving windows at a given time. L is assumed to be 10. In R_1, the left edge of rw_1 is equal to the *seq* of the next expected data ($rw_1.le = rw_1.ned = 100$), which prevents packets $seq = 101$ and $seq = 102$ from being made available for consumption. $rw_1.hr = 104$ means that R_1 does not know whether the packets from $seq = 105$ onwards are yet to be transmitted by the sender or have already been transmitted. Let RESP_1 and RESP_2 be the latest responses that the sender has received from R_1 and R_2, respectively. Comparing sw_1 with rw_1 indicates that since R_1 has sent RESP_1 it has received packets $seq = 99$, $seq = 102$, and $seq = 104$, and its rw_1 has slid (to the right) by two packets due to the consumption of packets $seq = 98$ and $seq = 99$. Similarly, sw_2 and rw_2 indicate that after sending RESP_2, R_2 has received packets $seq = 101$, $seq = 102$, and $seq = 105$, but the window has not slid ($sw_2.le = rw_2.le$). The consumption at R_2 seems rather slow because the packets of sequence *seq* such that $97 \leq seq \leq 102$ (see rw_2) are "consumable" but remain "unconsumed". Finally, $sw.hs$, which cannot be smaller than $\max(rw_1.hr, rw_2.hr)$, is taken to be 105 in the figure.

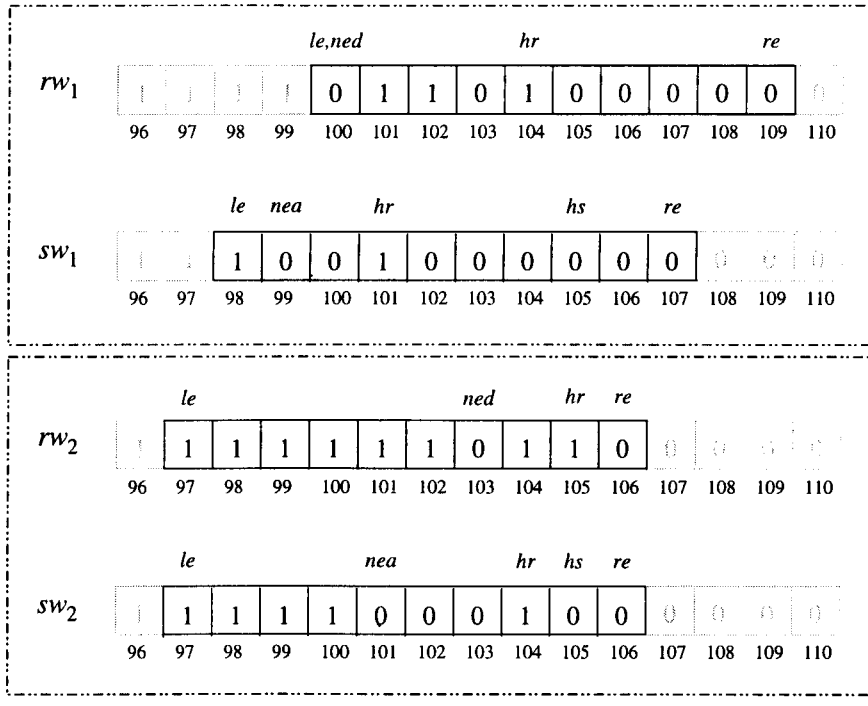


Figure 3.7: Snapshot of sliding windows.

3.3 Flow Control

PRMP employs window-based flow control, i.e., it uses the window to determine how many new packets can be safely multicast without causing buffer overflow at the receivers. Since the sender transmits packets to the entire destination set, the receiver judged to have the *smallest* number of free buffer spaces will be the limiting factor in the number of new transmissions. In other words, the pace in which new data packets are transmitted depends on the slowest receiver.

The aggregate window sw is used to determine how many new packets can be safely multicast, in the following manner. The left edge of sw is aggregated as the minimum left window currently recorded in all sending windows, that is,

$$sw.le \leftarrow \min\{sw_i.le \mid R_i \in \{*\}\}$$

Recall that the right edge derives from the left edge and length; so, the right edge of sw is computed as

$$sw.re \leftarrow \min\{sw_i.le \mid R_i \in \{*\}\} + L - 1$$

which can be expressed in a simpler form as

$$sw.re \leftarrow sw.le + L - 1$$

Before transmitting, the sender computes an "available window" (aw), which represents the number of *new* data packets that can be sent. It does so by computing the delta between the highest sequence number multicast so far ($sw.hs$) and the highest packet which even the slowest receiver is deemed able to store. Recall that the highest in-sequence packet a given R_i is able to receive is packet $rw_i.re$; thus, the sender can be certain that a given R_i can receive up to packet $seq = sw_i.re$. To the sender, the smallest $sw_i.re$ in sw indicates R_i as the receiver which appears to be the slowest. Therefore, the available window is computed as

$$sw.aw \leftarrow sw.re - sw.hs$$

The attribute $sw.aw$ completes the set of sending and receiving window attributes; they are summarized in Table 3.2.

Symbol	Name	sw	sw_i	rw
v	*bit vector*		√	√
le	*(sequence number of window's) left edge*	√	√	√
ned	*(sequence number of) next expected data (unit)*			√
nea	*(sequence number of) next expected* ACK	√	√	
hr	*highest (sequence number) recorded*	√	√	√
hs	*highest (sequence number) sent*	√		
aw	*available window*	√		

Table 3.2: Summary of window attributes.

The flow control strategy adopted by PRMP is conservative because of two reasons:

(a) the sender only transmits new data packets to a receiver when it can *guarantee* that there will be no buffer overflow at such receiver;

(b) the sender waits for the slowest receiver among all GS receivers.

More optimistic strategies in determining $sw.aw$ are possible, all of them based on the fact that one or more packets might be consumed since a receiver R_i transmitted its last response. Let "d" represent the time interval since the most recent response transmission by R_i. The time d is at least half RTT between the receiver and the sender, as receivers send feedback only periodically. Without losses, the longer the d, the more pessimistic the $sw_i.re$ estimate of $rw_i.re$

becomes. Additionally, when the sender calculates $sw.aw$, it does not take into consideration the half RTT that will take for the data to reach R_i.

Currently, PRMP avoids making assumptions about either the speed in which data is consumed at receivers or the set of RTTs; by preventing unnecessary retransmissions, PRMP emphasizes saving network bandwidth, which is the main motivation behind the use of multicast in the first place.

In addition to the window-based scheme, the protocol allows the user to set a *maximum transmission rate* by establishing an inter-packet gap (IPG). This IPG is defined as the minimum interval to be observed between the transmission of any two packets by the sender. Note that this does not require packets to be transmitted at times which are multiples of IPG; instead, it requires that any two transmissions must be at least 1 IPG apart.

The IPG is important to prevent the sender from transmitting a burst of data packets when the available window is large[10] (e.g., at the beginning of the transmission, when L packets can be transmitted). Another reason is that the polling mechanism of PRMP is timer-based, and the planning of polls (as described in the next section) depends on polling requests being sent at regular intervals after data has been transmitted.

3.4 Polling Mechanism

A sender-initiated unicast reliable protocol, such as TCP ([Stevens94]), typically triggers one ACK packet per DATA packet transmitted (or per 2 DATA packets transmitted, if the receiving end of TCP employs a "every-other-ACK" policy). A reliable multicast protocol, such as [Towsley87] and [Pingali94], leads to roughly one ACK per DATA packet per receiver. Expressing these in polling terms, the sender can be regarded as including a polling request to all receivers in every DATA packet it sends. Even though this allows the protocol to be simple at both ends, the protocol cannot scale due to ACK-implosion. To avoid implosion, PRMP is designed to request only a selected set of receivers at any given time so that the responses generated thereby do not arrive at a rate larger than some chosen value. This may result in a receiver not

[10] in general, protocols may employ a congestion control mechanism to prevent this problem; the discussion of PRMP's congestion control mechanism is delayed until Chapter 5.

being polled during the transmission of up to L data packets; so, a response from a receiver will "reference" (that is, positively or negatively acknowledge) not just a single packet, but all packets received/missed between successive polling requests (up to L data units).

Polling requests cause response packets to be transmitted to the sender. If the rate of responses exceeds a given threshold, there will be losses due to implosion. Such losses result from a shortage of resources at the host and network caused by the volume and synchrony of response packets. The maximum "allowable" arrival rate of incoming responses is defined as the *implosion threshold rate,* or ITR for short. Though it may not be possible to determine ITR precisely, it is assumed that ITR can be estimated with reasonable accuracy.

To avoid losses by implosion, the protocol controls the arrival rates and timings of response packets returned by receivers. The mechanism aims at implementing a given *response rate* (RR), the rate in which RESP packets arrive at the sender. RR is an input value of the protocol, likely to be taken from a default value associated with the capacity of the sending host hardware (some $RR \leq ITR$).

In general, the lower the RR, the smaller the likelyhood of implosion losses. Also, a smaller RR means that only fewer responses can be received in a given interval; this can lead to longer delays in obtaining feedback from all receivers. Recall that $sw.aw$ increases when sw slides forward, which is determined by $sw.le$ advance; because $sw.le \leftarrow \min\{sw_i.le \mid R_i \in \{*\}\}$, feedback from all receivers is required before sw can slide. Thus, using a smaller RR the sender may be blocked longer from making new transmissions, resulting in smaller throughput. A small RR used by the sender transmitting to a large group is likely to become the bottleneck, or limiting factor in throughput.

In order to keep the actual arrival rate of responses equal to RR in face of a potentially heterogeneous set of RTTs, the mechanism controls the arrival times of responses by planning ahead the time when every given receiver should be requested to respond. This mechanism is explained in more detail below.

Time is divided into *epochs*, intervals of fixed-length ε (measured in ms). As illustrated in Figure 3.8, epochs are denoted as E_n, with $n = 0, 1, 2..$, and E_0 corresponds to the first ε ms of the transmission. A *response quota*, denoted as RQ and calculated as $RQ \leftarrow \lfloor RR \times \varepsilon \rfloor$,

indicates how many responses can be allocated to each epoch. The sender estimates the arrival time of responses using the estimate of RTT between itself and each R_i, and schedules the transmission of polling requests such that at most RQ responses are expected during any epoch. For this purpose, a vector called *Anticipated Responses Count* (*ARC* for short) is maintained to keep track of the number of responses which have been allocated to each epoch. *ARC* is indexed by epoch number; $ARC[n]$ is initialized to 0 and incremented by 1 for every additional response which is expected to arrive during E_n.

The sender keeps an RTT estimate RTT_i for each receiver; the estimates are calculated like TCP, using a smoother α (e.g., $\alpha = 0.7$) for new RTT measurements.

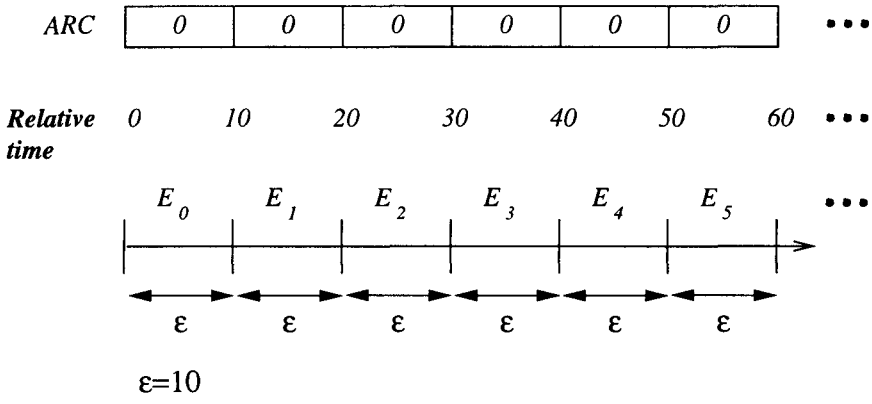


Figure 3.8: The division of time in epochs by the poll-planning mechanism.

The planning of a poll consists in determining what is the earliest time a polling request can be sent to R_i in order to evoke the desired response. If a polling request is immediately transmitted, the response is expected to arrive in RTT_i time, say epoch E_n. The planning is, however, restricted by the number of responses which are already being expected at E_n: it cannot exceed the quota RQ. If E_n cannot take another response, the arrival of RESP from R_i is delayed by some "*del*" time to a later epoch E_m ($m > n$), within which the response can safely arrive. The sending time of the polling request is delayed by the same amount of time *del*, to keep the difference between sending time and response arrival time equal to RTT_i. The time to send a polling request to R_i is planned using the following steps:

(step1) use the RTT_i to estimate the arrival time of a new response, and the corresponding epoch (E_n);

(**step2**) from E_n, check the ARC vector to find the first epoch "E_m" capable of taking another response ($ARC[m] < RQ$);

(**step3**) increment the corresponding entry in the ARC vector: $ARC[m] \leftarrow ARC[m] + 1$;

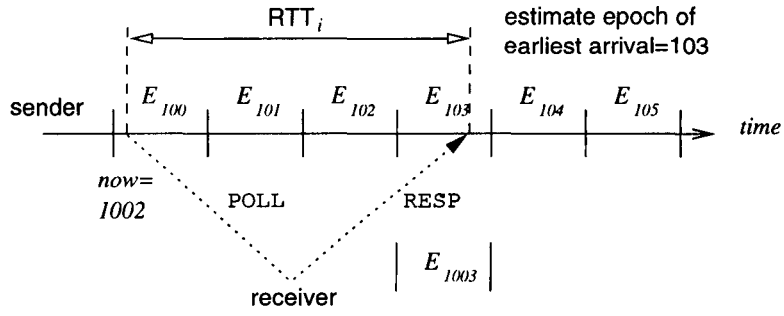
(**step4**) determine the *earliest sending time* of the polling request (denoted as "est_i") to R_i to be $est_i \leftarrow \max\{clock, m \times \varepsilon - RTT_i + begin\}$, where *clock* represents the value of the system clock and *begin* represents the value of the system clock at the beginning of the communication.

An example of the steps involved in the poll planning procedure is given in Figure 3.9. It is assumed: (a) quota per epoch $RQ = 3$ responses; (b) epoch length $\varepsilon = 10$ ms; (c) round trip time to receiver R_i $RTT_i = 35$ ms; (d) time at the start $begin = 1002$, and so current epoch is 100. In (**step1**), the sender adds the current time and RTT_i to find $1002 + 35 \Rightarrow 1037$, and determines the earliest epoch to be E_{103} ($\lfloor 1037/10 \rfloor \Rightarrow 103$. In (**step2**), the sender verifies that epoch E_{103} has already 3 responses ($ARC[103] = 3$), and finds the next epoch, E_{104}, with only 2 responses ($ARC[104] = 2$). In (**step3**), the entry of the vector for epoch E_{104} is updated ($ARC[104] \leftarrow 3$). Finally, in (**step4**), the sender determines est_i to be 1005, by subtracting the $RTT_i = 35$ from the time of E_{104}, 1040.

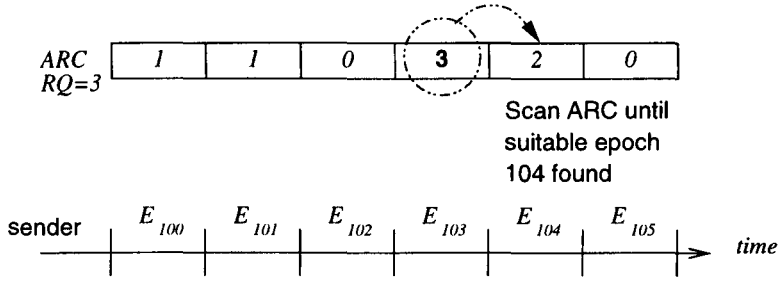
The mechanism maintains at most one planned polling ahead for each receiver, and this information is stored in a *Polling Table*; this table may contain up to one polling request planned for each receiver. A given receiver R_i is "added" to the i-th entry of the Polling Table when the sender *plans a time to send* a polling request to R_i. A poll is planned in any one of three situations, as long as R_i does not already have a planned poll assigned in the Polling Table:

(**before data**) before a data packet is transmitted to R_i (such a packet will be sent later as a DATA(POLL));

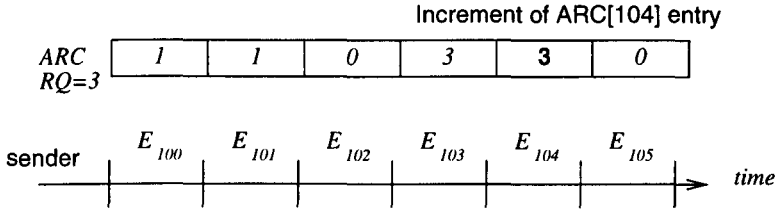
(**full buffer**) after feedback is received from R_i, which indicates that *all* packet buffers in R_i are occupied by unconsumed packets ("window full of 1s", i.e., $sw_i.v[seq] = 1$ for any seq such that $sw_i.le \leq seq \leq sw_i.re$); or



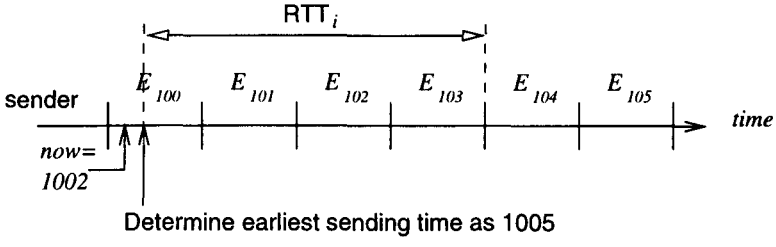
(a) step 1 - find the earliest arriving epoch



(b) step 2 - find suitable epoch



(c) step 3 - update the allocation vector



(d) step 4 - determine earliest sending time

Figure 3.9: The four steps involved in poll planning.

(**timeout**) when there is a retransmission timeout because a response from R_i failed to arrive within the expected time (more details in Section 3.5).

Receivers need eventually to acknowledge all data units they are sent; to acknowledge, they first need to be polled by the sender. The case (**before data**) above applies to transmissions and retransmissions of data. In the original transmission, the multicast of a given packet *seq* creates the demand to, sooner or later, send polling requests which will allow every R_i to acknowledge packet *seq* (and all packets with smaller sequence number). Thus, the polling mechanism ensures that all receivers will have a planned polling, planning new polls if required. In case of retransmitting *seq*, packets may be sent either via selective unicast or via multicast; if *seq* is sent via unicast to R_i, as R_i alone needs to acknowledge *seq*; as only R_i needs to be polled, only R_i is "added" to the Polling Table. If, however, *seq* is retransmitted via multicast, it may reach all receivers but only a subset may have to acknowledge *seq* (those which have requested the retransmission, as explained later in Section 3.6). In such cases, the polling mechanism will "add" to the Polling Table only those receivers which need to acknowledge the retransmission.

In case of (**full buffer**), the sender needs to continue to poll R_i because it cannot multicast any new data due to the lack of available buffers at R_i ($\forall seq \mid sw_i.le \leq seq \leq sw_i.re \wedge v[seq] = 1$). The sender plans a new polling request to R_i every time a response from R_i is received reporting full buffers. Consequently, RR permitting, the sender will poll R_i *once* every RTT_i until a response is received from R_i such that RESP.$rw.le > sw_i.le$ (buffers became available). The last case, (**timeout**), is related to the loss of polling requests and responses and its discussion is delayed until the next section.

After a plan to poll a receiver has been made, it stays in the Polling Table until its sending time (est_i) arrives, that is, the polling plan is due and a request is to be sent to R_i. The transmission of polling requests is carried out according to the timing information in the Polling Table. There are two types of situations which lead to the *examination* of the Polling Table for scheduled poll timings: before a data unit is transmitted, and when no data unit can be transmitted.

In the first case, the Polling Table is examined upon transmission of a DATA packet: after

all receivers have been added to the Polling Table, but before the packet is transmitted. The set of receivers with due scheduled pollings (that is, with earliest sending time $est_i \leq clock$) is "moved" from the Polling Table into a receiver set "*polled*"; if $polled \neq \{\}$, it is piggybacked onto the outgoing data packet (DATA becomes a DATAPOLL). The existing planning for each $R_i \in polled$ is erased from the Polling Table by changing est_i to an invalid time, $est_i \leftarrow -1$.

The second case refers to the situations where no data can be sent (either because the sending application is slow to produce data and there is not any to be sent, or because available data cannot be sent due to $sw.aw = 0$). In both cases, if there is any plan recorded in the Polling Table, the mechanism will carry out the transmission of a polling request on a POLL packet. The mechanism determines the next polling time (denoted as NPT), to be the minimum est_i currently recorded in the Polling Table. To prevent several POLL packets being sent close together (each POLL targeting a single receiver), the NPT is rounded up to the next IPG time:

$$NPT \leftarrow \max\{\min\{est_i \mid R_i \in \{*\} \wedge est_i \neq -1\}, lastTx + IPG\}$$

where $lastTx$ is the time of the most recent packet transmission. This ensures that the granularity in which polling requests are sent remains 1 IPG (no two POLL packets will be sent less than one IPG apart even when data cannot be sent). At the next NPT, a polling set is generated containing the set of receivers to be sent a polling request i.e., $polled = \{R_i \in \{*\} \mid est_i \neq -1 \wedge est_i \leq clock\}$; if $sw.aw$ remains 0 (no data can be sent), a POLL packet is transmitted, otherwise a DATAPOLL is sent.

Observe that in both cases in which the Polling Table is examined the mechanism allows up to one IPG to elapse between the *scheduled* and *actual* transmission times of a polling request. Note also that there is no *guarantee* that every given response will be received in the expected epoch. This is because the mechanism embodies three sources of unpredictability:

(d1) it allows a polling request planned for est_i to be up to 1 IPG after est_i;

(d2) the processing loads at the host CPU may cause the time between successive transmissions of polling requests to exceed IPG; this may further increase the difference between est_i and actual transmission times;

(**d3**) the RTT delays used are only estimates, and they need not be valid for the prevailing network conditions, particularly if conditions fluctuate widely.

Responses may thus arrive before or after the predicted time, and thus potentially outside the expected epochs. The amount of losses caused by such "rogue" responses are influenced by (**d1**), (**d2**) and (**d3**), as well as by the values adopted as ε. Using a higher value for ε (and thus higher RQ) results in fewer but larger epochs, decreasing the probability of a response arriving outside the expected epoch. However, with greater RQ, the arrival rate tends to be less uniform; this may cause some responses to be lost if all expected responses in a given epoch arrive *en masse* at the same point in that epoch.

3.5 Handling Absent Poll Responses

Polling requests and responses can be lost during transmission. So, to avoid waiting forever to receive a response from a given polled R_i, the sender waits on a retransmission timeout (RTO_i). The sender calculates the RTO_i based on the estimated RTT delay between itself and R_i, i.e., RTT_i. Since every RESP packet brings a fresh RTT measurement, the protocol can keep reasonably accurate RTT estimates without having to transmit additional packets.

PRMP's calculation of RTO_i is based on [Jacobson88]: this scheme keeps track of the RTT mean and deviation, and unlike the RTT estimate used for planning, it reflects the variation in the set of measured RTT values. The RTO_i is determined using a formula which "weighs" both RTT mean and RTT variance; the more "stable" the network is, the more similar measures will be, and the smaller/less important the variance becomes in RTO_i. In contrast, if measurements vary wildly, the value of the RTT mean will decrease but the RTT deviation will grow significantly.

After transmitting a polling request to R_i, if no response is received from R_i within RTO_i time, the sender assumes the expected response to be *absent*. The absence of a poll response from R_i can be due to one of the following:

(**c1**) the polling request transmitted by the sender did not reach R_i;

(**c2**) the response transmitted by R_i did not reach the sender;

(**c3**) RTT_i was underestimated, thus the RTO_i is too small, and the response is still in transit;

(**c4**) R_i has become permanently disconnected or failed.

It is impossible for the sender to know what exactly is the underlying cause for an absent response. The protocol deals with a suspected loss by "repolling" R_i and waiting (on timeout) for a response; this repolling is repeated for a finite number of times, with increasing RTO_i's (to cater for case (**c3**)). The status of receivers regarding the absence of responses is maintained in a separate table, the "Missing Polls Table". If the underlying cause is (**c1**), (**c2**), or (**c3**), then it is hoped that the sender will receive a response for at least one of the polls it has sent.

The absence of a poll response does *not* indicate the loss of DATA packets which were supposed to be ACK*ed*/NACK*ed* by that poll response. If a given R_i fails to respond within RTO_i, then R_i should be repolled as soon as possible. This is because the limiting factor in throughput is likely to be the absence of feedback from a receiver, increasing the likelyhood of a closed sw ($sw.aw = 0$). So, the polling mechanism plans a polling time for R_i with higher priority (and R_i is said to be in "repoll"). This high priority scheduling is done as follows.

First, RTT_i is used to estimate the epoch E_m in which the response would be received if the polling request to R_i were sent now (i.e., $est_i \leftarrow clock$). If this response can be allocated to the intended epoch E_m, assign the response to E_m using the standard procedure. However, if E_m cannot safely receive another response, examine the Polling Table looking for a non-repoll plan whose response has been allocated to E_m. If such plan exists (has not been sent yet), the right to receive a response at E_m is "stolen" from its holder R_j by R_i[11]. A new poll is planned to R_j using the normal procedure described in Section 3.4. The repoll scheduling algorithm is shown in Figure 3.10; est_i, $epoch_i$, and $repoll_i$ are the fields associated with each R_i in the Polling Table, used respectively to represent the earliest sending time, the epoch in which the response was allocated to, and if this plan is a repoll.

The sender assumes that if no response is received for a given number of consecutive polls[12],

[11] if more than one receiver R_j exists with such plan, there are two possibilities: to chose the one with smallest RTT_j, or to chose the one with largest est_j.

[12] this number is a user configurable, protocol variable, and represents how many times the protocol should retry in obtaining a reponse from a receiver (e.g., 10).

```
m ← (clock + RTT_i) mod ε  /* initial epoch */
repoll_i ← false
while not repoll_i {
  if ARC[m] < RQ { /* found available quota */
    ARC[m] ← ARC[m] + 1
    repoll_i ← true
  } else { /* try to steal from another receiver */
    if ∃R_j | i ≠ j ∧ epoch_j = m ∧ repoll_j = false {
      repoll_i ← true
      reschedule R_j as for normal plannings
    } else
      m ← m + 1
}
est_i ← max(m × ε − RTT_i + begin, clock)
epoch_i ← m
```

Figure 3.10: Repoll planning Algorithm.

then the underlying cause is (**c4**) and the non-responsive receiver is removed from the destination set $\{*\}$. Removing a persistently non-responsive receiver node from the destination set relieves the sender from having to wait for feedback from that node. The removal of a receiver R_j from $\{*\}$ occurs after a timeout, which causes sw to be re-examined: if $sw_j.le = sw.le$, then $sw.le$ is recomputed and may slide forward; if so, $sw.aw$ becomes greater than 0. With $sw.aw > 0$, the sender is allowed to transmit new packets. Further, as to be discussed in the next section, the removal of a receiver from $\{*\}$ may trigger packet retransmissions. Once a receiver is removed from $\{*\}$, any packet received from it is ignored.

3.6 Data Loss Recovery

The sender detects the loss of the packets it transmitted through the poll responses sent by receivers. The scheme employed by the sender to recover from these losses involves retransmitting the packets either via global multicast or selective unicasts. The way the scheme operates can influence the system throughput, network load, and the number of packets processed by a receiver. This section explains the rationale behind the design choices made for PRMP.

First, consider three simple, loss recovery schemes. The first one operates on the principle of *immediately* recovering from any detected loss: the loss of *seq* reported (through a NACK) by R_i is directly followed by a unicast retransmission of *seq* to R_i. Although very simple,

this scheme may be wasteful when a packet *seq* is lost by multiple receivers, since the same packet will be unicast multiple times. As discussed in Section 2.2.1, losses near the root of the multicast tree may lead to a given loss being shared by a large number of receivers. In such cases, it may be advantageous to retransmit via multicast.

The second scheme pessimistically assumes that if a packet is NACK*ed* by one receiver then it will be NACK*ed* by many other receivers as well. Based on this assumption, the packet *seq* is retransmitted via multicast soon after the first NACK for *seq* is received. When the number of receivers that share the same loss is likely to be large, this scheme speeds up recovery for those receivers whose NACKs have not yet reached the sender. On the other hand, resending a packet to a receiver that already has the packet incurs unnecessary network load and processing cost for that receiver. In the extreme case, a single lossy receiver can cause the rest of the group to be flooded with redundant retransmissions triggered by NACKs from R_i. Such a problem was coined by [Holbrook95] as the "crying baby" problem.

The third scheme uses a "wait-and-see" approach to minimize wasting of network bandwidth during recovery. It waits for a given time collecting NACKs; at the end of this waiting, if the number of collected NACKs exceeds a certain threshold value, the lost packet is multicast; otherwise the packet is unicast only to the appropriate receivers. Different criteria can be used to limit the waiting (e.g., a fixed interval, like the protocol described in Section 2.3.3) during which NACKs are collected. In general, the longer the collection time, the more appropriate the decision made for recovery will be, and hence the fewer unnecessary packets will be transmitted.

PRMP employs both the first and the third schemes, in the following manner. The third scheme is put in operation as soon as the sender receives the first NACK for a given packet *seq*. The sender waits collecting NACKs; if the percentage of receivers (out of GS) that has NACK*ed seq* reaches or exceeds the *multicast threshold ratio* (MTR), the collection period ends and the packet is retransmitted via multicast. If the number of NACKs does not reach the threshold, the collection will terminate after each receiver either has reported its status (ACK*ed*/NACK*ed*) regarding *seq* or is in repoll for having failed to respond to a polling request with $hs \geq seq$ (recovery is not delayed waiting for non-responsive receivers); this will be then followed by unicasting *seq* to each receiver that has NACK*ed seq*. After a multicast or a series of unicasts

is carried out, the workings of the third scheme terminate; the first scheme (i.e., immediate unicast retransmission) then becomes operative and will be in force until *seq* becomes fully ACK*ed*. The first scheme is more appropriate after the third scheme because the number of receivers which still require retransmissions of *seq* is likely to be small. Also, the resulting mechanism is simpler.

The waiting of the error control mechanism of PRMP is driven by a set of functions that, each compute a given kind of receiver set from the status currently stored in the sending window. These functions are the basis of the error control mechanisms, and will be described in Section 3.6.3.

3.6.1 Detecting data loss

Section 3.5 showed that the loss of *polling requests and responses* is detected by the sender via timeouts. Also, that the error control scheme of PRMP is *optimistic*, and does not interpret the absence of a response as the negative acknowledgment of all data packets that the response would (positively or negatively) acknowledge.

The loss of *data packets* is detected by receivers and reported to the sender through poll responses. The network may drop, duplicate or reorder packets; to allow receivers to detect such abnormalities in the set of packets they receive, the sender identifies each data unit with a unique sequence number. A receiver detects reordered, duplicate or absent packets by comparing DATA(POLL).*seq* with the corresponding packet status in its receiving window. Recall that, when polled, a receiver R_i transmits a copy of rw_i in a RESP packet; as discussed below, this allows the sender to detect losses experienced by receivers.

A receiver R_i keeps in its $rw_i.hr$ the highest sequence recorded so far; $rw_i.hr$ indicates to R_i that the sender has multicast all packets with $seq \leq rw_i.hr$. There may be one or more packets (with $seq > rw_i.hr$) which have been multicast by the sender but that are still in propagation. Any bits which (may) occur in $rw_i.v$ to represent any of these packets ($seq > rw_i.hr$) are set to 0, as R_i has not received the data of such packets. Further, $rw_i.hr$ indicates that R_i has not received a polling request with sequence number (DATA)POLL.$hs > rw_i.hr$. Therefore, when the sender receives RESP.rw, the data packets being negatively acknowledged are those whose

seq is such that $seq \leq rw.hr \wedge rw.v[seq] = 0$.

Figure 3.11 shows an example of the above loss detection process; it shows a communication in which the window length is 5 ($L = 5$). Suppose that the sender starts in the Figure with $sw.aw = 3$ and $sw.hs = 30$; it increments $sw.hs$ and multicasts DATA $seq = 31$, repeating this for packets DATA $seq = 32$ and DATA $seq = 33$. $sw.aw$ reaches 0 and some time later the sender transmits a POLL to R_i; such a POLL carries POLL.$hs = 33$. Packets DATA $seq = 32$ and DATA $seq = 33$ happen to be lost. When the POLL reaches the receiver, $rw_i.hr$ is updated to 33 (recall that $rw_i.hr$ records the highest sequence), exposing a "sequence gap" at $seq = 32$ and $seq = 33$ even though packet $seq = 34$ has not reached R_i. After updating rw_i ($rw_i.hr \leftarrow 33$), R_i verifies that its id is included in POLL.*polled* and as a result transmits a RESP packet. Meanwhile, feedback arrives at the sender from some other receiver (this is not shown in Figure 3.11), resulting into the slide of sw by two packets, that is, increasing $sw.aw$ to 2; with $sw.aw = 2$ the sender transmits two new packets: DATA $seq = 34$ and DATA $seq = 35$. As a result, $sw.hs$ becomes 35. When the poll response later arrives at the sender, RESP.rw contains RESP.$rw.le = 31$, RESP.$rw.ned = 32$, RESP.$rw.hr = 33$, and RESP.$rw.v = 10000$. RESP.rw informs the sender about the loss of packets $seq = 32$ (as RESP.$rw.v[32] = 0 \wedge$ RESP.$rw.ned \leq 32 \leq$RESP.$rw.hr$) and $seq = 33$ (as RESP.$rw.v[33] = 0 \wedge$ RESP.$rw.ned \leq 33 \leq$RESP.$rw.hr$). Finally, RESP.$rw.hr = 33$ and $sw.hs = 35$ together indicate that though packets $seq = 34$ and $seq = 35$ have been multicast, RESP.rw does not refer to them.

3.6.2 Packets susceptible to recovery

Before retransmitting, the loss recovery mechanism of PRMP waits so that it can judiciously decide between multicast and selective unicasts, depending on the number of receivers that have NACK*ed* the packet. After a packet seq has been NACK*ed* for the first time, ACKs and NACKs regarding seq coming from any receiver $R_i \in \{*\}$ may lead to the awaited retransmission of seq (e.g., because it may end the collection period). The range of packets seq that may be affected by recovery at any given time is (as explained below):

$$sw.nea \leq seq \leq sw.hr$$

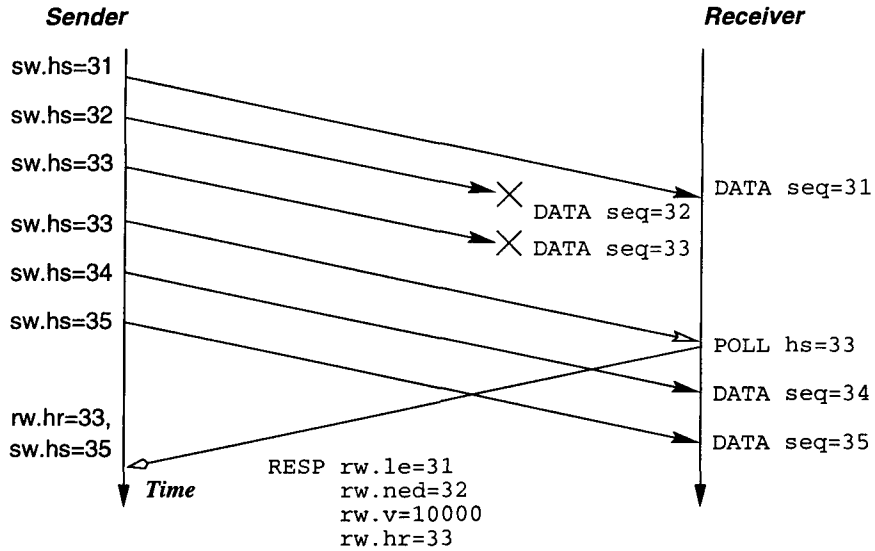


Figure 3.11: Example of case the sender selects which 0s are actually NACKs.

Recall that $sw_i.nea$ represents the first packet which R_i has not acknowledged. The aggregated $sw.nea$ represents the first packet which has not been *fully* acknowledged. Thus, $sw.nea$ is aggregated as the minimum of all next expected acknowledgments:

$$sw.nea \leftarrow \min\{sw_i.nea \mid R_i \in \{*\}\}$$

The attribute $sw_i.hr$ represents the highest *seq* a receiver R_i has *referenced* according to RESP.$rw.hr$ values. The $sw.hr$ represents the highest sequence to have been referenced (ACK*ed*/NACK*ed*) by *any* of the receivers in their responses; thus

$$sw.hr \leftarrow \max\{sw_i.hr \mid R_i \in \{*\}\}$$

Per definition, $sw.hr$ must be equal or less than $sw.hs$, as only transmitted packets can get acknowledged.

Therefore, only the packets with sequence *seq* such that $sw.nea \leq seq \leq sw.hr$ may be in recovery, as all packets with sequence *seq* such that $seq < sw.nea$ have been fully ACK*ed* and all packets with sequence *seq* such that $seq > sw.hr$ have been neither ACK*ed* nor NACK*ed* by any of the receivers.

3.6.3 Receiver set functions

As with other aggregated attributes, PRMP employs a function which aggregates the ACKs and NACKs with respect to a packet *seq* such that $sw.nea \leq seq \leq sw.hr$ and generates as a result

a receiver set. The state of a packet *seq* regarding R_i is determined through one of three predicates which examine the corresponding sw_i, *true* if:

Acked(R_i, seq): packet *seq* has been ACK*ed* by receiver R_i (i.e., $seq < sw_i.le \vee sw_i.v[seq] = 1$);

Nacked(R_i, seq): packet *seq* has been NACK*ed* by receiver R_i (i.e, $seq \leq sw_i.hr \wedge sw_i.v[seq] = 0$);

Refed(R_i, seq): packet *seq* has been *referenced* by receiver R_i (i.e., $seq \leq sw_i.hr$).

The above predicates are used to compute an aggregated state from sw, returning a receiver set when evaluated:

Acked(seq): receivers which have ACK*ed* *seq*, i.e., $\{R_i \in \{*\} \mid Acked(R_i, seq)\}$;

Nacked(seq): receivers which have NACK*ed* *seq*, i.e., $\{R_i \in \{*\} \mid Nacked(R_i, seq)\}$;

Refed(seq): receivers which have *referenced* *seq*, i.e., $\{R_i \in \{*\} \mid Refed(R_i, seq)\}$ (equivalent to $Acked(seq) \cup Nacked(seq)$).

Figure 3.12 shows one example of sw for $GS = 4$; all the attributes of sw (namely *le*, *nea*, *hr*, *re*, *hs*, and *aw*) are shown. Taking packet $seq = 104$ as an example, $Acked(R_1, 104)$ and $Acked(R_3, 104)$ are *true*, whereas $Acked(R_2, 104)$ and $Acked(R_4, 104)$ are *false*; thus, $Acked(104)$ is determined to be $\{R_1, R_3\}$. Likewise, $Nacked(104)$ is computed by applying $Nacked(R_i, 104)$ to the four sw_i's: $Nacked(R_1, 104)$ and $Nacked(R_3, 104)$ are *false* as $sw_i.v[104] = 1$; $Nacked(R_2, 104)$ is also *false* because R_2 has not referenced the packet yet (as $sw_2.hr < 104$); only $Nacked(R_4, 104)$ is *true*, as $sw_2.hr = 104 \wedge sw_2.v[104] = 0$. Thus $Nacked(104) = \{R_4\}$. Finally, $Refed(104)$ is computed as above, using $Refed(R_i, 104)$: $Refed(104) = \{R_1, R_3, R_4\}$, since all receivers apart from R_2 have $sw_i.hr \geq 104$.

There is an additional receiver set function, which has not been illustrated: *Repolled*(seq). This function determines the set of receivers according to the contents of the Missing Polls Table (see Section 3.5) using the following predicate, which is *true* if:

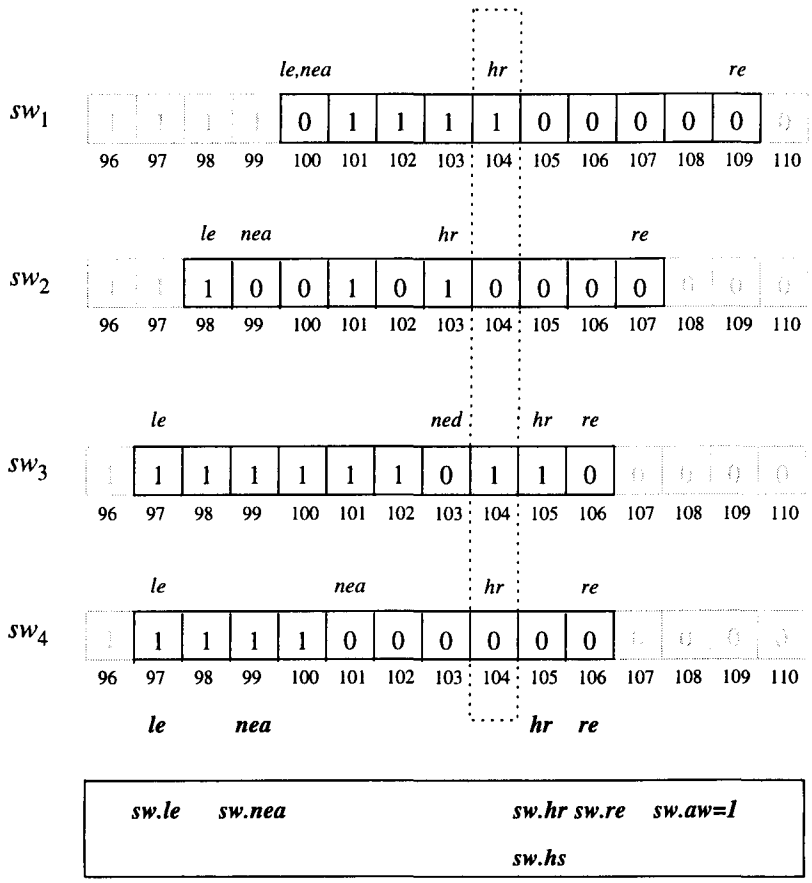


Figure 3.12: Example of sw attributes computed from a given sw.

Repolled(R_i, seq): the sender has failed to receive responses from R_i for all requests sent to R_i with (DATA)POLL.$hs \geq seq$ and $R_i \in$ (DATA)POLL.*polled*.

That is, if *Repolled*(R_i, seq) evaluates as *true*, it indicates to the sender that R_i's state in regards to packet *seq* is outdated. The predicate *Repolled*(seq, R_i) is used by an aggregating function which returns a receiver set:

Repolled(seq): receivers which have *not referenced seq because they are in repoll*, i.e., $\{R_i \in \{*\} \mid Repolled(R_i, seq)\}$.

3.6.4 Recovery of data transmissions

This section and the next complete the description of the error control mechanism of PRMP. A diagram (see Figure 3.13) is used to depict the set of states a given packet *seq* goes through; though the emphasis is on the loss detection and recovery, the complete "lifetime" of the packet is shown.

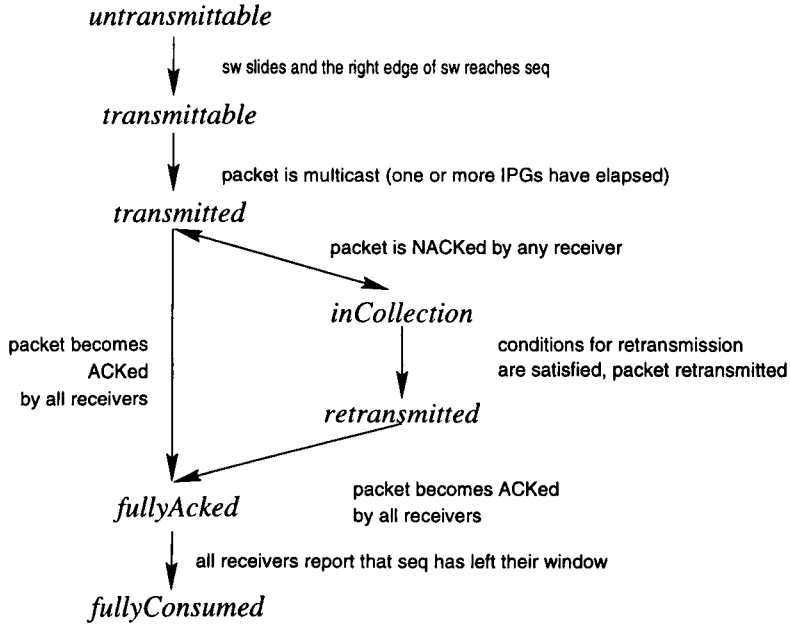


Figure 3.13: Diagram with packet life cycle.

At the beginning of the communication, the sending window has the following settings: $sw.le = sw.nea = 1$, and $sw.re = L$; as no packet has been sent yet, $sw.hs = 0$ and $sw.aw = L$. As the available window is equal to L, the first L packets of the sequence, i.e., $\forall seq \mid 1 \leq seq \leq$

L, can be transmitted. At any point in the communication, a packet *seq* can only be transmitted if $seq \leq sw.re$ (or otherwise the receivers may not have buffer space to accommodate the packet). So, all packets apart from the first L start in state *untransmittable*; packet *seq* remains in state *unstransmittable* while $seq > sw.re$.

The window *sw* slides forward and eventually $sw.re$ reaches *seq* (i.e., $seq \leq sw.re$), when it becomes possible to transmit packet *seq*; the packet becomes *transmittable*. However, because packets are sent in order and at a rate which is limited by the IPG, *seq* can only be transmitted when all packets with sequence smaller than *seq* have already been sent.

The packet *seq* is eventually transmitted, and its state changes accordingly to *transmitted*. At this point, $sw.hs$ is equal to *seq*. After the transmission of *seq*, all receivers need to be sent a polling request so that each can return a response containing a reference (ACK or NACK) to packet *seq*. The polling requests or responses that happen to be in transit when *seq* is transmitted will not refer to packet *seq*. That is, one or more responses may arrive containing RESP.rw such that RESP.$rw.hr < seq \leq$RESP.$rw.re$; these will have RESP.$rw.v[seq] = 0$, but are *not* negative acknowledgments (as explained in Section 3.6.1). As the sender transmits polling requests to receivers, RESP packets such that RESP.$rw.hr \geq seq$ begin to arrive; *seq* will be referenced at least once per receiver before it becomes fully acknowledged. A reference is either a 0 or a 1 in RESP.$rw.v[seq]$, and the sender interprets it as a NACK or a ACK, respectively. When the first response which negatively acknowledges *seq* (that is, $seq \leq$RESP.$rw.hr \wedge$RESP.$rw.v[seq] = 0$) arrives, the procedure associated with the loss recovery of *seq* starts; as shown in the diagram of Figure 3.13, the packet state changes from *transmitted* to *inCollection*. The packet remains for some time as *inCollection* so that any additional receivers (if any) which have also experienced the loss of *seq* have a chance to negatively acknowledge *seq* too.

With the arrival of new responses, *sw* is updated and *Acked*(*seq*) and *Nacked*(*seq*) may result in larger sets. The mechanism waits before retransmitting so it can decide between multicast or multiple unicasts; also, it is possible that the collection ends with the *cancellation* of the recovery process for packet *seq*. For the collection to end, one of these three conditions has to be satisfied:

(cancel retx) $Nacked(seq) = \{\} \wedge state = inCollection$: packet *seq* was "wrongfully"

NACK*ed* (see explanation below) by one or more receivers, which later rectified the situation by ACK*ing* *seq*, *before* it could be retransmitted;

(**mcast retx**) $|Nacked(seq)| \geq \lfloor MTR \times GS \rfloor$: the number of receivers that have NACK*ed* *seq* is sufficient to justify a multicast retransmission of *seq*;

(**ucast retx**) $Refed(seq) \cup Repolled(seq) = \{*\}$: all receivers in the destination set have either referenced *seq* or have failed (within time) to return a response which would have referenced *seq*.

In (**cancel retx**), after a RESP containing "ACK *seq*" is received and *sw* updated, $Nacked(seq)$ becomes empty. So, with $Nacked(seq) = \{\}$ there is no receiver to retransmit to, the recovery process is *cancelled*, and the packet status of *seq* changes back from *inCollection* to *transmitted* (note that the corresponding arrow in the diagram is bidirectional). This situation arises when the network reorders packets *seq* and $seq + 1$, and one or more receivers wrongly assume that the packet *seq* has been lost because of the sequence gap at *seq* (see example in Figure 3.14). If a receiver is polled after $seq + 1$ is received but before the arrival of *seq* (i.e., while

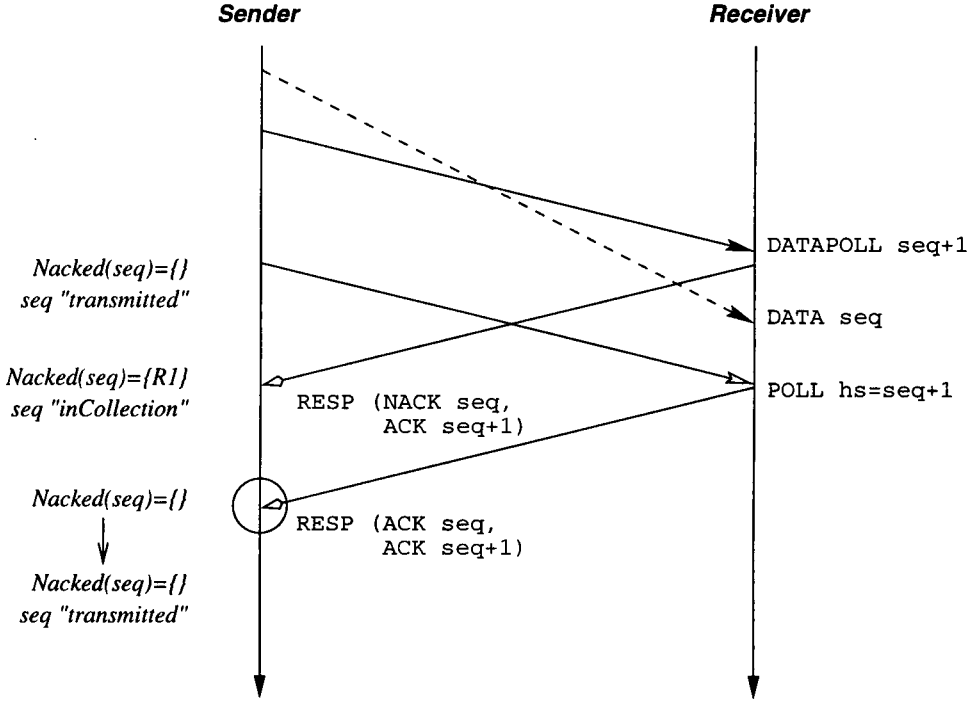


Figure 3.14: Example of false loss detection and recovery cancellation.

$rw_i.v[seq] = 0 \wedge rw_i.hr = seq + 1$), the receiver will send a copy of rw_i containing the gap (at *seq*) in the RESP packet transmitted to the sender. After transmitting such a response packet, the packet *seq* arrives at R_i. Suppose that packet *seq* stays as *inCollection* for some time; a time sufficient to allow the receiver to be polled again and, this time, positively acknowledge *seq*. After this response arrives at the sender, and $sw_i.v[seq] \leftarrow 1$, *Nacked*(*seq*) may become empty. If so, the recovery may or may not be cancelled: if the packet has not been retransmitted yet (state is *inCollection*), the recovery is cancelled; otherwise, (i.e., the packet has been retransmitted) there is no advantage in canceling the recovery and any NACKs which follow the retransmission will be treated with immediate retransmission.

When either (**mcast retx**) or (**ucast retx**) conditions are satisfied, recovery goes on and the packet *seq* is retransmitted to *at least* all receivers resulting from *Nacked*(*seq*) evaluation. Accordingly, the packet status changes from *inCollection* to *retransmitted.* Depending on the number of NACKs recorded for *seq*, the latter may be retransmitted via selective unicast to each $R_i \in$*Nacked*(*seq*), or retransmitted via multicast to all receivers in $\{*\}$. The length of time a packet spends in state *inCollection* varies, and can even be nil. This may happen if the *MTR* is equivalent to a single receiver, in which case the first "NACK *seq*" to arrive is enough to satisfy (**mcast retx**); it can also occur if the last reference to *seq* to arrive is the first "NACK *seq*". More precisely, the latter case will happen if *Nacked*(*seq*) $= \{\}$ and *Repolled*(*seq*)$\cup$*Refed*(*seq*) $= \{*\} - R_i$ before a "NACK *seq*" arrives in a RESP packet from R_i, making *Repolled*(*seq*)$\cup$*Refed*(*seq*) $= \{*\}$, and at the same time allowing *Nacked*(*seq*) to return $\{R_i\}$ instead of $\{\}$.

Suppose that the collection phase has ended and one or more retransmissions of packet *seq* have been done; the sender receives a NACK *seq* from R_i. As retransmitted packets themselves can be lost, the sender needs to re-send the packet. Having already used the third scheme of "wait-and-see" to decide which way to retransmit, the recovery mechanism switches to the first mechanism of "immediate retransmission" and without collection serves NACKs regarding the loss of the retransmitted packet; any re-retransmission is executed via unicast.

Now consider the case where soon after the retransmission of *seq* to R_i, the sender receives a NACK *seq* from R_i: if the polling request which originated this response was sent *before*

the retransmission, then the NACK *seq* should be ignored, as it is *obsolete* with respect to the retransmission. After the retransmission of *seq* to the set of receivers indicated by *Nacked*(*seq*), each of these receivers needs to be polled so that it can positively or negatively acknowledge *this retransmission*. With the aid of an example, Section 3.6.5 discusses how the mechanism distinguishes the former NACKs (obsolete) from the latter (valid) ones.

After all receivers have acknowledged *seq*, on computation *Acked*(*seq*) results in $\{*\}$, and the packet state changes to *fullyAcked*. The packet *seq* may be released from the buffers, as a retransmission of *seq* will not be necessary. When the sender learns that all receiving applications have consumed the data corresponding to *seq* ($\forall i : sw_i.le > seq$), *seq* reaches its last state, *fullyConsumed*, and the sender can advance the *sw*.

3.6.5 Recovery of data retransmissions

This section explains through an example the problem of *obsolete* NACKs and how the PRMP mechanism prevents them. Consider the scenario in Figure 3.15; the sender transmits three packets: DATA $seq = 31$, DATAPOLL $seq = 32$ $hs = 32$, and a packet POLL.$hs = 32$. Packet DATA.$seq = 31$ is lost, but both the DATAPOLL $seq = 32$ and POLL.$hs = 32$ arrive at the receiver and trigger responses from R_i. The first response contains RESP.rw such that: RESP.$rw.v[31] = 0 \wedge$RESP.$rw.hr \geq 31$; that is, the response contains a "NACK 31". Suppose that after this response either (**mcast retx**) or (**ucast retx**) are satisfied, and DATA.$seq = 31$ is retransmitted. Soon after the retransmission of DATA.$seq = 31$, the second response arrives, which still negatively acknowledges packet $seq = 31$. As shown in Figure 3.15, a simplistic approach wrongly infers that the retransmission of packet DATA.$seq = 31$ has been lost, and re-retransmits DATA.$seq = 31$.

When the sender receives a NACK *seq* from receiver R_i, what makes this NACK valid or obsolete is whether the NACK refers or not to the *most recent* retransmission of *seq* to R_i. The sender establishes the *causal relation* between (the response which contains) the NACK and the most recent retransmission of *seq* to R_i; to be valid, the NACK must causally succeed the retransmission. The causal relation is worked out using timestamps; recall that when the sender transmits a given polling request, it includes its transmission time: (DATA)POLL.ts is used to

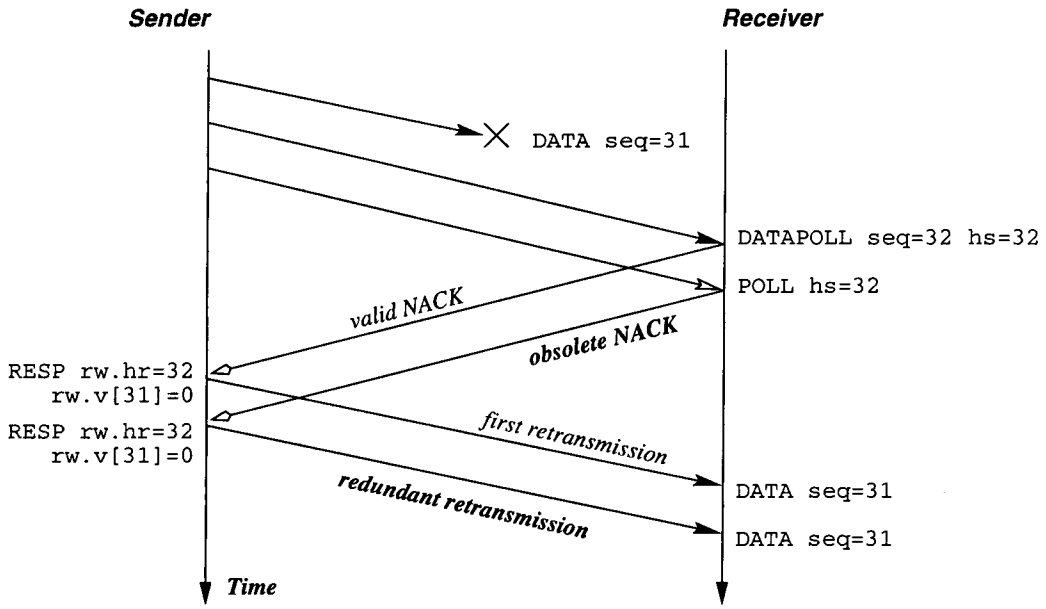


Figure 3.15: Example of obsolete NACK and redundant retransmission.

allow RTT estimation (see Section 3.4). When a response is received from R_i containing a NACK to a retransmitted packet (of sequence *seq*), the sender compares the timestamp RESP.ts with the time of the latest retransmission of *seq* to R_i: if the time ts in RESP is equal to or larger than the time of the latest retransmission of *seq*, then the NACK is valid.

The times of retransmissions are recorded in a special table, the "Retransmission Times Table". The table has $GS + 1$ entries: one entry per receiver, to record times of selective retransmissions, and one entry to record times of global multicast retransmissions. Each entry contains up to L tuples of the form $\langle seq_i, ts_i \rangle$, where seq_i indicates a sequence of a packet retransmitted to R_i and ts_i the time it was sent. In case of multicast retransmission, the tuple is denoted as $\langle seq_m, ts_m \rangle$.

The tuples recorded in the Retransmission Table indicate past multicast and unicast retransmissions of *seq* to receivers. There are three possibilities regarding the retransmissions of *seq*: (a) *seq* has *not* been retransmitted yet; (b) *seq* has been retransmitted via multicast; (c) *seq* has been retransmitted using selective unicast. According to the workings of the recovery mechanism as defined in Section 3.6.4, (b) can only occur before (c). However, because at the time of a retransmission is executed one or more receivers may need to be sent a polling request, the destination set for the retransmission may grow and a selective unicast retransmission to

some receivers may be transformed into a multicast retransmission; hence, (b) and (c) can occur multiple times and in any order. To determine the time of the most recent retransmission of packet *seq* to receiver R_i, the mechanism looks up if multicast and "selective unicast to R_i" tuples exist:

only $\langle seq_i, ts_i \rangle$	*seq* was retransmitted via selective unicast to R_i at time ts_i;
only $\langle seq_m, ts_m \rangle$	*seq* was retransmitted via multicast at time ts_m;
both $\langle seq_i, ts_i \rangle$ and $\langle seq_m, ts_m \rangle$	*seq* has been retransmitted via multicast at time ts_m and via unicast to R_i at time ts_i;
neither $\langle seq_i, ts_i \rangle$ nor $\langle seq_m, ts_m \rangle$	*seq* has not been retransmitted to R_i (however, if $st(seq) =$ *retransmitted* then *seq* was retransmitted via selective unicasts to a set of receivers which did not include R_i).

Upon arrival of a RESP packet from a given receiver R_i, if the response negatively acknowledges a packet which has been already retransmitted (i.e., $st(seq) = retransmitted$), the the procedure followed by the recovery mechanism is determined according to the tuples in the Retransmission Times Table and the state in *sw*, as follows:

- if only $\langle seq_i, ts_i \rangle$ exists and if RESP.$ts \geq ts_i$ (valid NACK), *seq* will be retransmitted to R_i;
- if only $\langle seq_m, ts_m \rangle$ exists
  - if RESP.$ts \geq ts_m$ (valid NACK), *seq* will be retransmitted to R_i;
  - if RESP.$ts < ts_m$ (obsolete NACK), no retransmission is required; however, make sure that there exists a polling request to R_i succeeding ts_m by adding R_i to the Polling Table;
- if both $\langle seq_i, ts_i \rangle$ and $\langle seq_m, ts_m \rangle$ exist and RESP.$ts \geq \max\{ts_i, ts_m\}$ (valid NACK), *seq* will be retransmitted to R_i;
- if neither $\langle seq_i, ts_i \rangle$ nor $\langle seq_m, ts_m \rangle$, the packet *seq* will be retransmitted to R_i.

The example of Figure 3.15 has been modified to show how the second retransmission of $seq = 31$ is avoided (see Figure 3.16). In ① , DATA packet $seq = 31$ is transmitted and lost; in ② , packet DATAPOLL $seq = 32$ arrives at the receiver, allowing the latter to assume the loss of $seq = 31$; since the packet contains a polling request, the receiver sends a response NACK*ing* $seq = 31$. In ③ , a new polling request arrives (in a POLL packet) at the receiver and triggers a new response (the status in rw_i regarding $seq = 31$ has not changed, so that $seq = 31$ is still being NACK*ed*). In ④ , the poll response arrives at the sender; seq has not been retransmitted yet; assume that the threshold corresponds to 1 receiver, and a retransmission takes place. In ⑤ , the second response arrives; $seq = 31$ has been retransmitted, and the status in the Retransmission Times Table indicates that the most recent retransmission of $seq = 31$ to R_i has been at time 160 ($\langle 31_i, 160_i \rangle$), therefore *after* the transmission of the polling request which generated this NACK, that is, RESP.$ts = 120$.

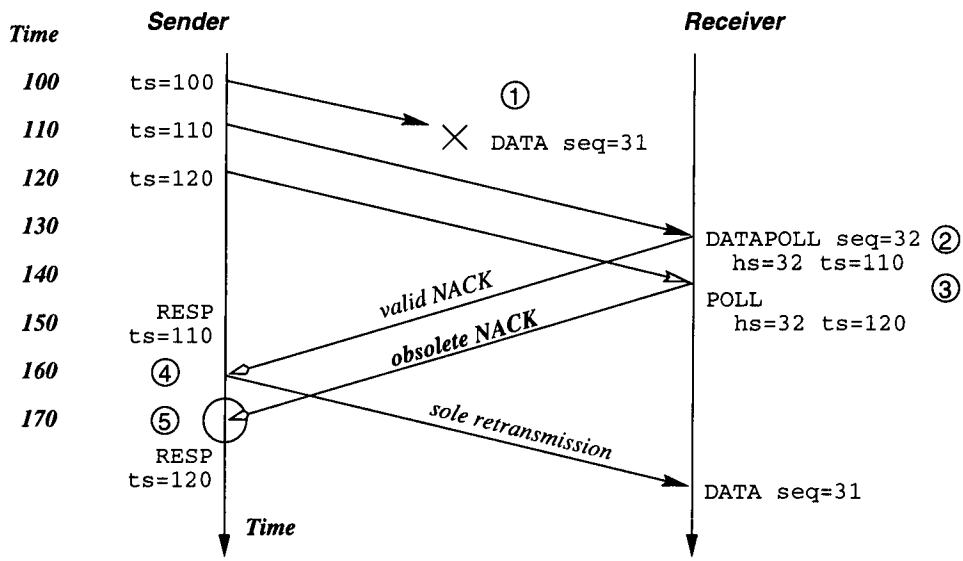


Figure 3.16: Recovery mechanism identifies the obsolete NACK and prevents the redundant retransmission.

3.6.6 Recovery algorithm

Figure 3.17 presents in pseudo-code the error control algorithm used for loss detection and recovery of data packets; such code is executed upon the arrival of a poll responses from a given receiver R_i. A packet seq may be in one of three states in this algorithm: *transmitted*,

inCollection, or *retransmitted*, as indicated by $st(seq)$.

Recall that, when evaluated, $Repolled(seq)$ returns the set of receivers which have failed to respond to all polling requests (DATA)POLL, such that (DATA)POLL.$hs \geq seq$. Therefore, when a retransmission timeout expires, $Repolled(seq)$ may return a larger receiver set, which may be sufficient to satisfy condition (**ucast retx**). Figure 3.18 shows the algorithm required to handle a retransmission timeout of a polling request.

```
if (rw.le > sw_i.le) sw_i.le ← rw.le /* update left edge */
for ∀seq such that sw_i.nea ≤ seq ≤ rw.hr {
  if (sw_i.v[seq] = 1) continue; /* already acked, ignore */
  if (sw_i.hr ≤ seq ≤ rw.hr) { /* first ref to seq */
    if (seq < rw.ned ∨ rw.v[seq] = 1) { /* this reference is ack */
      sw_i.v[seq] ← 1 /* update the window */
      if (st(seq) = inCollection ∧
          Refed(seq) ⋃ Repolled(seq) = {*}) {
        plan poll for each R_i ∈ Nacked(seq)
        retransmit seq to Nacked(seq)
        record retransmission time(s)
        st(seq) ← retransmitted
      }
    } else { /* first reference from receiver, and is a nack */
      if (st(seq) = transmitted) st(seq) ← inCollection
      if (st(seq) = inCollection) {
        if (|Nacked(seq)| ≥ MTR × GS ∨
            Refed(seq) ⋃ Repolled(seq) = {*}){
          plan poll for each R_i ∈ Nacked(seq)
          retransmit seq to Nacked(seq)
          record retransmission time
          st(seq) ← retransmitted
        }
      } else if (st(seq) = retransmitted) {
        /* packet has been retx'ed, but not aiming at this receiver*/
        plan poll for R_i
        if (¬∃ ⟨seq_m, ts_m⟩ ∨ t_m <RESP.ts) {
          retransmit seq to R_i
          record retransmission time
        }
      }
    }
  } else { /* there was a previous reference (nack) */
    if (rw.v[seq] = 0 ∧ rw.nea ≤ seq ≤ rw.hr){/*rw has another nack*/
      if (st(seq) = retransmitted∧/* already retransmitted */
        RESP.ts ≥ max{ts_m, ts_i}){ /* valid nack */
        plan poll for R_i
        retransmit seq to R_i
        record retransmission time
      }
    } else { /* this reference is ack */
      sw_i.v[seq] ← 1
      if (st(seq) = inCollection ∧ |Nacked(seq)| = 0)
        st(seq) ← transmitted /* cancel recovery */
    }
  } /* there was previous reference */
} /* for all packets which have been referenced by response */
if (rw.hr > sw_i.hr) sw_i.hr ← rw.hr /* update the highest recorded */
while (sw_i.v[sw_i.nea] = 1) sw_i.nea ← sw_i.nea + 1 /* advance nea */
```

Figure 3.17: Recovery algorithm upon arrival of RESP packet.

```
/* after RTO, Repolled(seq) may result into a larger set */
/* poll.hs is the highest sequence sent with the expired poll */
for ∀seq such that sw.nea ≤ seq ≤ poll.hs {
  if (st(seq) = inCollection ∧ Refed(seq) ∪ Repolled(seq) = {*}) {
    /* retransmit using selective unicast */
    st(seq) ← retransmitted
    retransmit seq to R_i ∈ Nacked(seq)
    plan poll for R_i ∈ Nacked(seq)
    record retransmission time(s)
  }
}
```

Figure 3.18: Recovery algorithm upon retransmission timeout.

Chapter 4

Prototyping & Simulation of Flat PRMP

This chapter is divided in three main parts. First, Section 4.1 describes the protocol architecture designed to realize flat PRMP. A prototype of the protocol was implemented using this architecture; it was used to carry out simulation experiments under various settings, allowing the analysis of flat PRMP's behavior. Section 4.2 describes such experiments, and discusses in depth the results obtained. The third and final part, in Section 4.3, reviews the simulation results, discusses the main findings and comments on their implications regarding the scalability of PRMP.

4.1 Protocol Architecture

As shown in Chapter 3, most of the complexity of the protocol lies at the sender. The architecture which implements the protocol at the sender is "multi-threaded": each *thread* runs synchronously (never preempted by another PRMP thread) and, continuously following the same basic set of steps, executes a simple task of the protocol. The set of threads mainly interacts according to the "producer-consumer" model: items are "produced" and stored in priority *queues* by threads, and "consumed" from such queues by other threads. Threads also cooperate by reading and writing state stored in shared *tables*. The architecture is object-oriented: tables,

queues and threads are implemented as objects. So, the architecture is basically designed using three kinds of components:

Queues store items which are to be processed by a given thread;

Tables keep protocol state in general, including windows and other recovery information;

Threads active entities, scheduled non-preemptively.

4.1.1 Queues

The sender employs two priority queues:

`ToQ` *Timeout Queue*;

`TxQ` *Transmission Queue*;

The `ToQ` is a priority queue containing asynchronous events. It is sorted according to the event time associated with each entry, so that the head of `ToQ` represents "the next asynchronous event", and the system timer can be set with the event time of this element. There are two kinds of events: retransmission timeouts (RTOs) and "poll gaps".

Recall from Section 3.5 that after transmitting a polling request to a receiver R_i, the sender expects to receive a response within RTO_i time. The absence of RESP within the expected time constitutes a retransmission timeout. Note that as responses provide cumulative acknowledgment of packets, any RESP succeeding the poll, that is, RESP.$ts \geq$(DATA)POLL.ts, is sufficient to prevent the timeout.

Most entries of `ToQ` are retransmission timeout events. To reduce the number of RTO events, a single event is created for each (DATA)POLL sent, with the maximum retransmission timeout among polled receivers. That is, the expire time of a retransmission timeout event is determined according to the maximum RTO_i in the set of receivers being polled: $clock + \max\{RTO_i \mid R_i \in$ (DATA)POLL.$polled\}$.

After transmitting a polling request, the sender creates and enqueues an entry "`to`" in `ToQ`; the entry contains: `to.ev_time`, the event time (the sorting key), which is set to the expiration

time of the RTO; `to.ts`, the timestamp on (DATA)POLL.*ts*; `to.hs`, the value of *sw.hs* at transmission time (a receiver is supposed to acknowledge all packets up to `to.hs`); `to.to_resp`, the set of receivers from which a response is being expected, initially set to (DATA)POLL.*polled*; and `to.type=RTO`, indicating an event of type retransmission timeout.

Whenever a response packet arrives from R_i, all retransmission timeouts of requests (regarding R_i) which (causally) precede the response are cancelled. This is achieved by removing R_i from all `to.to_resp` in entries `to` such that `to.ts` $\leq$ RESP.*ts*. If `to.to_resp`$= \{\}$, the `to` is removed.

The second kind of entry in `ToQ` is the poll gap. Recall that when no data can be transmitted, but there is one or more receivers that need to be sent a polling request (i.e., $\exists R_i \in \{*\} \mid est_i \neq -1$), such request is transmitted through a POLL packet. A POLL is to be sent at $\min\{est_i \mid est_i \neq -1\}$ or no more than one IPG after this time; the poll gap is the period between now and the time the next POLL is to be sent. A poll gap entry in `ToQ` represents the asynchronous event that is the *end* of the poll gap. This entry has two fields: the entry type `to.type`, set to `POLL_GAP`, and its event time, `to.ev_time`. When the event time arrives, it indicates that the Polling Table should be examined for due polls. If data cannot be sent, an entry `POLL_GAP` will be created at the Transmission Queue, `TxQ` (see below), else the poll gap is ignored.

The `TxQ` contains entries that are related to the transmission of packets. Recall that packets may contain a data unit, a polling request, or both, and that the packet type is determined accordingly. Each entry "`tx`" in `TxQ` consists of: `tx.seq`, a sequence number (which is the key); `tx.type`, indicating the entry type; and `tx.dest`, the destination set for the packet. The entry type can be any of three: `DATA_ORG`, `DATA_ENTRY`, or `POLL_GAP`, which respectively represent the original multicast transmission of a data packet to all receivers, a retransmission to a subset of receivers, and the end of a polling gap. `DATA_ORG` entries have `tx.dest`$= \{*\}$, while in `tx.type=DATA_RETX` entries `tx.dest` will indicate the set of receivers which are to be re-sent `tx.seq`. Entries of `TxQ` are kept sorted in ascending order according to `tx.seq`. As entries are consumed from the head of the queue, retransmissions have priority over normal transmissions: if there exists an entry `tx` with `tx.type=DATA_RETX` and `tx.seq` $=$ *seq* (for the retransmission

of packet *seq*), as well as an entry `tx'` with `tx'.type=DATA_ORG` and `tx'.seq`= *seq*+1 (original multicast of *seq* + 1), `tx` will be processed before `tx'`, and *seq* will be retransmitted before *seq* + 1 is transmitted.

The third type of entry in `TxQ` is the `POLL_GAP`. Recall that the Timeout Queue may hold a "poll gap event" to mark the end of a poll gap. When such a poll gap arrives, one or more planned polling requests may be due. To force the examination of the Polling Table, a `TxQ` entry of type `POLL_GAP`, with `tx.seq`= 0 (highest priority) is enqueued in `TxQ`; when consumed, this entry will lead to the examination of the Polling Table, and transmission of a polling request with due polls.

`tx.seq` uniquely identifies an entry, and hence there cannot be more than one entry with the same `tx.seq` at a given time. In case of data units, multiple retransmission entries can be joined together, by merging their `tx.dest` sets. Recall from Section 3.6.4 that when a retransmission is NACK*ed* the mechanism immediately deals with it: a unicast retransmission entry is enqueued in `TxQ`. As it may take some time before this entry is consumed from `TxQ`, several requests to retransmit *seq* may be joined while awaiting transmission, enlarging `tx.dest`. When `tx` is finally processed, `tx.dest` may be further enlarged with the receivers to be sent a polling request now (if any). Thus, it is possible that *seq* is retransmitted via multicast, even though there have been previous retransmissions of *seq*.

4.1.2 Tables

The receiver only maintains a single structure, the receiving window. The sender has the following tables:

`PPT` *Planned Polling Table*

`MPT` *Missing Poll Table*

`RT` *Response Table*

`RST` *Recovery Status Table*

`RTxT` *Retransmission Times Table*

There are two tables that hold polling information, the PPT and the MPT. The PPT is the realization of the Polling Table described in Section 3.4; it has GS entries, each containing three fields: `PPT[i].est`, the earliest sending time; `PPT[i].epoch`, the epoch in which the response is expected to arrive; and `PPT[i].repoll`, a flag to indicate whether the entry represents a poll or a repoll. The value of `PPT[i].est` is -1 if there is no poll planned in the entry. The PPT object provides three main methods: before transmitting a data packet, "add a plan for a given subset of receivers to PPT"; after adding receivers, "generate a polling set with the due planned polls"; and when data cannot be sent, "return the time of the next planned poll to be due".

RT is the implementation of the sending window abstraction; as such, it contains the GS sending windows and provides operations on individual windows. Each sending window is implemented through a bit vector. RT has methods for evaluation of *Acked*(seq), *Nacked*(seq), etc. receiver sets, based on the GS sw_i's. It also provides the *aggregation* of attributes defined for sw. An aggregated attribute is determined upon demand by finding the $\min\{\}$ or $\max\{\}$ value for a given attribute in all sw_i's. This operation is optimized by RT with the aid of a *caching* mechanism, which is best described through an example. Consider $sw.le$: its value is determined as the minimum $sw_i.le$. If the value of $sw.le$ is computed repeated times without any change of state in sw, it will successively return the same value; in such cases, it is sufficient to compute $sw.le$ only the first time, storing the result, and then using the stored value the remaining times. When there is a change of state in sw that may affect the value of $sw.le$, the latter is marked "dirty". The mechanism will then recompute $sw.le$ the next time its value is required. For aggregated attributes that represent the minimum of a packet sequence (namely, $sw.le$, $sw.re$, and $sw.nea$), the caching mechanism may achieve substantial gain, due to the comparatively low frequency in which they need updating: as the values of $sw_i.le$, $sw_i.re$, and $sw_i.nea$ of any R_i are packet sequences which can only increase, aggregated attributes need only to be recomputed when the currently smallest value is increased. Thus, on average, $sw.le$ will have to be recomputed only in 1 out of the GS responses. For the other attributes, such as $sw.hr$, the maximum $sw_i.hr$, the gain is reduced: recomputing is required whenever any $sw_i.hr$, $sw_i.hr \neq sw.hr$, is increased.

To implement the caching, instead of keeping $sw.le$'s value, the mechanism keeps i, the id of the receiver with the smallest $sw_i.le$; it makes $i = 0$ if the $sw_i.le$ is "dirty" ($sw_i.le$ is not necessarily the smallest $sw_i.le$ anymore and hence $sw.le$ needs to be recomputed). The variable i is set to 0 if, after the receipt of a response from R_i, $sw_i.le$ is increased. (In case there are several receivers with the smallest value, i will point to the first one.) Whenever $sw.le$ is required (i.e., "read"), i's value is checked: if $i = 0$, before returning $sw_i.le$, recompute i. (In the experiments described in the next section, it was observed that the mechanism avoided between 87 and 90% of all $\min\{\}$ and $\max\{\}$ computations.)

RST is implemented to efficiently provide the variable $st(seq)$ described in the diagram of Section 3.6.4. Despite the fact that there are L entries in the table (one per packet in the window), only the packets with sequence seq such that $sw.nea \leq seq \leq sw.hr$ need to be represented. These are the packets which may be in recovery (they have been transmitted and referenced by at least one receiver). Each entry has a single value, the current status of the packet, namely: *transmitted*, *inCollection*, or *retransmitted*. The table is used by the sender to quickly determine the state of packet seq while doing error control; for example, to separate a valid from an obsolete "NACK seq", the sender only looks up the tuples recorded in RTxT if RST[seq] =*retransmitted*.

The RTxT is the realization of the Retransmission Times Table mentioned in Section 3.6. Recall that while recovering the loss of packet seq of R_i, seq may be retransmitted via multicast depending on the number of receivers resulting from $Nacked(seq)$ evaluation (of RT); if not, the packet is retransmitted to R_i via unicast. In order to prevent redundant retransmissions, RTxT is used to record the times of retransmissions. RTxT has $GS + 1$ entries, one per R_i, plus a "multicast entry". Each entry contains a list of tuples, each tuple representing the latest retransmission of a packet seq to R_i, or $\{*\}$ in the case of the multicast entry. Each tuple has the sequence number and timestamp of the retransmission; the tuple list is kept in ascending order according to seq.

Besides these five tables, there is the ARC vector, which is used for the allocation of responses to epochs. Each entry of the vector holds a value in the range $0..RQ$. The size of the vector depends on how many bits are required to represent the value RQ and on how many epochs

ahead the polling mechanism plans response arrivals. The latter depends on the maximum RTT among all receivers, RTT_{max}, and the epoch length, ε. Since at most one poll is planned ahead for each receiver, the maximum number of entries in ARC is $GS + \left\lfloor \frac{RTT_{max}}{\varepsilon} \right\rfloor$, as follows. First, suppose that all receivers have the same RTT, equal to RTT_{max}: $\forall R_i \in \{*\} \mid RTT_i = RTT_{max}$; second, the response quota per epoch must be the smallest possible: $RQ = 1$. Let *clock=begin*=0, $m = \left\lfloor \frac{RTT_{max}}{\varepsilon} \right\rfloor$. Suppose the Polling Table is empty when a polling request is to be planned to all receivers; the mechanism allocates the response of R_1 to E_m, and makes ARC$[m] \leftarrow 1$; allocates the response from R_2 to E_{m+1}, and makes ARC$[m+1] \leftarrow 1$, and so on, until the response of R_{GS} is allocated to epoch E_{m+GS-1}, and ARC$[m+GS-1] \leftarrow 1$.

At time 0, the sender transmits the poll to R_1 to elicit the response which was allocated to epoch E_m (i.e., expected to arrive in RTT_{max} time). The next poll planned for R_i, if happening before ε, will be assigned to E_{m+GS}, and ARC$[m+GS] \leftarrow 1$. The next polling, to R_2, will be sent only at time ε. A new polling to R_2 planned before $2 \times \varepsilon$ will occupy the entry ARC$[m+GS+1]$; as *clock* has advanced to ε, entry ARC[0] is not required anymore, so that only $GS + \left\lfloor \frac{RTT_{max}}{\varepsilon} \right\rfloor$ entries are kept.

Since most often RTT_{max} cannot be accurately predicted, the ARC vector dynamically increases in size with RTT_{max}; in such cases, the number of entries in ARC is incremented according to a step function.

Figure 4.1 illustrates the data structures used at the sender and receivers.

4.1.3 Threads

Each of the main tasks of the protocol at the sender is performed by an individual thread:

- the interface with the sending application;
- the transmission of packets to receivers;
- the handling of feedback packets;
- the processing of asynchronous events.

The receiver is very simple and contains a single task:

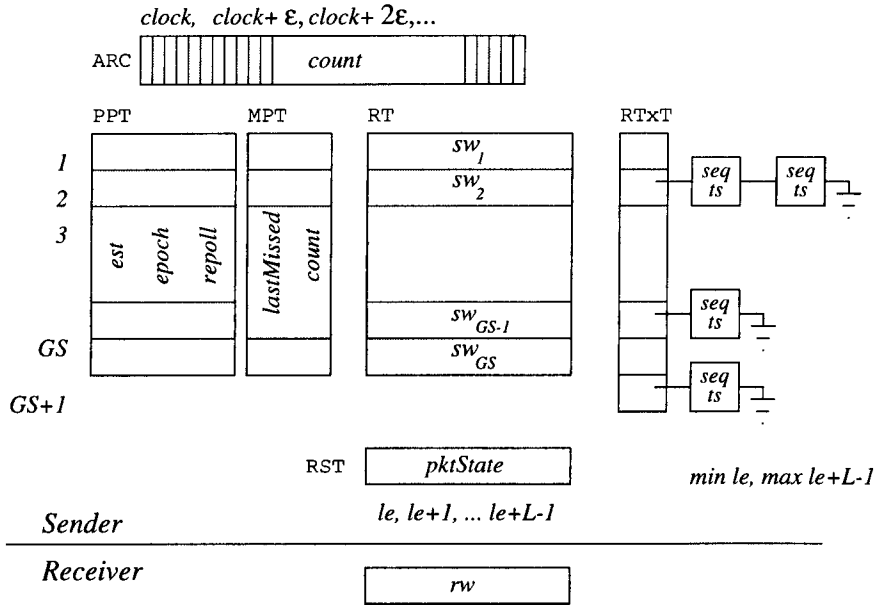


Figure 4.1: Structures used at sender and receivers.

- the reception of packets from the sender and transmission of feedback.

4.1.3.1 The interface with the sending application

The Generator Module's (`GM`) simple role is to receive downcalls from the sending application and process them. The production of data by the sending application is carried out through `write()` operations, and the `GM` is responsible for preparing such data for transmission. The steps followed by `GM` (until the sending application uses a `close()` to end the transmission) are summarized below:

(i) wait until data is made available by sending application;

(ii) wait until the flow control mechanism allows the transmission of a new data packet ($sw.aw > 0$);

(iii) increment $sw.hs$;

(iv) assemble a `TxQ` entry with `tx.seq` = $sw.hs$, `tx.dest` = $\{*\}$, and `tx.type=DATA_ORG`;

(v) enqueue the new entry in `TxQ`;

(vi) schedule the thread responsible for consuming `TxQ` entries, `TxM` (see below).

4.1.3.2 The transmission of packets to receivers

All entries produced in TxQ are consumed by the Transmitter Module (TxM). An entry produced by GM will result into the multicast transmission of a DATA(POLL) packet. While TxQ is non-empty, TxM executes the following sequence of actions:

(i) make sure at least one IPG has elapsed since the last transmission, blocking until the time of the next transmission;

(ii) consume the head packet from TxQ;

(iii) if tx.seq> 0, plan the polling for each receiver which is to receive this packet (given by tx.dest) and does not have a planned polling in PPT;

(iv) generate a polling set *polled* with due polls according to PPT and piggyback *polled* into the packet if $polled \neq \{\}$; if $polled = \{\}$ and $seq = 0$, tx is discarded;

(v) merge (union) (DATA)POLL.*polled* to tx.dest;

(vi) assign the appropriate type to the packet according to *polled* and *seq*;

(vii) check the number of receivers in tx.dest, and transmit the packet either through multicast or selective unicasts (without delay) by comparing tx.dest size with $MTR \times GS$; if doing multiple unicast transmissions of the same packet (data retransmission or POLL packets), these are done without any delay, regardless of the cardinality of tx.dest;

(viii) if tx.type=DATA_RETX add or update tuple(s) of RTxT to record retransmission time(s);

(ix) if the packet is (DATA)POLL, determine the RTO_{polled} for the polling request according to the set of receivers in *polled*;

(x) build and enqueue in ToQ a to entry with: to.ev_time$\leftarrow RTO_{polled}$, to.type$\leftarrow$RTO, to.to_resp$\leftarrow$ *polled*, to.hs$\leftarrow$ *sw.hs*, and to.ts$\leftarrow$ *clock*.

When TxM exits the loop, it checks if there is a planned poll in PPT; if so, TxM builds and enqueues an entry in ToQ with type POLL_GAP.

4.1.3.3 The processing of asynchronous events such as timeouts

The sender needs to limit the waiting for responses from receivers by setting a timer; the Event Manager (EM) is the thread responsible for managing the asynchronous events according to the contents of ToQ. While the ToQ is not empty, the steps executed by EM are:

(i) if the event time of the element at head of ToQ has *not* expired (i.e., to.ev_time> *clock*), set the system alarm to to.ev_time, and then block;

  (a) when woken up, re-read the element at the head of the ToQ, if any;

(ii) else, remove the expired event from ToQ, and process the expired event according to its type:

  (a) to.type=POLL_GAP: build and enqueue a POLL_GAP entry in TxQ, and schedule TxM to process it;

  (b) to.type=RTO:

    (b1) update MPT with to.hs for all $R_i \in$to.to_resp;

    (b2) plan a repoll in PPT for each $R_i \in$to.to_resp;

    (b3) if any repoll plan has been now included in PPT, schedule TxM to deal with it;

    (b4) for every *seq* in *sw.nea* $\leq$ *seq* $\leq$to.hs, if RST[*seq*] = *inCollection*, verify if condition (**ucast retx**) has been satisfied (see algorithm in Figure 3.18). If so, enqueue entry of type DATA_RETX in TxQ, and schedule TxM to consume it.

Note that when or if an alarm expires, EM keeps removing and processing all to entries as long as to.ev_time$\leq$ *clock*. If ever ToQ becomes empty, the EM does not set the system timer and then blocks waiting for a new entry to be enqueued in ToQ.

4.1.3.4 The handling of polling responses

Receivers will return responses to polling requests sent by TxM. The Response Handler Module (RHM) is the thread that handles all the feedback sent by receivers and coordinates the loss

detection and recovery process. The cycle executed by RHM is:

(i) wait blocked until an incoming RESP packet is available;

(ii) receive the arrived RESP packet (from a given R_i);

(iii) scan the ToQ and remove R_i from all to entries which causally precede the response, removing any entries left with to.to_resp= {} (and schedule EM if the head of ToQ is removed);

(iv) update MPT if required (if R_i was recorded in MPT as in *repoll*);

(v) execute error control and recovery actions by updating sw_i window in RT (using the procedure described in Figure 3.17); for all packets with *seq* such that $sw_i.le \leq seq \leq rw.hr$, check error control, if necessary, updating RST; identify whether a retransmission is required; if so, prepare and enqueue a tx entry in TxQ with tx.type=DATA_RETX, tx.seq=*seq*, and tx.dest = *Nacked*(*seq*), and schedule TxM to consume tx.

4.1.3.5 The reception of packets from the sender and transmission of feedback

At each receiver, there is a single thread, the Receiver Module (RM), which executes the following loop at R_i:

(i) block and wait for data or control packets coming from the sender;

(ii) take an arriving packet;

(iii) if required, update rw_i (as described in Section 3.2.1);

(iv) if a (DATA)POLL packet, check if this receiver is being requested to send a response to the poll; if so, transmit a RESP packet to the sender.

4.1.4 Overall structure

The overall structure is illustrated in Figure 4.2. The GM thread interacts with the sending application to obtain data for transmission; such data is assembled in a data unit, and stored

in a packet buffer. After producing an entry in TxQ, GM schedules TxM to run so that the latter can consume the new entry in TxQ. When TxM runs, it waits until the next transmission time (because of IPG); then it consumes the entry in TxQ, adds a plan for receivers that are not yet present in PPT, generates a polling set with all due polls, multicasts a data packet to the receivers, and finally enqueues an RTO entry in ToQ if the data packet sent contained a polling request. When (if) the TxQ becomes empty, and there are receivers to be polled, TxM creates a poll gap and determines the time it will end, and then enqueues a POLL_GAP event into ToQ. EM is scheduled by TxM if such event was added as head of the ToQ.

Receivers (RMs) which get the packet transmitted by TxM update their receiving window. If the packet contains a polling request (i.e., (DATA)POLL packet), the subset of RMs which are nominated by the polling request transmit a RESP packet to the sender. RMs deliver "consumable" data to their local receiving application as requested, and update the receiving window whenever data is consumed.

The RESP packets transmitted by RMs arrive at the sender; an arriving RESP is received by RHM, and the RTO events in ToQ are immediately updated. If, while removing a receiver from an entry to of type RTO in ToQ, the to.to_resp becomes empty, the entry is removed from ToQ; if the removed entry was at the head of ToQ, then EM is scheduled to reprogram the system timer. RHM updates the sending window corresponding to the RM which originated the RESP. This updating may affect aggregated attributes of the global sending window; if so, it may result in the end of collection period of certain packets and thus lead to their retransmission. If RHM is to retransmit one or more packets, it will enqueue one or more entries (one per different data unit) into TxQ, and then schedule TxM so that it can consume them. After RHM has updated the sending window, it may schedule GM to get new data from the sending application if the window has opened ($sw.aw$ increased from 0).

If one or more RESP packets do not arrive (in time), an RTO entry in ToQ will expire. EM will run to process such an event, and will re-evaluate any packets in the range affected by timeout and that are in collection; as a result, it is possible that some packets are to be retransmitted. If so, the EM will enqueue in TxQ one or more entries and schedule TxM to consume them, so that one or more packets will be retransmitted.

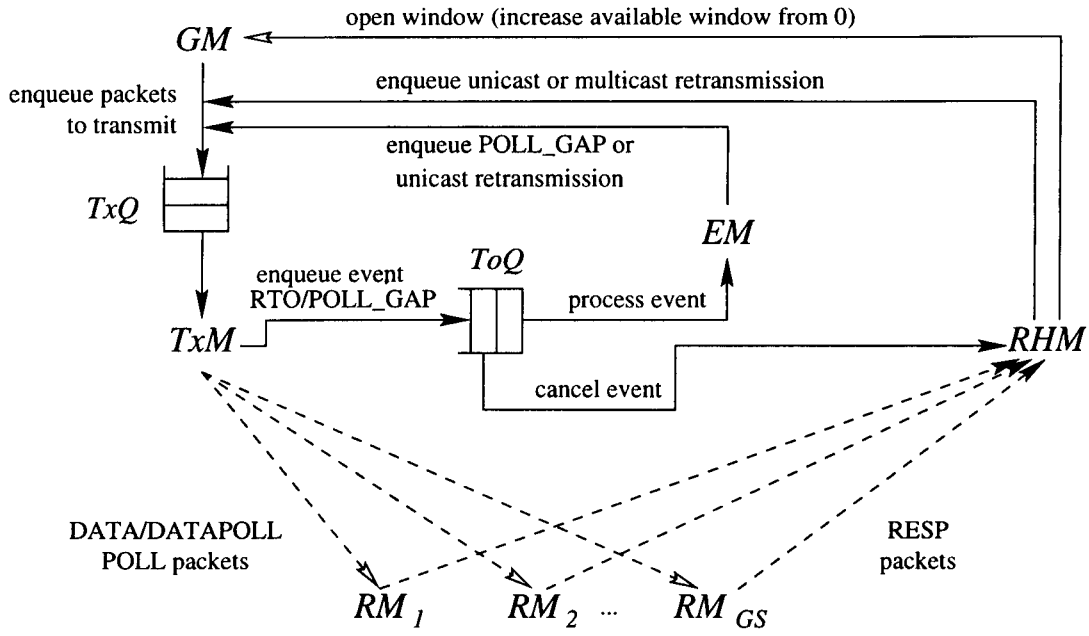


Figure 4.2: Overall structure of the protocol machine.

4.2 Simulation

This section describes the experiments carried out to evaluate the prototype implementation of flat PRMP. To perform a comparative analysis, the Full Feedback protocol described in Section 2.3.1 was implemented and tested under the same conditions as PRMP. This protocol was used by [Pingali94] to show the poor scalability of sender-initiated protocols in comparison with receiver-initiated ones. For this reason, the Full Feedback is also used in this thesis to assess PRMP's scalability.

4.2.1 Simplified network model

The network was modelled as a set of channels directly connecting sender and receivers, as shown in Figure 4.3. Three basic kinds of channels were defined: *short*, *medium*, and *long* (haul). Each kind of channel is characterized by a set of three attributes:

lat propagation latency (mean);

jit jittering (propagation latency standard deviation); and

err percentage error rate (of packets which are lost, apart from implosion).

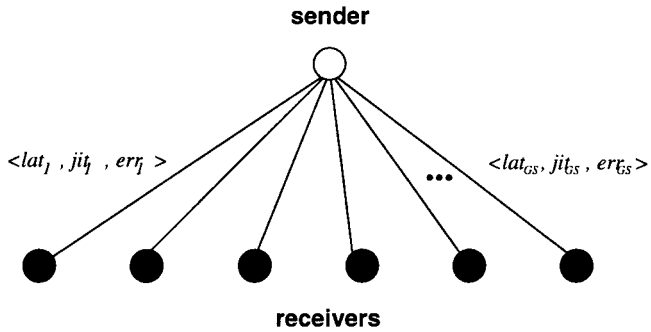


Figure 4.3: Network model employed in simulation experiments.

Each channel is completely independent of other channels. The values which were associated with each type are listed in table 4.1. Using a combination of these channels, two simple network configurations were defined:

`LOCAL` all GS channels are of type *short*; and

`WIDE` one third of GS channels is each of kind, *short, medium,* and *long* ($GS \geq 3$ in this configuration).

Channel Type	*lat*	*jit*	*err*
short	1.5 ms	0.08	1%
medium	5 ms	0.5	1%
long	75 ms	15	10%

Table 4.1: General properties assumed for kinds of channels.

It is not claimed that the defined channel types or network configurations are "typical"; it is very difficult, if at all possible, to specify a "general" configuration which is largely representative of existing networks. In particular, the loss rates used for *short* channels in the experiments were set higher than usually observed; the idea was to increase the packet loss rates so that the impact of loss detection and recovery mechanisms in PRMP could be better evaluated.

The implosion losses were simulated in the following manner: packets which arrive at the sender's host join a limited-size queue (denoted as "`INCOMING`"). If the queue has already reached its full capacity, then the incoming packet is dropped and counted as an implosion loss. Packets are consumed from the head of the `INCOMING` queue by the sender at a rate

equal to $\frac{1}{ITR}$ (recall that ITR is the implosion threshold rate and represents the capacity of the sending host in processing feedback). In the set of experiments reported in this section, INCOMING size was 16 packets, and $ITR = 1,500$ packets/s. ITR is expected to vary depending on hardware and network used; the value of ITR was taken from [Holbrook95], which indicates that an IBM-R6000 is capable of processing 1,587 ACKs per second.

4.2.2 Metrics

Three metrics were used in the evaluation process:

T throughput;

N network cost; and

I the number of implosion losses.

If there are DP data units of $unitSize$ (in bytes) to be reliably transmitted, and $\triangle t$ is the period of time (in seconds) between the transmission of the first DP data packets and the moment all packets become fully acknowledged (both events occurring at the sender), the throughput is calculated in *Kbps* as

$$T = \frac{DP \times unitSize \times 8 \times 10^{-3}}{\triangle t}$$

The maximum theoretical (thus best) value for T is close to the maximum transmission rate specified; the throughput is restricted by the propagation delays between sender and receivers as well as packet losses and (window-based) flow control. Because sender-reliability is required, the transmission only ends when the sender obtains confirmation that all packets have been successfully received at all receivers. Thus, with $err = 0$, infinite window and infinite ITR, the maximum throughput achievable is (note that IPG and RTT_{max} are measured in ms):

$$T_{opt} = \frac{DP \times unitSize \times 8}{DP \times IPG + RTT_{max}}$$

In other words, $\triangle t$ will be the sum of DP IPGs required to transmit all packets, and the waiting for the feedback from the farthest receiver to arrive after the DP-th packet has been

transmitted.

N is calculated as the total number of packets exchanged per receiver per data unit. N derives from the number of packets exchanged between sender and receivers, but is relative to the amount of data to be transmitted and the group size:

$$N = \frac{totPkts}{DP \times GS}$$

where $totPkts$ represents the total number of packets exchanged (multicast transmissions correspond to GS packets, and lost packets count as well). The ideal value for N is

$$N_{opt} = \frac{DP + 1}{DP}$$

which corresponds to DP multicast transmissions plus one feedback packet per receiver to acknowledge the DP packets. Clearly N_{opt} should converge to 1 with increase in DP.

I is simply the total number of implosion losses recorded. The best value for I corresponds to the absence of packet losses inflicted by implosion, that is

$$I_{opt} = 0$$

Both T and N will be affected by I: a non-nil value for I represents (packet) losses that need to be recovered, consuming both time (thus decreasing T) and bandwidth (thus increasing N).

Unless otherwise noted, in all simulation runs it is assumed:

(a) ready supply of data by the sending application;

(b) "hungry" receiving applications, with instantaneous consumption of consumable data;

(c) the set of latencies (lat) is modelled using values randomly generated according to the *Normal distribution* [Mitrani82];

(d) packet losses due to causes other than implosion are modelled by a statistical draw using *err*; therefore, losses are assumed to be *independent*;

(e) the probability of loss *err* refers to packets, and is therefore applied equally to data and control packets (despite their different size);

(f) data unit size $unitSize = 1,000$ bytes and $DP = 1,000$ packets;

(g) a top transmission rate of 1,000 packets/s (i.e., $IPG = 1$ms);

And, in particular, for PRMP runs:

(h) response rate equal to the maximum capacity (i.e., $RR = ITR$);

(i) epoch length $\varepsilon = 10$ms;

(j) $MTR = 20\%$. A lower MTR value would increase N, and also T; a small MTR was chosen with network cost in mind.

4.2.3 The Full Feedback Protocol

Recall from Section 2.3.1 that in the Full Feedback protocol:

- there is a sliding window scheme with selective retransmission (i.e., no go-back-N);
- receivers return an ACK packet for every data packet received;
- loss detection is timeout-based (with $RTO = \max\{RTO_i\}$, $RTO_i \leftarrow \beta \times RTT_i$, and $\beta = 2$);
- recovery is done immediately and via global retransmissions.

The protocol implementation used in the evaluation adds an IPG to limit the maximum transmission rate. This IPG is required because the time taken to transmit a packet was not included in the simulation model (under the assumption that a transmission would take less than 1 IPG); therefore, without an IPG separation between transmissions, the sender would be able to transmit L packets in 0 time. It is assumed that the time to send a packet is less than IPG. The protocol is summarized below.

Receivers are simple; they each return an ACK packet to every data packet received. The sender continuously executes the following steps, depending on the availability of feedback, expiration of retransmission timeout events, and the time allowed for the next transmission:

- request (to the underlying network) to receive a feedback packet from any of the receivers;
  - if there is one available now, receive it;
  - else, leave the request pending (the sender runs when feedback can be received);
- check for retransmission timeouts and, if the earliest of the timeouts (lowest in value) has expired, retransmit the corresponding packet if the last transmission was more than 1 IPG ago;
- if the available window is greater than zero, and the time of the most recent (re)transmission was more than 1 IPG ago, multicast a new packet.

4.2.4 Protocol runs

The protocol runs using the Full Feedback protocol shown above are denoted as "FF". The runs using the flat version of PRMP are denoted as "PF".

Because of the well-known impact of the window length (to "keep the pipe full") on throughput, both PF and FF were additionally tested considering a window of the same length of the transmission, that is, $L = DP$. Note that this corresponds to infinite buffers and is the standard assumption for several reliable multicast protocols (e.g., RMTP [Lin96], MFTP [Miller97], and LBRM [Holbrook95]). For simulations with the window length L set to DP, the polling feedback and full feedback protocols are respectively denoted as "PF-IW" and "FF-IW" ("IW" stands for infinite window, or infinite buffers).

Furthermore, to assess the impact of implosion losses on the Full Feedback protocol, the FF-IW protocol was run with the sender's host having an infinite implosion threshold rate ($ITR = \infty$); this "golden scenario" with infinite buffer and suppressed implosion losses is called "FF-IW-IT". Table 4.2 summarizes the protocol runs.

Run Name	*Window Length* (L)	*Implosion Threshold* (ITR)
PF	64	1,500
PF-IW	1,000	1,500
FF	64	1,500
FF-IW	1,000	1,500
FF-IW-IT	1,000	∞

Table 4.2: Protocol runs.

4.2.5 LOCAL configuration

Recall that in the LOCAL configuration the sender is at the same, short distance from all GS receivers, and is connected to them through a *short* channel.

Figure 4.4 shows the scalability of the five protocols in terms of T, for group sizes between 1 and 60 receivers. First note that the graphs for FF and FF-IW are *identical*; this means that the window did not have any impact on throughput of the full feedback protocols. This is because, given the low RTTs of *short* channels and the transmission rate, the chosen L is sufficient to allow continuous transmission (that is, $L \times IPG > RTT_{max}$). When there is a loss, the sender needs to detect it before it blocks the progress of the window: L must be sufficiently large to allow the continuous transmission of packets up to the RTO expiration. As the RTO is determined as twice the maximum RTT ($\beta = 2$), continuous transmission occurs if $L \times IPG > 2 \times RTT$. This is certainly the case in the LOCAL configuration.

It is clear from the graph that the throughput of FF and FF-IW degrades quickly as GS increases. The reasons are:

- *"cumulative loss"*: the probability of a given multicast packet not reaching at least one receiver (or feedback packets that it may generate not reaching the sender) increases with GS (as described in Section 2.2.1); and
- *implosion*: without an implosion avoidance scheme, implosion losses per packet transmitted increase with GS.

Both cumulative loss and implosion tend to increase the time it takes to get a packet fully acknowledged. The graph for FF-IW-IT (which has suppressed implosion losses) indicates that the poor scaleability of FF/FF-IW is mainly attributed to implosion losses: the decrease in T

for FF-IW-IT can only be due to cumulative loss, while the difference between FF-IW-IT and FF-IW is only due to implosion.

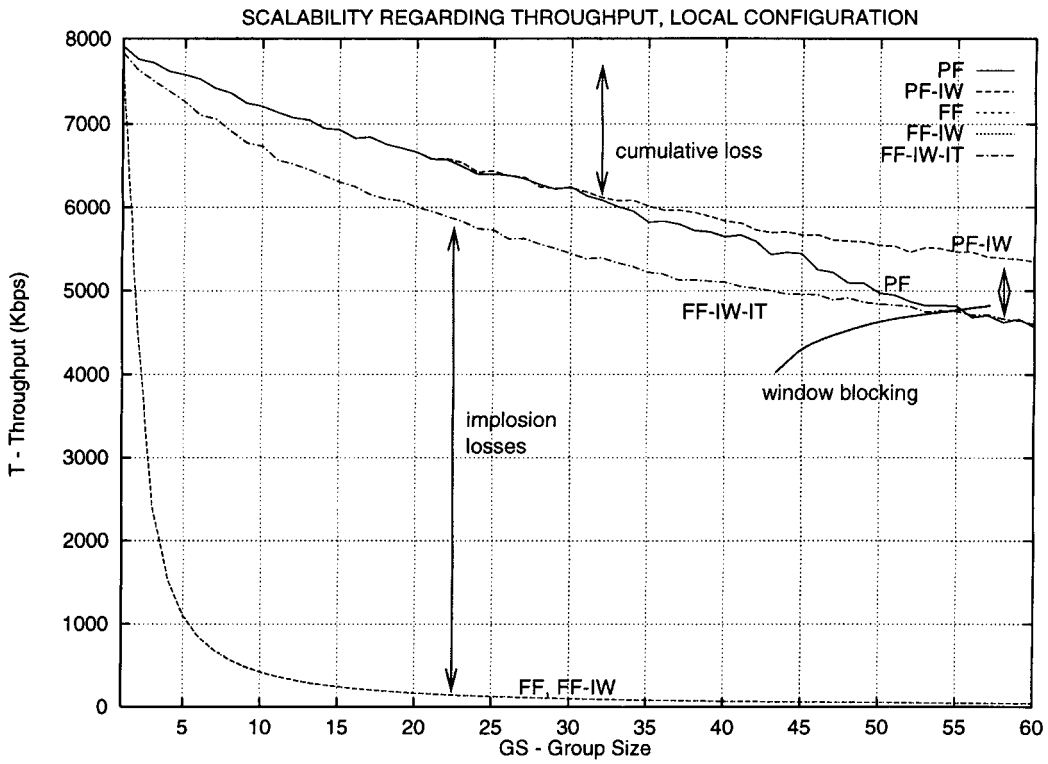


Figure 4.4: Throughput (T, in Kbps) in the LOCAL configuration.

The beneficial effect of the polling feedback mechanism in containing implosion losses can be seen by comparing PF-IW with FF-IW-IT; the T achieved in PF-IW is higher than that of FF-IW-IT and the difference becomes more or less uniform for $GS > 15$. This gain in T can be attributed to the error control of PF protocols, which allows polled receivers to *negatively acknowledge* packets. In contrast, the FF protocol relies on the absence of ACKs for loss detection; the sender can detect a packet loss only at the expire of the RTO[1]. Though it makes PRMP more complex, the use of NACKs can help achieve faster error detection and recovery and thus better throughput.

The implosion avoidance mechanism employed by PRMP restricts the flow of feedback from receivers, with two implications. First, with reduced feedback, a receiver will be polled less frequently. As receivers can only send NACKs when polled, on average, the period between the

[1]in this simulation, both PRMP and Full Feedback protocols calculate RTO as twice the largest estimated RTT_i, for all R_i from which responses are expected.

detection of the loss at the *receiver* and the sending of the NACK may take longer. So, despite the fact that NACKs are collected at the sender, this may affect the overall T. Second, the (limited) amount of feedback that is available to the sender may be insufficient to keep sw open. If sw closes, the sender is temporarily prevented from transmitting new data packets, affecting throughput.

The T of PF in Figure 4.4 is the same as that of PF-IW up to a point ($GS = 30$); thereafter it starts decreasing because of

- *window blocking*: to transmit new data packets, the sw needs to slide, which only happens when the sender has learned that receivers have slid their window. Several factors may delay the sw advance: frequent packet losses (high err and implosion), long RTTs (high lat), and limited feedback (low RR).

The effect of window blocking is marked in Figure 4.4 by the (relatively small) difference between the PF and PF-IW graphs, which starts at $GS = 30$. The primary cause for this window blocking with large GS values is the cumulative loss at the sender. Secondly, the increased chance of loss of polling requests and responses, which may delay the sender until the expire of an RTO.

The throughput gain achieved by PF and PF-IW is *not* at the expense of increased network cost, as illustrated in Figure 4.5. It shows the scalability of the protocols regarding the network cost N (shown in log scale on the y axis) for group sizes between 1 and 60 (on the x axis). Here again the graphs for FF and FF-IW are identical, indicating that the window length had no impact for the Full Feedback protocols. The harmful effect of implosion on N can be measured from the gap between FF-IW and FF-IW-IT.

The effect of cumulative loss increases the network cost of a transmission: the more retransmissions required, the stronger the negative effect on N. The actual increase depends on how a protocol deals with losses. In the Full Feedback case, the increase is substantial because all retransmissions are via multicast. This is shown by the graph of FF-IW-IT in Figure 4.5: with 1 receiver, the cost is approximately 2, while $N \cong 3.5$ for $GS = 60$. In contrast, cumulative loss has little impact on PRMP because the mechanism waits and collects retransmission requests in RESP packets before deciding which retransmission scheme, multicast or selective unicasts,

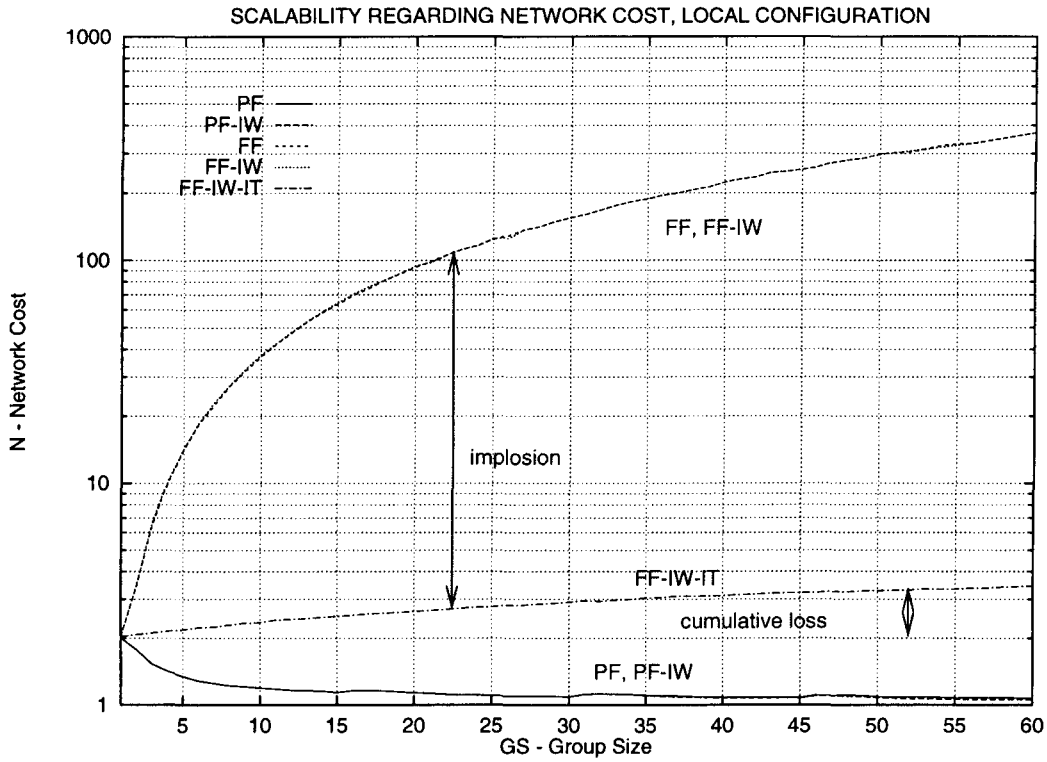


Figure 4.5: Network cost (N) in the LOCAL configuration.

can provide recovery with better cost. Also because PRMP's scheme is optimistic, as it does not interpret the absence of response as an implicit NACK for those packets that would have been referenced in the missing response. Finally, the polling scheme reduces the amount of feedback from receivers, and thus decreases the total number of packets ($totPkts$). The resulting N of PF/PF-IW is very close to N_{opt} for $GS \geq 10$, and hence shows that PRMP can deliver cost-effective multicast.

The above findings are corroborated by Figure 4.6, which shows the scalability of the protocols in terms of number of implosion losses. The graph of FF-IW-IT is omitted because in the latter implosion losses are suppressed with $ITR = \infty$. The number of implosion losses of FF/FF-IW protocols grows exponentially with GS because: (a) the number of implosion losses per (re)transmission grows with GS; and (b) the number of retransmissions required to get a packet fully acknowledged grows with GS.

Figure 4.6 shows that the window did not affect the implosion avoidance scheme of PRMP, which achieved near-optimal results ($I \leq 10$ for any GS value). The few implosion losses

which did occur were restricted to the beginning of the communication, in a period when the mechanism did not have RTT estimates available and thus could not plan the arrival of responses properly. These implosion losses occurred because there was an "overlapping" between planning using a large initial, default RTT, and the updated, smaller RTT after a response has been received. For $GS < 8$, the group size was not big enough to cause implosion losses despite the overlapping; for $GS \geq 30$, because it takes longer to poll the set of receivers, the overlapping did not occur.

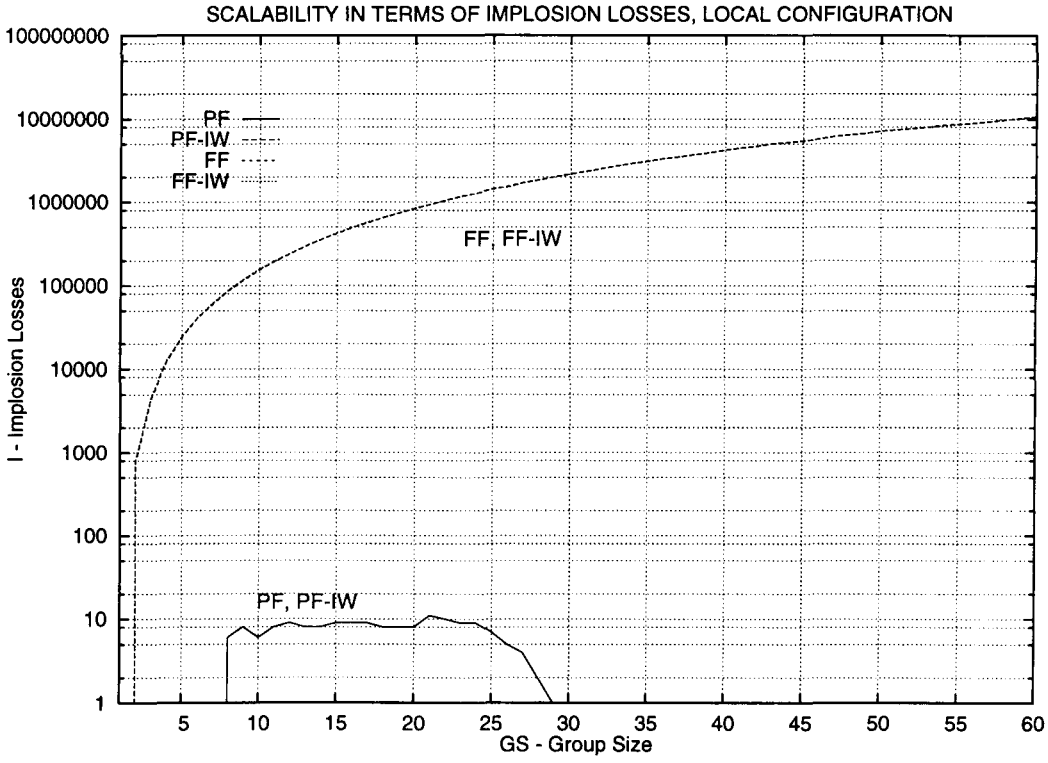


Figure 4.6: Number of implosion losses (I) in LOCAL configuration.

4.2.6 WIDE configuration

This section reports simulation results for experiments conducted with the WIDE configuration. This configuration differs from LOCAL in several aspects, in particular because 1/3 of the channels (of type *long)* have higher values for:

- *err*: effect of cumulative loss is stronger;
- *lat*: overall the window becomes more important;

- *jit:* RTT estimation becomes harder.

And finally, as the set of "distances" between sender and receivers is heterogeneous (as there are 1/3 of each kind of channel), feedback packets from a given data transmission (or poll request) are naturally spread in time, reducing the likelyhood of implosion losses.

The I values in the WIDE configuration are shown in Figure 4.7. Like in the LOCAL configuration, PF and PF-IW had few implosion losses. Note, however, that more losses were caused by PF than in its corresponding run in the LOCAL configuration: up to 61 RESP losses were recorded in PF runs. This increase in response losses is linked with the total number of RESP packets that are required throughout the transmission: a proportion of RESP packets do not arrive within the expected epoch and this may lead to losses; so, I is directly proportional to the *total number* of RESP packets that are exchanged during the communication. The communication with PF takes longer and the window blocks because of the high *lat* and *err* values; when the window blocks, instead of sitting idle, the sender will poll the receivers, increasing the number of RESP packets. Therefore, PF generates more RESP packets and, given a sufficiently large GS, it will have caused more implosions at the end of the transmission. For example, for $GS = 12$ and $GS = 36$, the average number of RESP packets required by PF/PF-IW was 5,409/3,040, and 8,698/3,558, respectively.

As shown in Figure 4.7, the window length significantly affected the results for the Full Feedback protocols: the sliding window restricts the effective data transmission rate, which in turns reduces the feedback rate, which in turn leads to fewer implosion losses. This is illustrated in Figure 4.7 by the gap between the graphs of FF and FF-IW, and is non-existent in Figure 4.6. Note that the gap between FF and FF-IW increases with GS (the y axis is in log scale); this is because the larger the group, the larger the impact of the smaller window holding new transmissions will be.

Figure 4.8 shows the throughput values for the five protocols in the WIDE configuration. Due to the infinite window, FF-IW performs better than FF for small GS values. This advantage, however, becomes a disadvantage as GS increases. Because of the resulting increased implosion losses, the performance of FF-IW rapidly becomes poor. The protocol FF provides consistently poor performance due to implosion losses and finite window. The impact of implosion losses

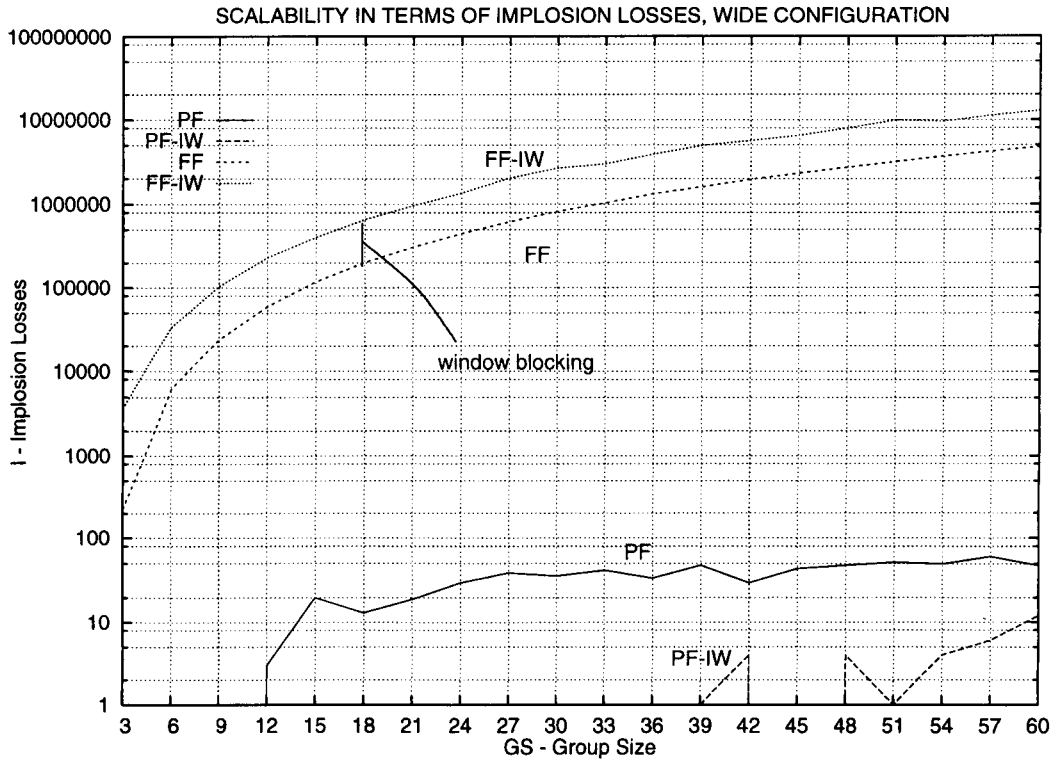


Figure 4.7: Implosion losses (I) in the WIDE configuration.

on T is shown by the gap between FF-IW and FF-IW-IT in Figure 4.8

As in the LOCAL configuration, for all group sizes, the performance of PF-IW is slightly better than the "golden scenario" portrayed by the FF-IW-IT graph, because of the error control mechanism of PRMP. PF performs substantially better than FF (and better than FF-IW-IT for $GS \geq 6$) because it prevents implosion losses, and because NACKs speed up loss detection. However, $L = 64$ is not sufficient to allow PF to keep continuous transmission. There are two reasons for this: (a) the transmission rate is high in comparison with the latencies of *long* channels ($IPG = \frac{lat}{75}$, that is, within 1 RTT 150 data packets could be transmitted); and (b) the 10% of loss employed with *long* channels.

Figure 4.9 shows the scalability of protocols with respect to the relative network cost N (with N in the y axis in log scale), for the WIDE configuration. As indicated by the gap between FF and FF-IW in Figure 4.7, the chosen window length $L = 64$ is small enough to limit implosion losses for FF and hence FF exhibits a lower cost than FF-IW. Observe that the difference in N for FF-IW and FF increases with GS, just like the differences in I increased in Figure 4.7.

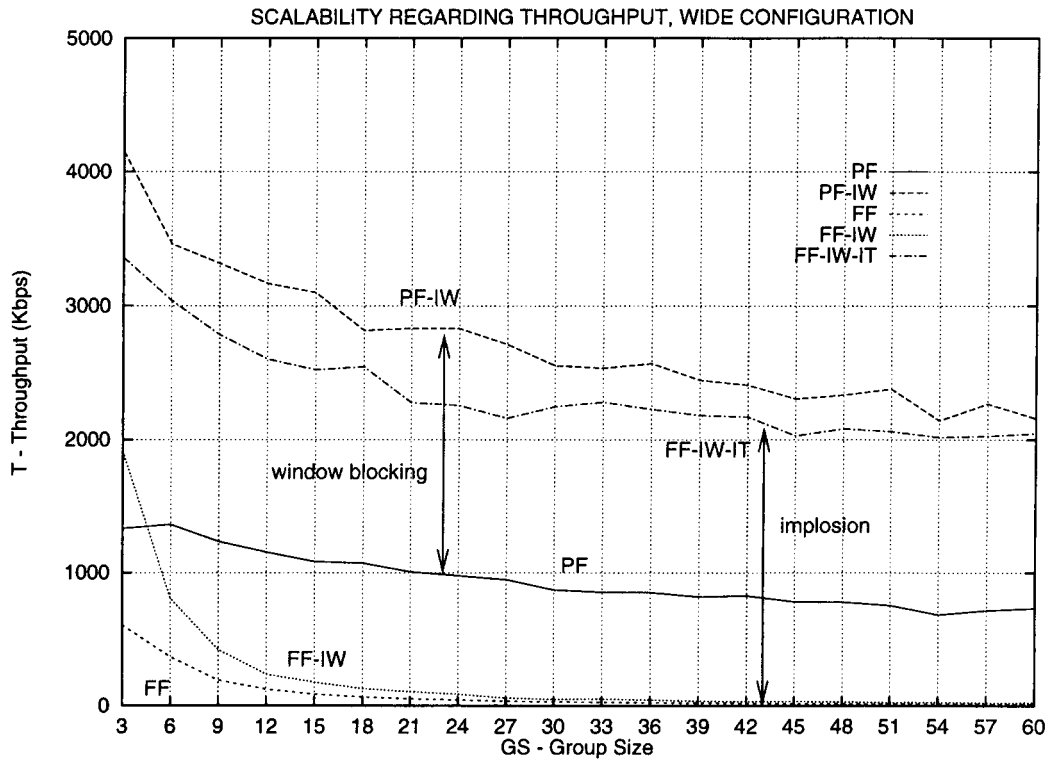


Figure 4.8: Throughput (T, in Kbps) in the WIDE configuration.

As mentioned above, in the WIDE configuration the *err* values are on average higher, and hence the effect of cumulative loss is stronger. Recall that the lost packets are counted in *totPkts*, and so increased number of losses will lead to increased N. Therefore, all graphs in the WIDE configuration (Figure 4.9) will present higher N than their corresponding graph in the LOCAL configuration (Figure 4.5). Thus the graphs for PF and PF-IW cannot be as close to 1 as in Figure 4.5, although PF and PF-IW graphs *do* converge to N_{opt} (i.e., 1). Likewise, the N of FF-IW-IT increases slightly faster in the WIDE than in LOCAL. Because of cumulative loss alone, with $GS = 60$, the sender of FF-IW-IT will exchange on average 4.5 packets with each receiver for every one of the DP data units to be transmitted. In contrast, the sender of PF-IW will need on average only 1.2 packets, only 26% of FF-IW-IT's network cost.

There is a noticeable difference in the network cost between PF-IW and PF for all values of GS. This is due to the number of polling requests that have to be sent using POLL control packets, i.e., the "POLL overhead": if the sending window closes too often, new data packets cannot be transmitted; since all receivers have to be polled at least once after the last data

packet of the window has been multicast, more polling requests will be sent through POLL packets (instead of DATAPOLL).

In PF-IW, the POLL overhead is not present because the sending window never closes, and hence poll requests are guaranteed to be piggybacked with DATA packets (except after all DP units have been sent). In PF, however, explicit POLL packets have to be transmitted when DATA cannot be sent due to a closed sending window. In the LOCAL configuration (Figure 4.5), the window was sufficiently large in comparison with IPG and short lat's to allow continuous transmission, and hence the effect of using POLLs was negligible.

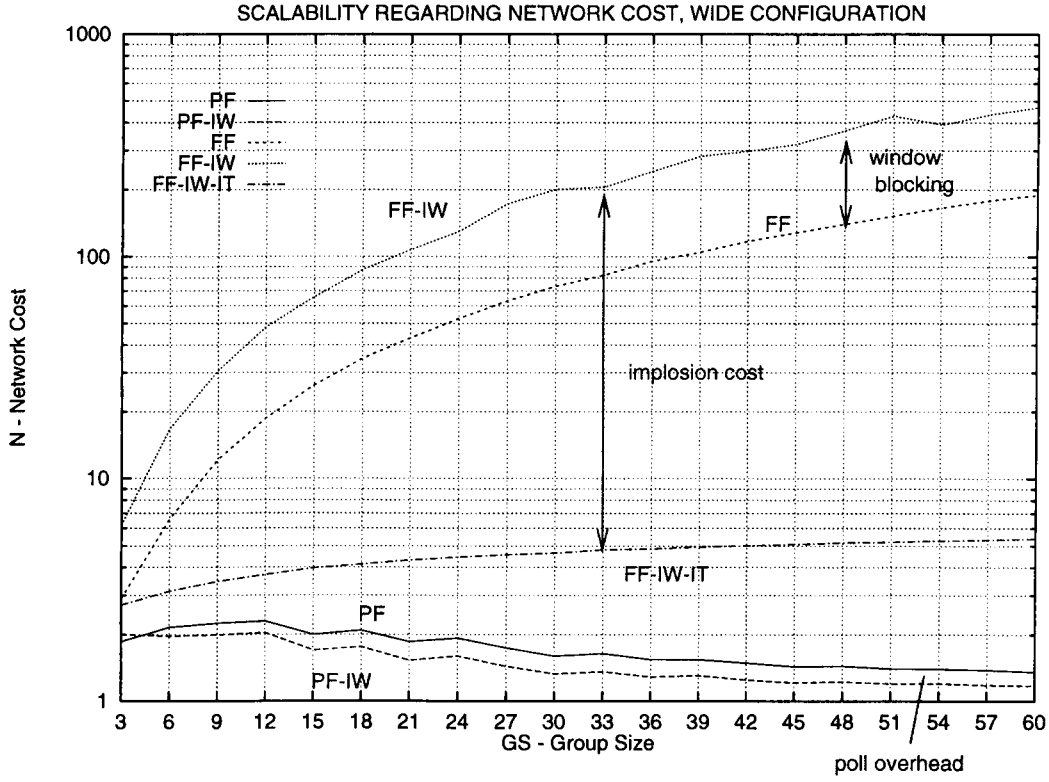


Figure 4.9: Relative network cost (N) in the WIDE configuration.

Finally, the impact of implosion in the network cost of FF protocols in the WIDE configuration is apparent in the gap between FF-IW-IT and FF-IW in the Figure 4.9; clearly, the predominant factor in the N cost of the Full Feedback protocols is I, i.e., the very large number of implosion losses which follows the increase in GS.

4.2.7 Impact of input variables

The simulation experiments reported in Sections 4.2.5 and 4.2.6 are based on assumptions (a) to (j) as listed in Section 4.2.2. These values can be regarded as "proper" input values, and some may represent the best choice for the tested configurations. This section addresses the impact of varying the input variables of PRMP. As shown in Figures 4.7, 4.8, and 4.9, the infinite window always had a positive impact (on I, T, and N, respectively). Or, put in a different way, the window blocking adversely affects the performance of PRMP. Thus, of special interest is the impact of the input variables in the window blocking, and this is discussed below.

lat The higher the propagation latencies, the longer it takes to get packets received and acknowledged, so that higher *lat* values increase the probability of the window blocking. (note: with larger *lat*, the "latency × bandwidth product" of the network increases, so that a larger L may be effective in increasing T).

L Clearly, the higher the L the lower is the probability that the window will block. Limiting factors to increasing L are the amount of memory available at the sender and receivers.

err Packets which are lost require a retransmission; this involves the sender detecting the loss and then repeating the process of transmission (while retransmitting occupies 1 IPG, it is necessary afterwards to poll a subset of receivers and wait for responses). The window cannot advance if a packet has not been fully acknowledged. For PRMP, higher error rates mean that the probability that either a polling request or a response is lost increases; the error control of PRMP is optimistic, saving unnecessary retransmissions, but also making PRMP more sensitive to the loss of polling requests or responses. Thus, higher *err* values will increase the chance of window blocking.

IPG The IPG determines the maximum transmission rate; a low IPG allows the transmission of a window of data packets in a short time. So, a larger IPG decreases the probability that the window will block, as by the time the last data packet is transmitted, all receivers had enough time to acknowledge the first packet of the

window. However, for PRMP, IPG also determines the frequency in which receivers are polled; a large IPG will lead to an increase in the average number of receivers polled with each (DATA)POLL, that is, increasing the granularity of polling. Therefore, the increase in IPG may help maintaining the window open, but will on the other hand negatively affect the implosion avoidance mechanism (increased delay (**d1**), as explained in Section 3.4).

EL Epoch length does not directly affect window blocking.

RR The lower the RR, the higher is the probability the window will block because all receivers need to be consulted before the window can slide (recall that $sw.le \leftarrow \min\{sw_i.le \mid R_i \in \{*\}\}$), and thus this takes longer with a lower RR.

GS The larger the group size, the longer it will take to obtain the feedback from all receivers, and hence the higher the probability the window will block.

To evaluate numerically the impact of the window length and the response rate on PRMP, experiments were performed varying:

- the window length (L), and
- the response rate (RR).

4.2.7.1 Window Length

This section describes experiments in which the PF and FF-IT (for Full Feedback with $ITR = \infty$) protocols were evaluated using different values of L. Three group sizes were tested, all in the WIDE configuration. PF runs are denoted as PF-10, PF-30, and PF-60, for 10, 30, and 60 receivers, respectively. FF-IT runs are denoted likewise: FF-IT-10, FF-IT-30, FF-IT-60. Table 4.10 lists the runs. The values for T, N, and I were measured for L varying between 1 and DP (which corresponds to an infinite window).

Figures 4.11.(a), (b), and (c) show the impact of the window length L on the T of PF and FF-IT protocol runs. The "knee" of each graph is marked with a circle and represents the best choice for L ("the smallest L such that the highest T is achieved"). It is clear from the graphs

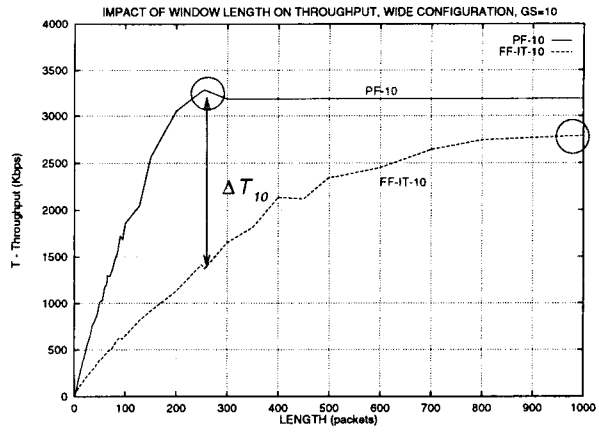
run name	protocol	GS	ITR
PF-10	PRMP	10	1,500
PF-30	PRMP	30	1,500
PF-60	PRMP	60	1,500
FF-IT-10	Full Feedback	10	∞
FF-IT-30	Full Feedback	30	∞
FF-IT-60	Full Feedback	60	∞

Figure 4.10: List of protocol runs for window length experiments.

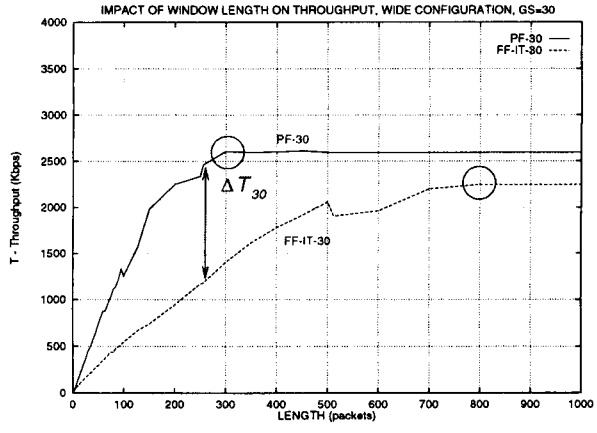
that PRMP requires a substantially smaller window length than the "golden scenario" runs of FF-IT for all three group sizes tested. In other words, PRMP outperforms the golden scenario runs in all group sizes for any window length; the advantage in T of PRMP over the FF-IT runs is marked in the figures as "$\triangle T_{GS}$". In all group sizes, the PF run reached its peak T at around the same window length, 260 packets, while the FF-IT runs reached their best T much later (that is, required larger L values). Note that the advantage of PRMP reached its maximum in this same L range, between 280 and 310: $\max \triangle T_{10} \cong 1,750$ at $L = 280$, $\max \triangle T_{30} \cong 1,250$ at $L = 280$, $\max \triangle T_{60} \cong 1,000$ at $L = 310$. These values show clearly that, for a given L, the advantage of PF over FF-IT reduces with the increase in group size. This is because the T of PF run uses a limited feedback rate (RR) to recover from increasing losses, whereas in the FF-IT runs implosion was suppressed ($ITR = \infty$) and large amounts of feedback were promptly available.

The effect of L seen in T does not appear on the same scale for the network cost or implosion losses. Figure 4.12 shows the impact of L on N; for the FF-IT runs, it made virtually no difference. In the PF runs, however, the cost dropped when increasing the L to 100, because the window blocked less often and thus the overhead stemming from POLL packets was lower. The N value for all PF runs was smaller than the corresponding FF-IT run, and was best with $GS = 60$.

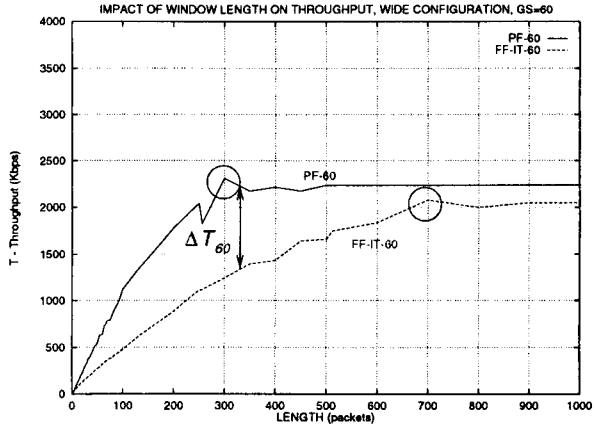
Figure 4.13 shows the impact of varying the window length on the number of implosion losses recorded in PF runs; note that FF-IT runs have $ITR = \infty$ and hence have $I = 0$. The figure shows a high number of implosion losses for the PF-60 run when small L values ($L \leq 10$) are used. The reason for such an unusually high I for PF is the increased number of responses that are required during the communication, as discussed when addressing the implosion losses



(a) group with 10 receivers



(b) group with 30 receivers



(c) group with 60 receivers

Figure 4.11: Impact of window length on the throughput (T) of PF and FF_IT runs for group sizes 10, 30, and 60.

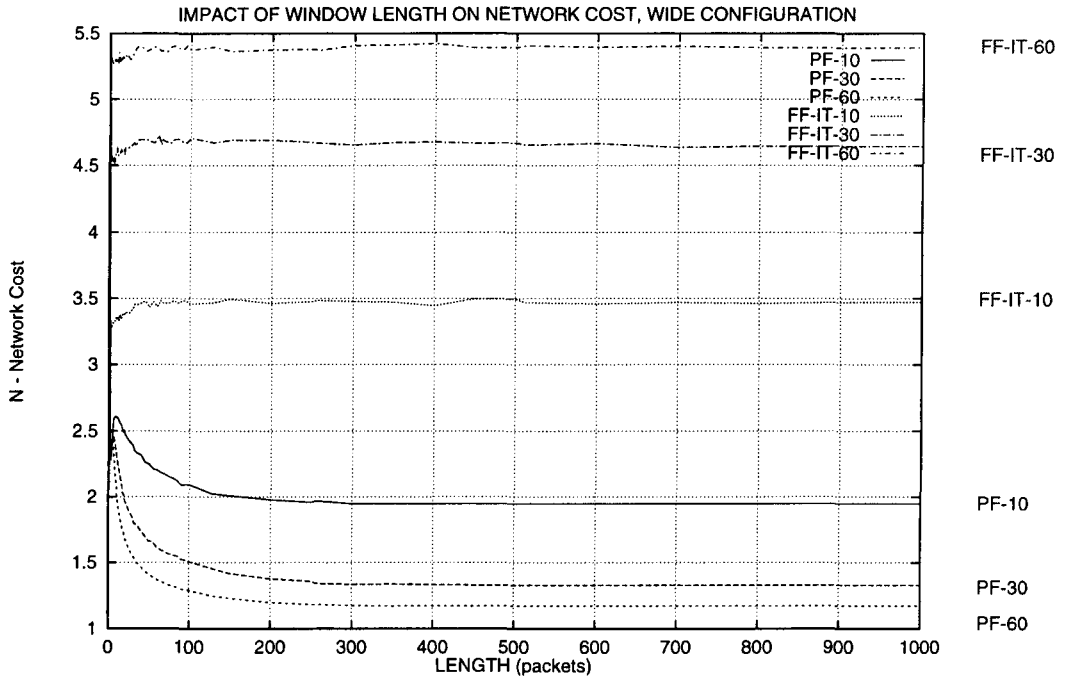


Figure 4.12: Impact of window length in the network cost (N) of PF and FF_IT runs for group sizes 10, 30, and 60.

in the WIDE configuration (see Figure 4.7). With a sufficiently large group and a small window, the high latencies and error rates will make the sending window block more often; if the window blocks and opens regularly, more polling requests are sent than in a continuous transmission; with more RESP packets elicited from receivers, the total number of implosion losses at the end of the communication will be higher. In the extreme, for $L = 1$, after every data packet transmission, all GS receivers will have to be polled through one or more POLL packets before the next data packet can be transmitted. For PF-30, there are also implosion losses, but fewer. Losses are not recorded with PF-30 for $L < 3$ because the *actual* response rate is limited by the window blocking (as with FF).

4.2.7.2 Response Rate

To prevent implosion, PRMP plans the polling of receivers so that the amount of feedback arriving at the sender does not exceed RR RESP packets per second. RR is an important input value of the protocol: intuitively, RR should match or be slightly lower than the bottleneck capacity, represented by ITR. In practice, implosion losses can occur in the network at any

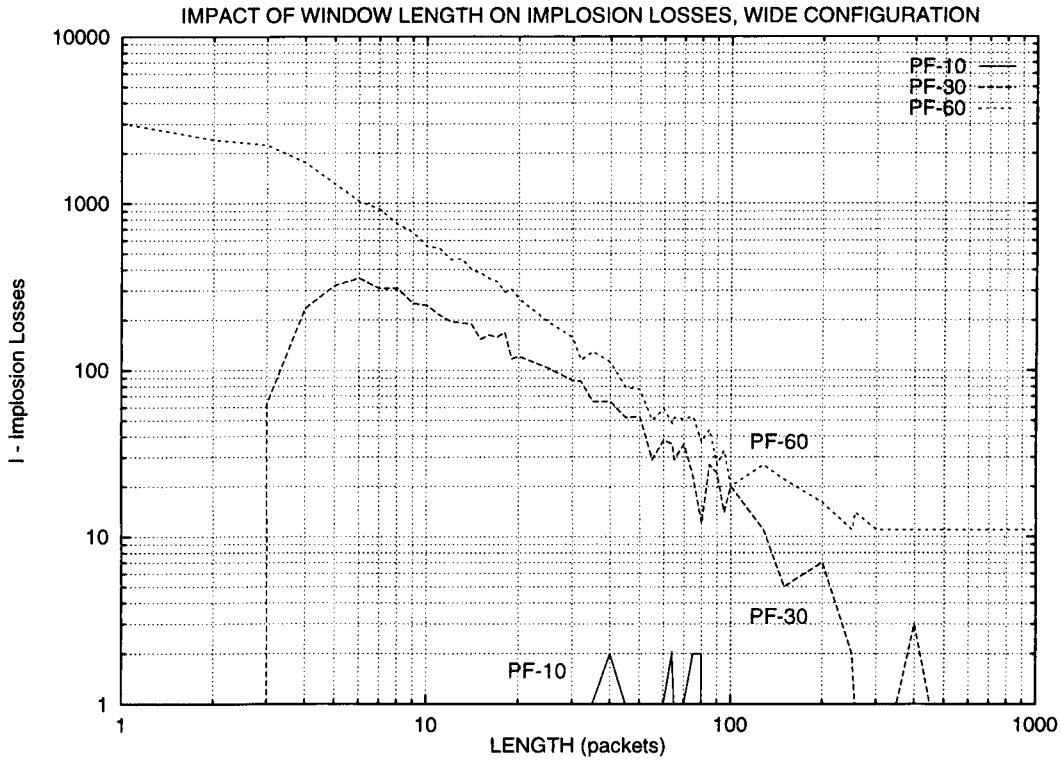


Figure 4.13: Impact of window length in the implosion losses (I) of PF for group sizes 10, 30, and 60 (FF-IT runs have suppressed implosion).

point between the sender and receivers, and are not restricted to the sending host. In the experiments performed, however, implosion losses were limited to the INCOMING queue at the sender. The number of implosion losses will thus be determined by the difference between RR and ITR, as well as how uniformly RESP packets arrive (the smaller the INCOMING queue, the more sensitive the sending host will be with respect to peaks in the actual feedback arrival rate). Smaller RRs will reduce the risk of implosion.

On the other hand, the protocol mechanism depends on having a certain amount of feedback, so that the state in *sw* is kept "fresh" with respect to receivers. Very low RRs may cause the sending window to block more often (in particular with higher loss rates), due to the wait for needed feedback to arrive.

The simulation results shown up to now refer to experiments performed using RR set to ITR. The impact of the RR value for PRMP was evaluated by testing 3 representative group sizes (10, 30, and 60) in the LOCAL configuration, measuring the values of T, N, and I.

Increasing RR has two major effects:

(**fresh state**) poll requests to any given receiver are sent more frequently; hence a packet is likely to be referenced sooner (less time to detect losses and less time to get full acknowledgment);

(**feed excess**) However, if RR exceeds ITR, response packets are lost due to implosion, delaying loss recovery and packet acknowledgement.

Note that (**fresh state**) and (**feed excess**) compete in promoting and suppressing the gain in loss detection time allowed by negative acknowledgments.

Figure 4.14 shows the impact of RR on T. First, note that the runs with infinite window, PF-IW-10, PF-IW-30, and PF-IW-60, were only mildly affected by RR's value: any $RR \geq 500$ ($\frac{ITR}{3}$) was sufficient to achieve the maximum T. This is because, with infinite window, the sender can always transmit a packet every IPG, either a new transmission or a retransmission. *After* the last packet is transmitted, the RR becomes important, as it is necessary to poll at least once each receiver. This arguments applies to PF-10 as well: with $L = 64$, low RTT_{max}, and low cumulative loss, the group size was so small that even a small RR was sufficient to allow

continuous transmission. It is also clear that in these cases, when $RR > ITR$ and implosion losses occurred, they did not affect the T. The reason is that with $L = DP$, the sender never blocks despite losses, and hence can continuously transmit until the DP-th packet is sent.

The runs PF-30 and PF-60 follow a completely different pattern: their throughput increases rapidly as RR approaches $1,000$, due to (**fresh state**); for $1,000 < RR \leq 1,500$, the rate of increase is reduced probably because (**feed excess**) comes into effect even before RR approaches ITR. Note that RR can be exceeded in certain epochs, due to potential time delays (see Section 3.4) and the jitter. For values of RR larger than ITR, (**feed excess**) is more dominant and T falls.

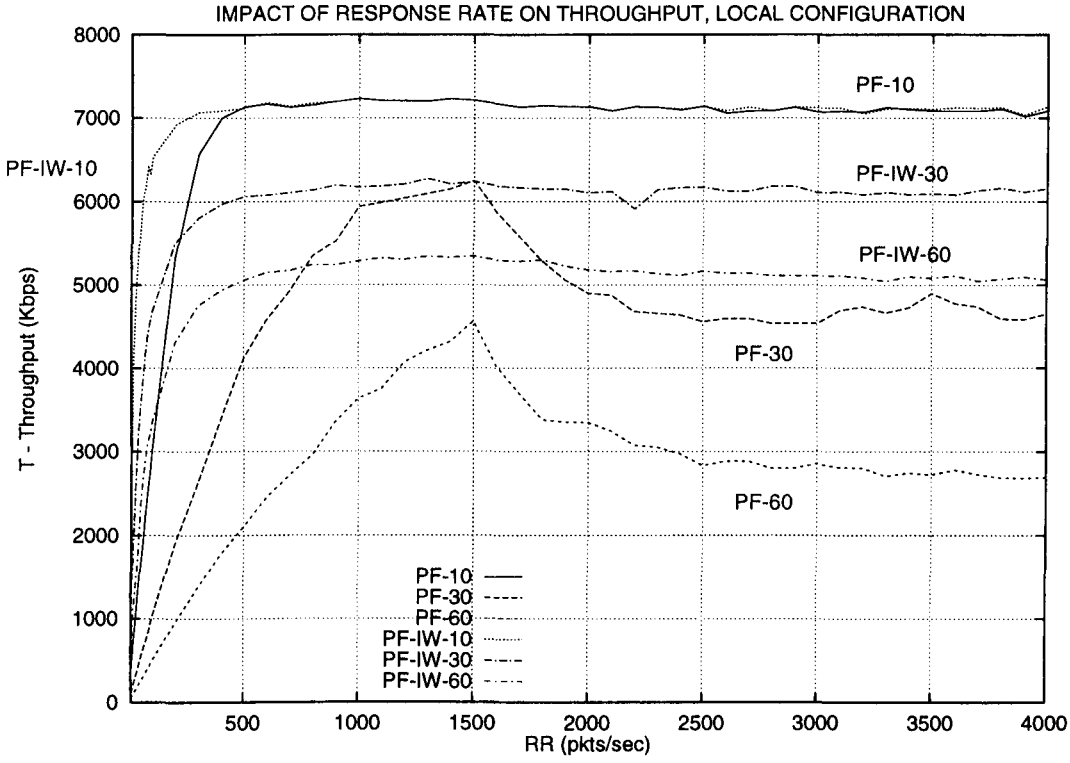


Figure 4.14: Effect of the RR value in the throughput T in the LOCAL configuration.

As expected, losses start occurring once $RR > ITR$, as illustrated in Figure 4.15. This can be observed for all group sizes, with or without infinite window. The number of implosion losses is notably higher for PF-60 because the increased cumulative loss experienced by the sender leads to window blocking (this is confirmed by Figure 4.4, where the graph of PF shows loss of T for $GS > 30$). Window blocking leads to more POLLs and responses being exchanged;

with a proportion of all responses being lost due to implosion, I increases. The network cost

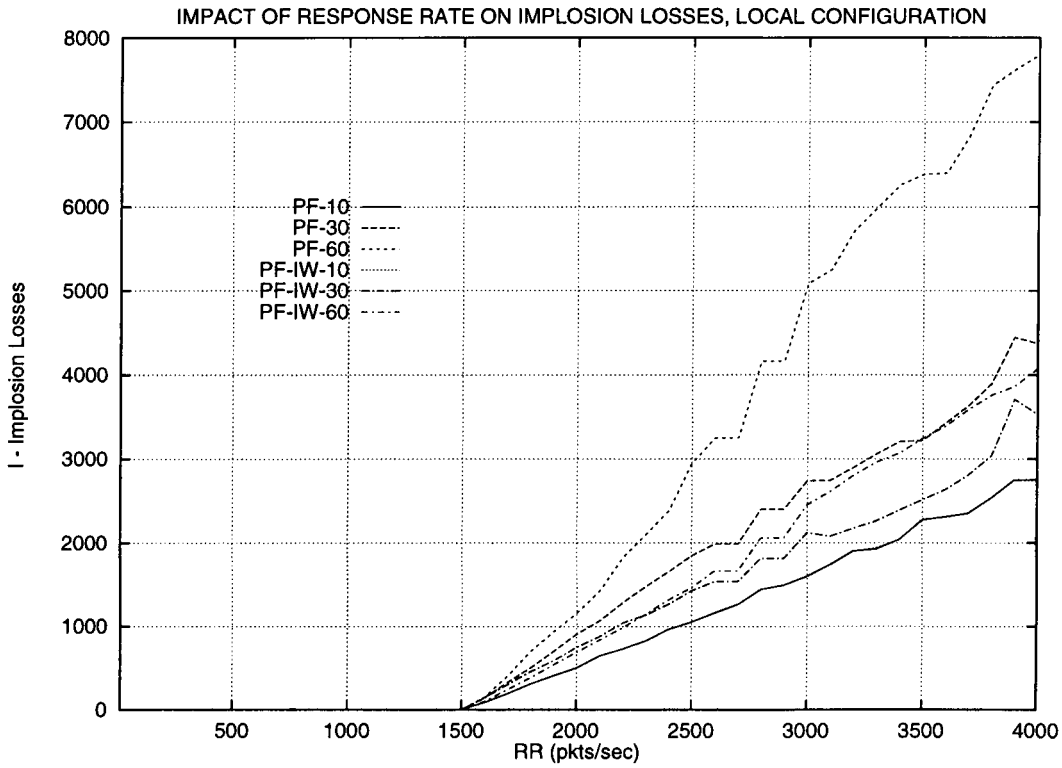


Figure 4.15: Effect of the RR value in the number of implosion losses in the LOCAL configuration.

reflects the implosion losses shown in Figure 4.15. The graphs for all runs increase with RR, as the receivers will be returning more packets during the communication.

The graphs of PF-IW runs for the three different group sizes in Figure 4.14 indicate that even $RR = \frac{ITR}{3}$ is sufficient to achieve the best T; that is, the runs with infinite window are not significantly affected by lower RRs. In contrast, if $L = 64$, the best T is achieved with the RR set to an exact estimation of ITR. Figures 4.15 and 4.16 show that the smaller the RR, the fewer the implosion losses, and also smaller the network cost. Therefore, if ITR cannot be estimated accurately, an underestimation is preferable to overestimation, in particular for the cases with infinite window.

4.3 Concluding Remarks

The past two chapters have described flat PRMP, a sender-, fully-reliable multicast protocol. The core of PRMP protocol is its novel, one-to-many sliding window mechanism. This window

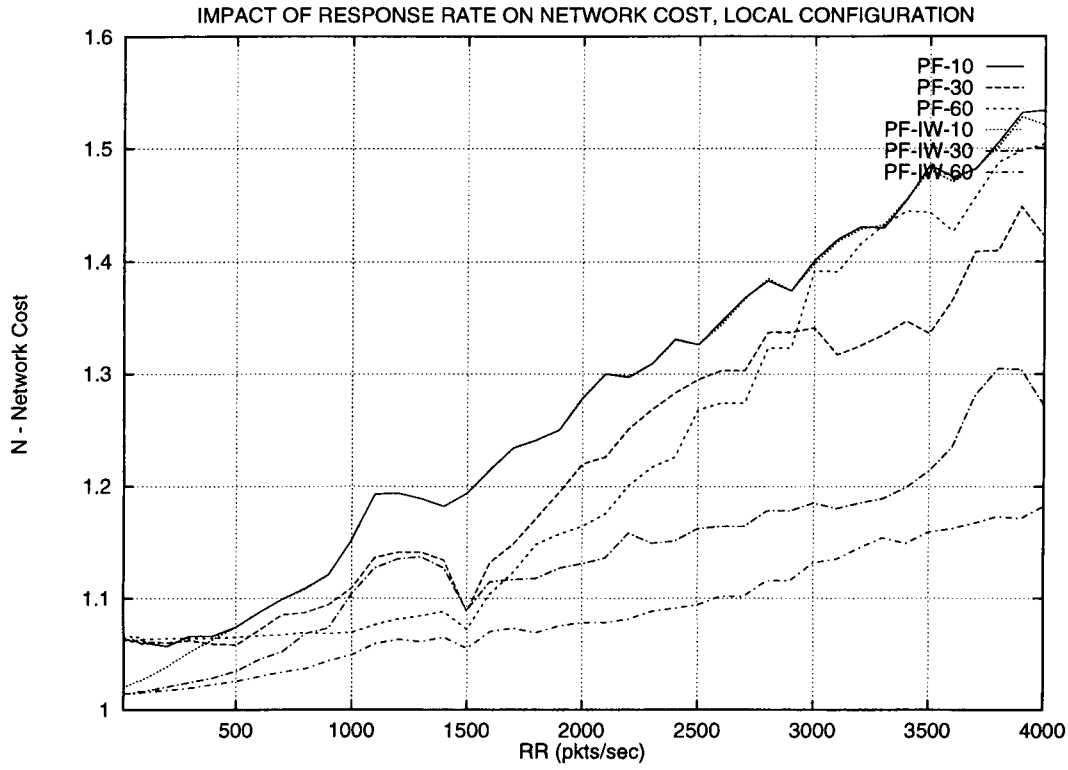


Figure 4.16: Effect of the RR value in the relative network cost N in the WIDE configuration.

scheme devised for PRMP has been shown, through simulation results presented in this chapter, to achieve efficient error and flow control. Traditional sliding window mechanisms, such as the one employed by TCP, do not scale for reliable multicast due to ACK-implosion. PRMP embodies a polling-based implosion avoidance mechanism, in which the polling of receivers is timely planned. Although the processing requirement per feedback packet is increased at the sender, the amount of feedback packets can be immensely reduced, avoiding implosion. The low network cost stemming from reduction of feedback, as well as careful use of unicast and multicast while retransmitting, provides *cost-effective* reliable multicast.

From the set of simulation results, two main conclusions can be drawn about the protocols:

- the throughput T and network cost N of the Full Feedback protocol were dominated by I. The Full Feedback protocol runs with $ITR = \infty$ performed well; because of implosion losses, FF-IW and FF scale very poorly (prohibitive N and low T for all group sizes over half a dozen receivers);

- both T and N of flat PRMP were affected by the window length L employed. This is

no surprise, as (sliding window) reliable unicast protocols are very much affected by the bandwidth and latencies of the network between the sender and the receiver. In fact, overall PRMP *was not affected* by the L value because of its implosion avoidance scheme, but because of the high latencies and error rates used in the `WIDE` configuration. In this configuration, the T and N of PF for *all group sizes* between 3 and 60 receivers were better than the T and N of FF for 3 receivers only (see Figures 4.8 and 4.9).

However, the scalability of PRMP is limited. The implosion avoidance mechanism reduces the flow of feedback to the sender so that no feedback packets are wasted resulting from implosion losses. Because PRMP neither assumes an infinite window (i.e., L will be smaller than DP) nor infinite feedback capacity (i.e., ITR will be finite), there will be a given GS which is large enough to make RR become the bottleneck in the communication. The window will start to block frequently, leading to a decrease in T and an increase in N (because of POLL packets).

The approach followed to increase the scalability of PRMP is to use *hierarchy*: to extend the flat PRMP protocol to a tree-based version. The set of receivers is organized according to a tree, with the sending application at the root, and receiving applications at the leaf or internal nodes. Each set of parent and child nodes runs basically the same protocol which has been described in Chapter 3. The next chapter addresses the changes which are required to flat PRMP to extend it to hierarchic PRMP, including issues related to wide-area network multicasting like *localized loss recovery* and *congestion control.*

Chapter 5

Hierarchic PRMP

Chapter 3 described the flat version of PRMP; Chapter 4 showed through simulation results that this flat version can scale well up to group sizes of (at least) 60 receivers. The scalability of the flat PRMP protocol is, however, limited. In large networks, where RTT delays can be large, data and control packets have to travel to and from distant receivers; it becomes more expensive to poll receivers using POLL packets.

Additionally, flat PRMP cannot explore spatial correlation of losses, typical of multicasting in WANs. As discussed in Section 2.2.1, when a packet is lost in a non-leaf node of the tree, all nodes which are downstream (of the loss) will experience the same packet loss, and request its retransmission. In case there is a lossy node or link near the "bottom" of the tree (close to a leaf), it is likely that certain losses will be typical of a given region or domain. A multicast protocol may employ remote "representatives" which are able to detect the loss of a nearby receiver and quickly recover from it with a retransmission. This is often referred to as "local recovery", and it provides two potential benefits: firstly, the latencies between the representative and the lossy receivers are overall smaller, speeding up recovery; secondly, the loss recovery process (via retransmissions) is *isolated* from the rest of the network.

In the flat PRMP, the sender communicates directly with all receivers, with no such representative agents in between. This chapter extends this protocol for a tree structure in which the sender ("source") is a *parent* of only a subset of receivers ("children")[1], and each of the

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

source's children is the parent of another subset of receivers and so on. This tree-based extension involved developing enhanced flow control schemes and congestion control mechanisms, and it has enabled PRMP to scale for a large number of receivers.

This chapter discusses the extension of the flat PRMP to hierarchic, and is organized as follows. The tree structure and notation are presented in Section 5.1; Section 5.2 describes the process of *forwarding packets* through this tree structure. The remainder of the chapter is organized in four sections, each addressing one main mechanism of hierarchic PRMP: error control (5.3), flow control (5.4), congestion control (5.5), and session control (5.6).

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 node 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 with the number of receivers the source interacts with;
- *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.

5.1 The Tree Structure

In PRMP's case, the tree structure is used not only for error recovery but also for data propagation: each node receives data packets from its parent and transmits (*forwards*) data packets to its children. In other words, the source multicasts data packets *only* to its children, and not to the complete tree. The motivation for this tree-based propagation of data is twofold:

- *polling mechanism*: when a parent node multicasts data packets to its children, it can piggyback the polling requests to selected children and thereby avoid the overhead of sending explicit POLL packets;

- *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 position to detect and deal with problems regarding its receivers (such as congestion) more quickly and effectively.

Figure 5.1 illustrates an example of the tree-based structure in PRMP. The nodes taking part in a given session are denoted either as "source" or "receiver"; the source is at the *root* of the tree, and receivers are either *internal* or *leaf* nodes. Receivers interact with each other according to the logical hierarchic organization, which has been previously established (and remains static during the session). Not all receiver nodes contain a receiving application, and this is the case for receiver R_3 (shaded) in Figure 5.1.

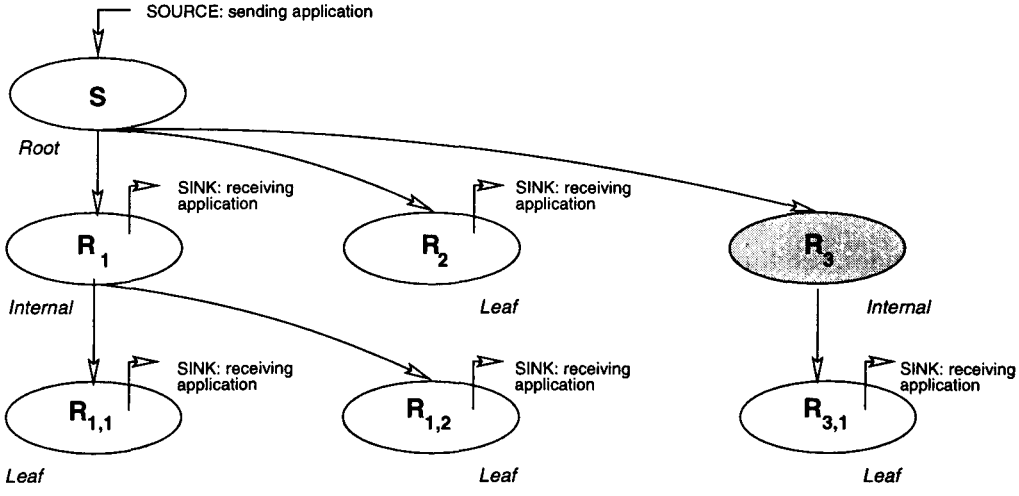


Figure 5.1: Example of the tree-based structure in PRMP.

As shown in the example of Figure 5.1, hierarchical naming is used to uniquely identify nodes in the PRMP tree. The source has "nc" children and is denoted as S, and its set of children are $\{R_1, R_2, ..., R_{nc}\}$. Whenever the context requires, the node's name is prefixed to a variable (so that "$S.nc$" can be used to represent the nc of S). R_1 has $R_1.nc$ children, which are $\{R_{1,1}, R_{1,2}, ..., R_{R_1.nc}\}$, etc. The set of R_i's children is denoted as $\{R_{i,*}\}$.

The source, at the root, does not have a parent, but has one or more children; leaf child nodes have a parent, but do not have children of their own; internal nodes have both a parent and one or more children.

Parent nodes are expected to send packets to their children, while child nodes are expected to receive packets and, when polled, send a response to their parent. The functionality of nodes can be divided accordingly:

- *sending role*: to transmit packets to receivers (child nodes), poll receivers for responses, and retransmit data packets when required;
- *receiving role*: to receive packets from the sender (parent node), return acknowledgments when polled, and deliver data to a local receiving application, if one is present.

So, nodes in the tree can have a sending role, a receiving role, or both. The association between roles and nodes is shown in Table 5.1.

Node	sending role	receiving role
source	√	
internal receiver	√	√
leaf receiver		√

Table 5.1: List of node types and their roles.

In internal nodes, sending and receiving roles coexist, and share a buffer of L packets. As in the flat scheme, sliding windows are used to represent the allocation of data units to buffers. Therefore, a receiving role employs a receiving window, rw, to keep track of which packets have been received and/or consumed; the sending role maintains a sending window, sw, which represents the status obtained in responses returned by receivers. The one-to-many sliding window scheme described in Chapter 3 is employed for the communication between a parent and its children, much like the sender and receivers do in the flat scheme. In fact, the protocol is virtually unchanged for the root and leaf receivers.

However, for *internal* receivers, changes are required: the L buffers available are *shared* between the sending and receiving roles. When a packet *seq* arrives at a given internal receiver R_i, it is stored in the buffers and remains there until it is of no use for both receiving and sending roles of R_i. Therefore, the receiving window of R_i ($R_i.rw$) is *tied*[2] to the sending window of R_i ($R_i.sw$). A schematic view of all three types of nodes is illustrated in Figure 5.2.

[2]the explanation of *how* they are tied is delayed until Section 5.4.

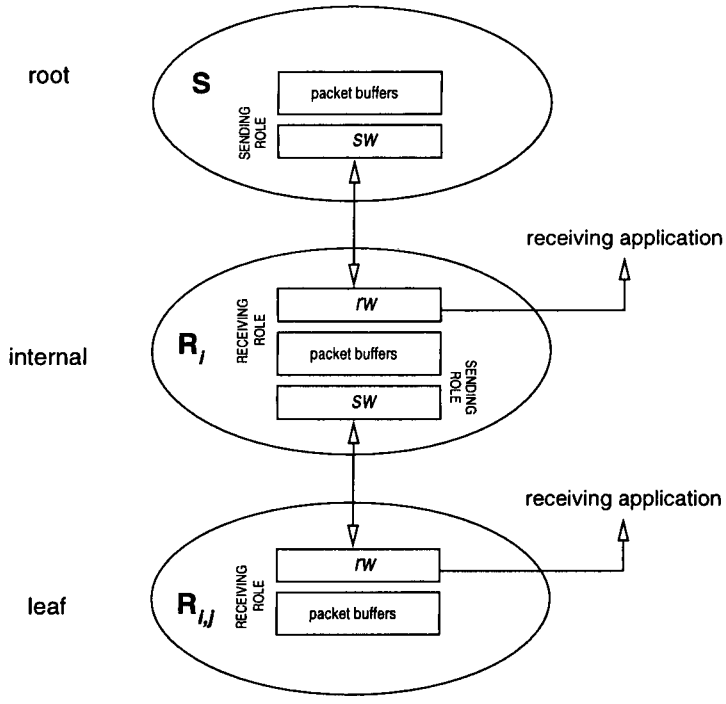


Figure 5.2: Schematic view of internal node R_i.

5.2 Forwarding Packets

In the root node, the sending application uses `write()` calls (like described in Chapter 3) to produce data for transmission. The source transforms the byte stream into data units and transmits them as data packets. Before these data packets can reach the child nodes, they are subject to duplication, loss, and reordering. Duplication is easy to tackle: a receiver detects that packet *seq* has been received ($rw.v[seq] = 1$) and discards the redundant copy; loss and reordering, however, allow packet $seq + 1$ to arrive at R_i before *seq*. Upon reception of packet $seq + 1$, it is R_i's role to forward $seq + 1$ to its children, $\{R_{i,*}\}$. Hence, unlike the production of data by the sending application at the source, the production of data at an internal node is *unordered.*

From the protocol design point of view, there are two options for forwarding of packets by internal nodes:

- *force sequential forwarding*: only transmit a given packet after all packets with smaller sequence have been transmitted (i.e., a packet *seq* cannot be transmitted if $seq > sw.hs + 1$); or

- *unordered forwarding*: do not restrict the forwarding of packets because of ordering.

To prevent introducing delays in the transmission of packets, PRMP adopts unordered forwarding. A time diagram illustrating this forwarding process is shown Figure 5.3, and commented below.

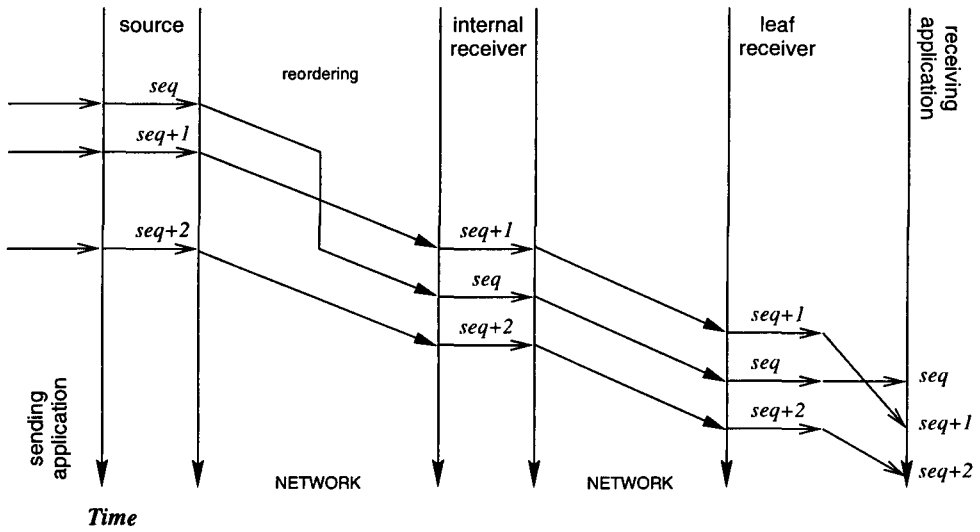


Figure 5.3: Example of forwarding of packets.

In the example, there are three nodes: the source, an internal receiver, and a leaf receiver. The production of data by the sending application results in data units being packed and transmitted through data packets with sequences *seq*, $seq + 1$, etc. Packets *seq* and $seq + 1$ are reordered by the network, and $seq + 1$ arrives before *seq*. Assume that the internal receiver can immediately forward the packet $seq + 1$ to its children (in the example, the leaf receiver). Packets arrive at the leaf receiver in the order they were transmitted by the internal receiver (that is, *out of order*). The packets are stored and delivered in the correct order to the receiving application.

The forwarding of packets applies only to data units: any control information in packets, such as a polling request in a (DATA)POLL packet, regards only the communication between a sender (parent) and its receivers (children). It is the receiving role of a node that deals with polling requests, responding in the traditional manner with a RESP packet. When a new data packet is received, its data unit is stored in the corresponding packet buffer and *rw* updated like in the flat scheme (see Section 3.2.1).

The sending role of an internal receiver R_s checks whether the new data packet can be forwarded to its children, that is, $\{R_{s,*}\}$. This check involves verifying whether all children are known to have buffers available to receive data packet *seq*. Recall that the in the flat scheme, it is sufficient to check the value of the available window, $sw.aw$, to determine the number of "new" data packets that the sender can transmit ($sw.aw \leftarrow sw.re - sw.hs$). In the hierarchic scheme, $sw.aw$ is not sufficient; because it is possible that an internal node transmits packet $seq + 1$ before *seq*, and then $sw.hs \leftarrow seq + 1$, not necessarily all packets with sequence *seq* such that $seq < sw.hs$ have been transmitted.

So, instead, R_s checks whether the packet is "transmittable"; recall from Section 3.3 that a packet *seq* is transmittable if $seq \leq sw.re$ (otherwise receivers may not have a buffer to store *seq*). If *seq* cannot be transmitted now, R_s will check again whenever $sw.re$ slides forward (see Section 5.4). If after a response has been processed, $sw.re$ slid forward from re_1 to re_2, all packets with *seq* such that $re_1 < seq \leq re_2$ that have been received by R_s ($rw.v[seq] = 1$) will be then forwarded.

5.3 Error Control

As described in the previous section, in the hierarchic scheme it is possible that an internal node R_s sends packets out of order: if $seq + 1$ arrives at R_s before *seq*, and if flow control at R_s allows its forwarding, $seq + 1$ is transmitted before *seq*. This is a major difference to the flat protocol, and its implications are discussed below.

Recall from Chapter 3 that in the flat scheme, the sender indicates to receivers which is the highest sequence number transmitted so far by including the value of $sw.hs$ in polling requests (as (DATA)POLL.hs). In the hierarchic scheme, $sw.hs$ still represents the highest sequence *seq* sent, but not necessarily all packets up to $sw.hs$ have been sent. Therefore, $rw.hr$ and $sw_i.hr$ cannot be reliably used to detect gaps in the packet sequence, and therefore, losses (i.e., using the scheme in Section 3.6.1).

Therefore, the method of using RESP.$rw.hr$ to identify NACKS needs to be adapted, since packets with $seq \leq$RESP.$rw.hr$ may not have been forwarded at the time the polling request was sent. Figure 5.4 shows a scenario where the above case is exemplified. Data packets DATA *seq*

and DATA $seq+1$ are transmitted and reordered by the network before reaching R_s. The packet DATA $seq+1$ arrives at R_s, and is forwarded to $R_{s,i}$ immediately. Soon afterwards, R_s transmits a POLL packet with POLL.$hs = seq+1$. $R_{s,i}$ receives DATA $seq+1$ and the POLL; it responds to the POLL by sending a RESP packet with RESP.$rw.v[seq] = 0$ and RESP.$rw.hr = seq+1$. In the mean time, the delayed DATA seq arrives at R_s, and is forwarded to $R_{s,i}$. Later, the response from $R_{s,i}$ arrives at R_s: as RESP.$rw.hr = seq+1$ and RESP.$rw.v[seq] = 0$, R_s could wrongly infer that RESP.$rw.v[seq] = 0$ is a NACK. But seq has been received and transmitted by R_s *after* the POLL, and therefore is not a loss.

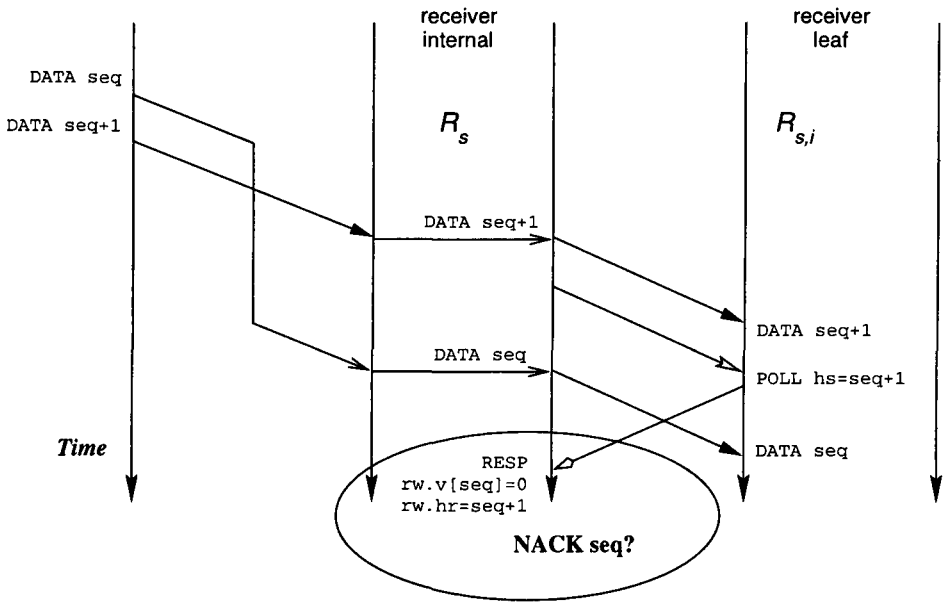


Figure 5.4: Example of scenario where the loss detection inferences used in the flat PRMP do not work.

To solve this problem, the sender works out the causal order between multicast transmissions and polling responses, by comparing the timestamp RESP.ts in responses with the time a packet has been originally multicast. The sender records the time a data packet is first multicast, denoted as Tx_{seq}, for all packets seq such that $sw.nea \leq seq \leq sw.hs$. (All packets before $sw.nea$ have already been fully acknowledged, whereas all packets $seq > sw.hs$ have not been transmitted.) The values of Tx_{seq}, which are recorded in the "Transmission (Times) Table", are initially set to ∞. A sender updates Tx_{seq} when it transmits the packet seq[3].

Recall that RESP.ts is a copy of POLL.ts, the timestamp of R_s included in the POLL which

[3] Tx_{seq} remains ∞ while seq is enqueued for transmission in the Transmission Queue, TxQ.

prompted RESP to be sent. So, similarly to the scheme employed to detect obsolete NACKs (Section 3.6.5), the sender compares the timestamp in the response (RESP.ts) with the forwarding time recorded for seq, Tx_{seq}: it considers RESP.$rw.v[seq] = 0$ as a NACK only if RESP.$ts \geq Tx_{seq}$ (polling request was transmitted after or with the transmission of seq).

The above rule must also be used in the evaluation of receiver sets (and their corresponding predicates). As the most recently arrived RESP packet may not be available anymore when the predicate is evaluated, the response's timestamp must be recorded. Hence, a new attribute is added to sw_i:

hts the highest timestamp recorded in responses from R_i (updated as $sw_i.hts \leftarrow \max\{sw_i.hts, \text{RESP}.ts\}$).

The predicates $Refed(seq, R_i)$ and $Nacked(seq, R_i)$ defined in Section 3 are amended so that they are *true* if,

$Refed(R_i, seq)$: packet seq has been *referenced* by receiver R_i if the sender has received from R_i a RESP packet which causally succeeds the multicast transmission of seq (i.e., $sw_i.hts \geq Tx_{seq}$);

$Nacked(R_i, seq)$: packet seq has been negatively acknowledged by receiver R_i if, having referenced seq (i.e., $Refed(R_i, seq)$=*true*), the corresponding bit in v is 0 (i.e., $sw_i.v[seq] = 0$).

Figure 5.5 shows one case where $seq+1$ has been referenced but not seq. Packet DATA.$seq = 32$ is transmitted at time 10; a POLL is sent at time 20, thus with POLL.$ts = 20$, and at time 30, packet DATA.$seq = 31$ is transmitted. When the response arrives, at time 55, its timestamp RESP.$ts = 20$ is smaller than $Tx_{31} = 30$, which indicates that $seq = 31$ was transmitted after the poll, and is thus unreferenced ($Refed(R_i, seq) = false$). However, POLL.$ts = 20$ is greater than Tx_{32}, so that RESP.rw refers to the transmission of $seq = 32$, which occurred at time 10.

The functions which generate receiver sets based on sw remain unchanged, as they apply the already changed corresponding predicate. The attribute $sw.hr$ is still used in error control to limit the range of packets which may be in recovery, as it still represents the highest packet

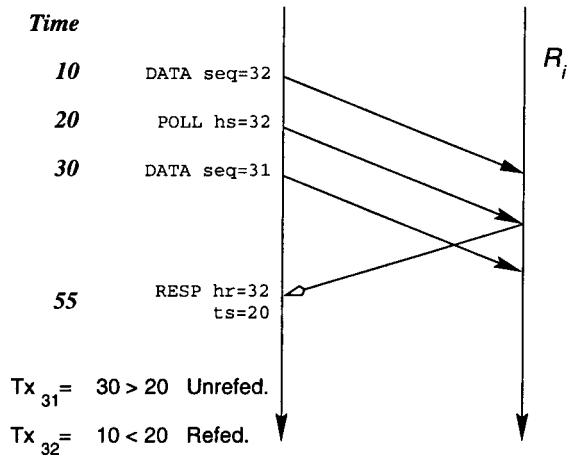


Figure 5.5: Example of use of Tx_{seq} to identify which packets have been referenced by receivers.

which has been referenced by receivers. This may be alternatively determined as the maximum Tx_{seq} (but not ∞) that has been referenced (i.e., is equal or smaller to $sw_i.hts$) by all receivers:

$$\max\{Tx_{seq} \mid Tx_{seq} \neq \infty \wedge Tx_{seq} \leq \min\{sw_i.hts \mid R_i \in \{R_{s,*}\}\}\}$$

The *sw.hr* approach was chosen because it is clearly the more efficient of the two.

A more comprehensive example of error control is illustrated in Figure 5.6, which shows the communication between the source, an internal node and its child.

The parent of R_s, R, 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 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 ($Tx_{33} = \infty$). Just before time 110, a retransmission of $seq = 33$ from R arrives at R_s, and is forwarded to $R_{s,i}$ as DATA.$seq = 33$; R_s sets its Tx_{33} with 110. Soon after forwarding $seq = 33$, R_s receives a response from $R_{s,i}$ with RESP.$rw.v[33] = 0$ and RESP.$rw.ts = 90$; R_s compares the timestamp in the response, RESP.$ts = 90$, with $Tx_{33} = 110$, and finds out that this response does not reference $seq = 33$ (response 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. As R_s updated $sw_i.hts$ when RESP arrived ($sw_i.hts \leftarrow 90$), if it later evaluates *Nacked*(33), the set will not contain $R_{s,i}$ and Refed(seq, R_i) will be *false*.

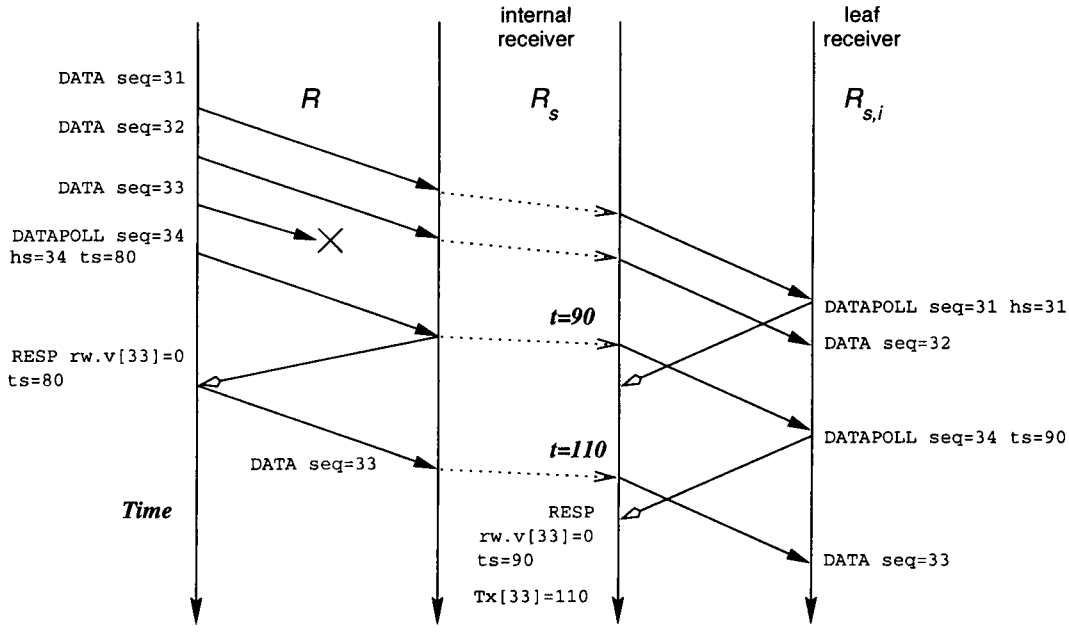


Figure 5.6: Example of communication involving three levels: R, R_s, and $R_{s,i}$.

The remaining aspects of the error control mechanism are preserved in the hierarchic version (as described in Chapter 3). This includes the packet states, waiting mechanisms for decision between multicast and selective unicast retransmission, and distinction between valid and obsolete NACKs.

5.4 Flow Control

Recall from Section 3.3 that the sender only transmits packets which it knows can be safely stored at receivers. Receivers report the highest sequence number they can store through RESP.$rw.re$. The sender uses $sw.re$, the aggregated right edge attribute (the minimum $sw_i.re$) to determine the available window: $sw.aw \leftarrow sw.re - sw.hs$. The $sw.aw$ indicates how many new packets after $sw.hs$ (i.e., $seq > sw.hs$) can be transmitted. Packets may be transmitted out of order, and the sender uses $sw.re$ to determine if a given packet seq is transmittable (i.e.,

if $seq \leq sw.re$).

At leaf receivers, the rw works exactly as in the flat scheme: $rw.re$ is increased when packets are consumed by the receiving application, that is, when $rw.le$ is increased (recall that $rw.re = rw.le + L - 1$). In internal nodes, however, the L packet buffers are *shared* between the sending and receiving roles. Each role demands the use of received packets, and restricts their removal from the buffers. As in a leaf node, the receiving role can release a packet seq only when seq is consumed by the local receiving application (i.e., $seq < rw.le$). As in the root node, the sending role has to keep packets with sequence seq until they have become fully acknowledged (i.e., $seq < sw.nea \vee Acked(seq) = \{*\}$). Hence, in an internal node R_s, a packet may be released from the buffers only when:

(**a**) packet has been consumed (i.e., $seq < rw.le$), and

(**b**) packet has been fully acknowledged (i.e., $seq < sw.nea \vee Acked(seq) = \{R_{s,*}\}$).

So, because the sending role may need to keep packets which the receiving role *would have released*, the number of packets that rw is able to keep may be smaller. This happens when the rw slides ahead of sw, and there exists one or more non-fully acknowledged packets that have to be kept by the sending role but that would have been otherwise released by the receiving role. The sending role needs these packets because it may have to retransmit them. They are, therefore, "subtracted" from the amount of available buffers which R_s reports to R with RESP.$rw.re$. Let *nfacked* be the number of packets seq in sw which *precede* the rw and that have not been fully acknowledged: all packets seq such that $sw.nea \leq seq < rw.le$ and $Acked(seq) \neq \{*\}$. If $sw.nea \geq rw.le$, *nfacked*=0. The receiving role calculates the right edge of its receiving window using:

$$rw.re \leftarrow rw.le + L - 1 - nfacked \ .$$

Figure 5.7 shows an example of the above case, where the rw has advanced 4 packets beyond sw, but the $rw.re$ has to be reduced by 1 because of *nfacked*=1. In the example, $sw.nea = 97 < rw.le = 100$, so the sender counts the *nfacked* packets and finds only $seq = 97$:

$Acked(97) \neq \{*\}$, $Acked(98) = \{*\}$, $Acked(99) = \{*\}$. The packets $seq = 98$ and $seq = 99$ (and equally $seq = 96$) have been discarded as they have been fully acknowledged. Since $nfacked$=1 and $L = 10$, $sw.re = 109 - 1$; hence, for R_s's receiving role, it can receive packets up to $seq = 108$.

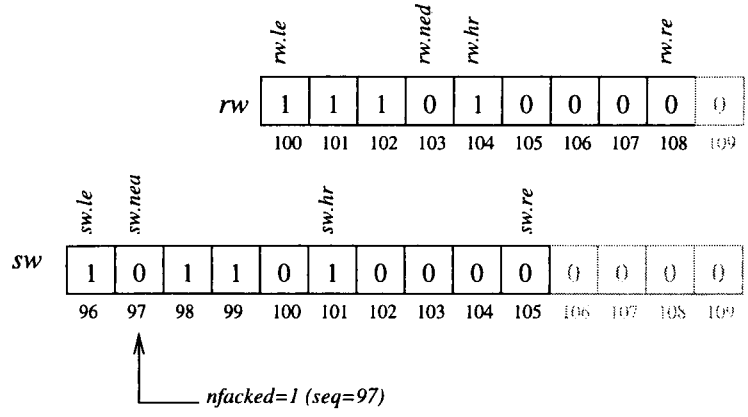


Figure 5.7: Example of sliding windows in internal node.

Recall that in the flat scheme the sender derives $sw_i.re$ directly from $sw_i.le$ using $sw_i.re = sw_i.le + L - 1$. Because the assumption $re \leftarrow le + L - 1$ does not hold in the hierarchic case, receivers have to include $rw.re$ in RESP.rw when transmitting a response (alternatively, the value of $nfacked$ may be sent, allowing the sender to compute itself the smaller, adjusted $sw_i.re$). The sender keeps status for the $sw_i.re$ attribute, instead of deriving it from $sw_i.le$, and updates it with $sw_i.re \leftarrow \max\{sw_i.re, \text{RESP}.rw.re\}$. In the flat scheme, the sender calculates $sw.re$ as $sw.re \leftarrow \min\{sw_i.le \mid R_i \in \{*\}\} + L - 1$; as $sw_i.re$ values are not necessarily $sw_i.le + L - 1$, $sw.re$ needs to be computed as

$$sw.re \leftarrow \min\{sw_i.re \mid R_i \in \{*\}\}$$

Figure 5.8 represents an extreme case where the sending role has all packets fully acknowledged, but cannot slide because one of its receivers $R_{s,i}$ is slow to consume. $nfacked$=0 because $sw.nea \geq rw.le$; so, the receiver reports RESP.$rw.re = 106 + 10 - 1 \Rightarrow 115$ when polled. All packets in the $[sw.le..sw.re]$ interval have been released, as they have satisfied both conditions (a) and (b) above. R_s has already received 5 new packets in the range $[rw.le..rw.hr]$; when sw eventually slides, these packets will immediately be available for forwarding.

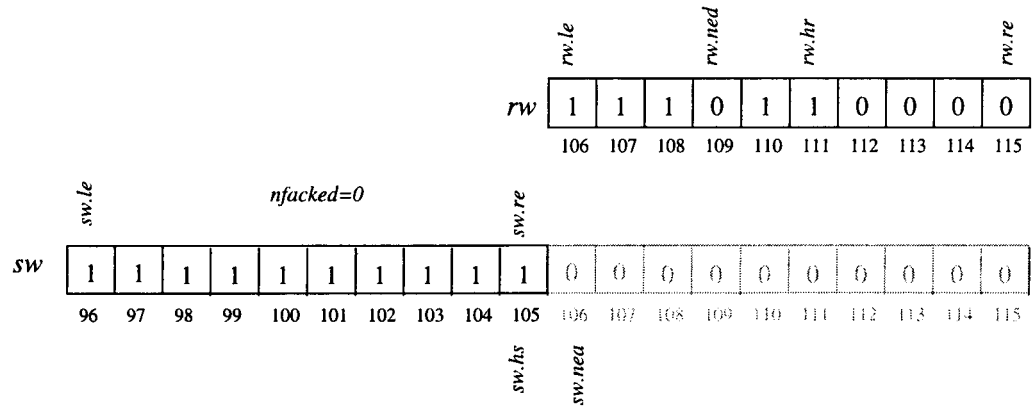


Figure 5.8: Example of case where sw and rw are completely "disjoint".

Continuing with the example of Figure 5.8, Figure 5.9 illustrates a case where rw "shrinks": $rw.le$ can slide forward as packets are delivered to the receiving application; $rw.re$, in contrast, cannot advance, since for every packet which is consumed ($rw.le \leftarrow rw.le + 1$), *nfacked* will increase accordingly (note that all packets seq such that $sw.re < seq < rw.le$ have not been forwarded yet, and thus will count as non-fully acknowledged). As $rw.re$ is computed adding $rw.le$ and subtracting *nfacked,* the end result will be the same. If the situation persists with the sending role, eventually the left edge of rw will reach its right edge, as illustrated in Figure 5.9.

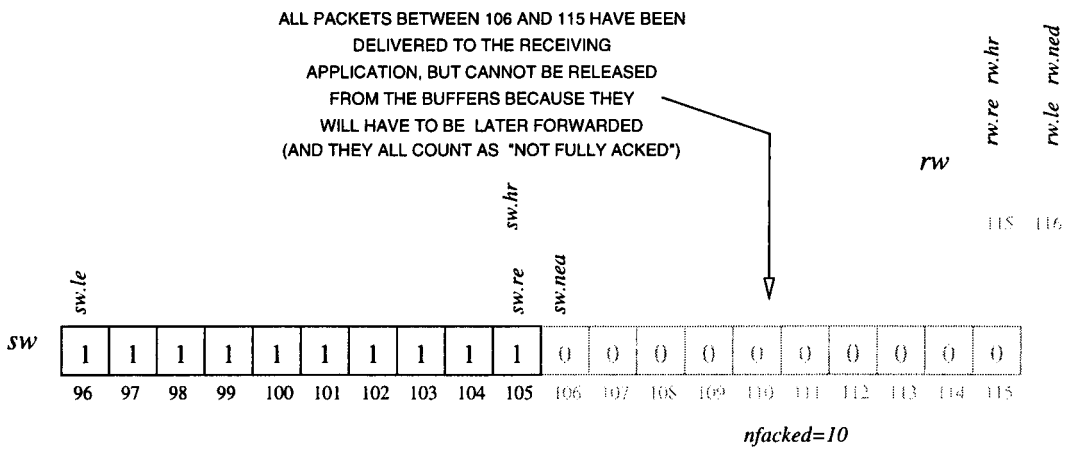


Figure 5.9: Example where the left edge of rw advances and reaches the right edge.

This flow control scheme ensures that if a given receiver falls back (is slow), there will be "backpressure" towards the source. If a given $R_{s,i}$, as shown above, continually reports the same RESP.$rw.re$, its parent, R_s will also eventually block because its $rw.re$ will not be able

to advance. If the problem persits for long enough, the root will eventually be blocked by the right edge of its sending window, and the sending application will not be able to *produce* more data. The pace in which the sending application is able to produce data for multicasting is restricted by the pace of the receiving applications in the destination set.

5.4.1 The "Nagging Parent" Syndrome

In TCP, when a receiver is unable to take more packets because its buffers are full (data has not been consumed by the application), the window advertised to the sender drops to 0. The sender will keep polling the receiver for every RTT until some data is consumed and the advertised window grows. When data consumption rate is very small compared to 1/RTT, many packets may be unnecessarily exchanged between sender and receiver. Like TCP, in PRMP the sender keeps polling at every RTT any child whose buffers are reported to be full.

A child reports a full buffer through a "window full of 1s"; recall from Section 3.2.1 that this situation is characterized in the sw_i of the parent R_s as $sw_i.nea = sw_i.re + 1$. In other words, all the transmittable packets in $R_s.sw$ have been transmitted and acknowledged by $R_{s,i}$ but $R_{s,i}$ has no space to receive new data. Since R_s cannot multicast additional data, it polls the slow $R_{s,i}$ once every RTT until $R_{s,i}.rw$ slides. That is, the parent R_s "nags" the slow child $R_{s,i}$ until the desired response is received. When the RTT is short, a packet pair once per RTT for each blocked child can represent a significant overhead.

In the flat scheme, this nagging syndrome occurs when the receiving application in one or more receivers is slow to consume the received data. In the hierarchical scheme, it can also be caused by grand-children that are slow to consume and/or to fully acknowledge the packets sent by the parent. Referring to Figure 5.1, the source S might "nag" the child R_3 (the shaded node in the figure) if the communication with R_3's child, $R_{3,1}$, was slow.

When the receiving application of $R_{3,1}$ is slow, R_3 is blocked from sending new packets and has to report a window full of 1s to R_3. The child nodes $R_{3,*}$ (only $R_{3,1}$ in this case) can be slow to fully acknowledge the packets transmitted by R_3 for three reasons:

(a) one or more nodes in the set $\{R_{3,*}\}$ may be experiencing frequent losses due to a congested router or a malfunctioning link;

(b) the rate at which R_3 can receive responses without implosion losses (i.e., RR) is low and/or the number of child nodes $\{R_{3,*}\}$ is relatively large. In these circumstances, R_3 may take a long time for R_3 to receive feedback from all its children and hence $R_3.sw$ may progress slowly;

(c) the latency between R_3 and the receivers $\{R_{3,*}\}$ is very large compared to the latency between S and R_3. This disparity in latency at two successive levels of the tree causes R_3 to receive the required acknowledgments from receivers $\{R_{3,*}\}$ more slowly (even in the absence of network problems) than it can acknowledge its parent S, leading to the nagging of R_3 by S.

Figure 5.10 shows a time diagram depicting a scenario in which the nagging parent syndrome occurs. R, the parent of R_s, transmits packets $seq = 31$ to $seq = 34$, and with $seq = 34$ a DATAPOLL. R_s receives the packets and immediately forwards them to $R_{s,i}$. The network discards packets $seq = 32$ and $seq = 34$ before they can reach $R_{s,i}$. As the packet DATAPOLL.$seq = 34$ is lost, R_s waits until the retransmission timeout expires. When it finally expires, R_s sends a POLL to $R_{s,i}$; $R_{s,i}$ responds and $seq = 32$ and $seq = 34$ are to be retransmitted (not shown in the figure). In the mean time, R keeps nagging R_s, until R_s can slide its rw.

PRMP prevents the nagging syndrome with a simple mechanism which *delays* the polling of children that keep reporting full buffers. Normally, upon receipt of a response indicating full buffers at the child, the parent would plan a poll to the child using RTT_i and the ARC vector so that a polling request is sent as soon as possible. If the ARC indicates that response quota is available in the aimed epoch, the polling request to R_i will be sent with the next packet transmission.

With the delaying mechanism active, however, the parent reduces the frequency of polling through a "*polling restriction delay*", denoted as prd_i (part of Polling Table entries), as follows. Initially, all $prd_i \leftarrow 0$. Upon receipt of a response from R_i with full buffer, the parent plans the next polling request to R_i using the current value of prd_i to delay the response: the POLL is not to be sent before the time $clock + prd_i$. Recall from Figure 3.9 that the first step in planning a poll is to use RTT_i to determine the earliest time a response could be received from R_i, regardless of the ARC contents. To delay a response, the parent simply adds prd_i

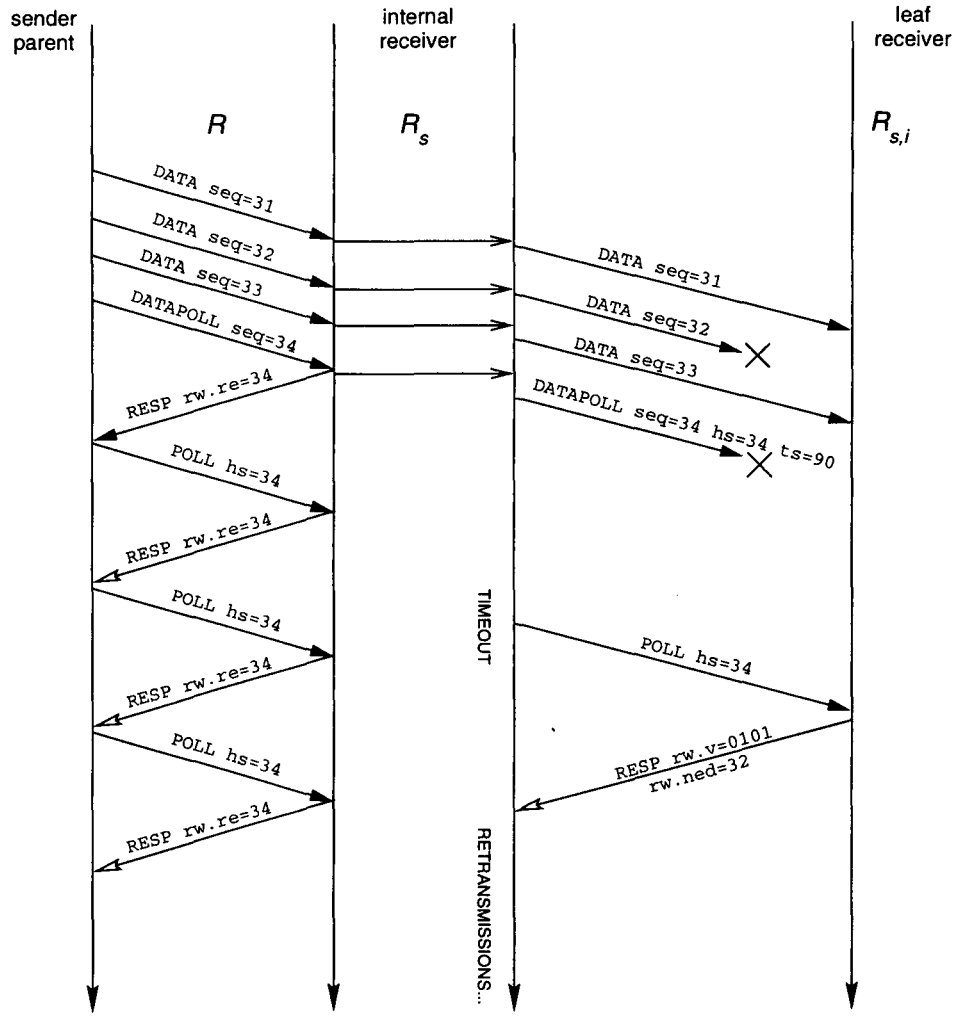


Figure 5.10: Example of scenario where the nagging syndrome may appear.

to RTT_i in determining this earliest time; given the way est_i is calculated, the transmission of the polling request is likely to be delayed too (depending on epoch length ε and IPG values). After "using" the current prd_i, the sender increases the prd_i delay in the following manner:

if this is the first POLL to be sent after full buffers were reported (thus $prd_i = 0$), the parent initializes prd_i with a default, initial value prd_{ini}: $prd_i \leftarrow prd_{ini}$;

otherwise, increase prd_i exponentially up to a maximum limit, prd_{max}: $prd_i \leftarrow \min\{prd_i \times k,\ prd_{max}\}$, for some $k > 1$.

Hence, if R_i keeps reporting full buffers whenever polled, prd_i will be backed-off exponentially successive times, so that R polls R_i with decreasing frequency. When the first response from R_i arrives such that sw_i slides forward, the delay is reset: $prd_i \leftarrow 0$.

This backing-off has, intuitively, negative impact on throughput. Ideally, as soon as a child "unblocks", a polling request will enable the child to report the slid window to the parent, but in practice, the child will have to wait some time until it gets a chance to send a response (see the time t in Figure 5.11). If a child is polled less frequently, this time will on average increase. Consequently, it will take more time until the sending window at the parent slides, probably slowing down the communication. The decision of using the mechanism or not, or how aggressively (the exponential factor k), is a trade-off between throughput and network cost.

5.5 Congestion Control

To deal with congestion in communications involving large internetworks, PRMP embodies two distinct congestion control schemes:

- *window-based (wb)*, and
- *rate-based (rb)*.

The window-based scheme is a variation of the [Jacobson88] congestion control scheme for TCP (in fact, this discussion applies particularly to the Internet context). The rate-based

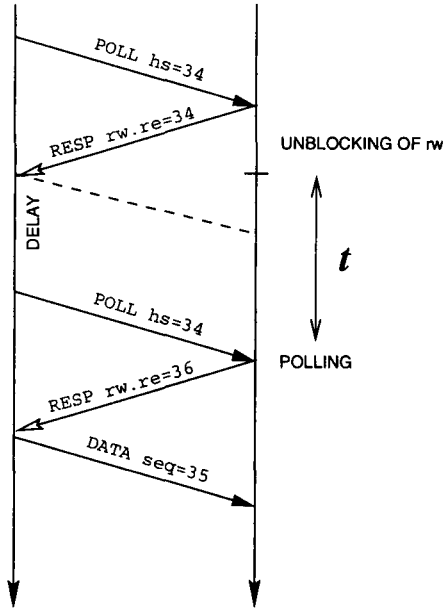


Figure 5.11: Delaying caused by anti-nagging mechanism.

scheme follows the same principles as the window-based, but varying the transmission rate in a different manner. Like error and flow control, each parent node "independently" applies congestion control in the communication with its receivers.

5.5.1 Detecting congestion

Based on the assumption that most losses in the Internet are caused by queue overflow in congested routers, and not by packet corruption, both *wb* and *rb* detect congestion through packet losses reported by receivers. A negative acknowledgment is seen as "hint" of congestion somewhere between the parent and its children. A parent "evaluates" the network according to feedback from receivers. It does so by aggregating "packet status" in *sw*, in the following manner: for any *seq* such that $sw.nea \leq seq \leq sw.hs$, packet *seq* is:

unreferenced if $Acked(seq) \neq \{*\} \wedge Nacked(seq) = \{\}$;

nacked if $Nacked(seq) \neq \{\}$; or

acked if $Acked(seq) = \{*\}$.

Recall that packets *seq* with $seq \leq sw.hs$ which have not been transmitted are distinguished by an infinite transmission time ($Tx_{seq} = \infty$); while a packet has not been transmitted, its state

is *"unreferenced"*, since $Acked(seq) = \{\}$ and $Nacked(seq) = \{\}$. After *seq* is transmitted, it stays *unreferenced* as long as it has neither negatively acknowledged by any receiver nor become fully acknowledged.

In the former case, as soon as any receiver R_i negatively acknowledges packet *seq*, $Nacked(seq)$ becomes $\{R_i\}$, and so, *seq* becomes *nacked.* In unicast communication this corresponds to *seq* being negatively acknowledged, an indicative that the sender must slow down. Additional NACKs referring to packet *seq* increase the cardinality of $Nacked(seq)$; the packet state does not change, remaining *nacked.* This is so because if n receivers negatively acknowledge a given packet, the load must not be reduced n times. Hence, *seq "remains" nacked* and the congestion control mechanism, either *wb* or *rb*, need not apply any measure to reduce the load.

In the latter case, no receiver reports the loss of *seq*, and packet *seq* remains *unreferenced*; positive acknowledgments arrive at the sender, so that the cardinality of $Acked(seq)$ increases. When all receivers (i.e., $\{R_{i,*}\}$), have ACK*ed seq* to R_s, $Acked(seq) = \{R_{i,*}\}$, and hence *seq* state changes to *acked.* This corresponds in unicast terms to the receipt of an "ACK *seq*", and is taken as an indication that the load may be slightly increased.

Following the optimistic approach employed in the design of PRMP's loss recovery mechanism, the congestion detection mechanism *does not act* on retransmission timeouts (which are likely to be caused by the loss of polling requests and responses). As explained in Section 3.6.4, the loss of a RESP packet *does not represent* the loss of the data packets it was supposed to acknowledge. It would be too conservative to multiplicatively decrease the transmission rate of the sender n times due to the loss of a single response packet (assuming that n packets would be acknowledged by it), especially considering these packets may have successfully arrived at the receiver.

Alternatively, PRMP can be configured to be more conservative and interpret a retransmission timeout as a *single packet loss.* The congestion control mechanism which is active, *wb* or *rb*, is signaled and reacts accordingly to slow down. This option has been tested with good results.

In conclusion, the detection mechanism evaluates the network load and provides feedback to *wb* or *rb,* which are prompted to control, increasing or decreasing, the current load. The

mechanisms, described in the next two sections, differ in the way they attempt to achieve the "right" load.

5.5.2 Window-based congestion control

In the window-based scheme, the transmission of packets is restricted by an additional aggregated attribute in *sw*, a "congestion window", denoted as *cwnd* (*sw.cwnd*). The value of *sw.cwnd* will vary between 1 and L packets. *sw.cwnd* is normally subject to:

- multiplicative decrease (halve *sw.cwnd*) when the detection mechanism indicates congestion (with an aggregated NACK, as explained in the previous section).
- additive increase (increase by 1 packet) when the detection mechanism indicates that a full congestion window (*sw.cwnd* packets) has been successfully transmitted (aggregated ACK for *sw.cwnd* packets). This is to probe for bandwidth which may become available.

Recall from Section 5.4 that flow control dictates that *sw.re*, the highest "transmittable" *seq*, is computed as $sw.re \leftarrow \min\{sw_i.re \mid R_i \in \{*\}\}$. The *wb* scheme alters the way the right edge of the sending window (*sw.re*) is calculated, reducing the range of transmittable packets. Because PRMP employs *selective retransmission*, only the packets in *sw* that are not fully acknowledged are counted as outstanding data. Let *"facked"* represent the number of *fully acknowledged* packets in the interval $(sw.nea \ldots sw.hs]$. When the *wb* congestion control is active, the definition of *sw.re* provided in Section 5.4 is modified so that:

$$sw.re \leftarrow \min\{sw_i.re, sw.nea + \lceil sw.cwnd \rceil + facked - 1 \mid R_i \in \{*\}\}$$

So, the lower values of *sw.re*, the more restricted the sender will be regarding the transmission and retransmission of packets.

At the beginning of the communication, the sender does not know in which condition the network is. TCP (like PRMP) employs window-based flow control, but unlike PRMP, it does not enforce a maximum transmission rate; so, when the transmission starts, a burst with L packets may be sent. This burst may overflow queues in routers, if their available capacity is insufficient to sustain it.

To solve the above problem, Jacobson's congestion control scheme employs *slow start*: the congestion window is initialized with 1 packet only, so that at first the sender will only be able to send a single packet. When the ACK for this packet is received, the sender increments the congestion window by 1, doubling the available window, so that two new packets are transmitted; the cycle is repeated: when each ACK arrives, the sender increments the congestion window by 1; after the two ACKs have arrived, the value of the congestion window will be 4. This exponential behavior of slow start continues for every RTT until either a loss is reported or L is reached.

PRMP employs slow start at the beginning of the communication. The value of *sw.cwnd* is initialized to 1 and is increased by 1 at every aggregated ACK as long as it remains in slow start phase. When the congestion detection mechanism indicates that the "right load" has been reached (that is, the aggregated state of a packet in *sw* has become *nacked*), the value of *sw.cwnd* is halved and the slow start phase terminates.

PRMP employs "fast recovery" (see [Stevens94]): slow start is applied only at the beginning of a session, and not after every loss.

5.5.3 Rate-based congestion control

The limitation of the above approach is that when the congestion control mechanism acts and closes the sending window, it does not impede the transmission of POLL packets. Recall from Section 3.4 that when a sw_i becomes "full of 1s", the condition (**full buffer**) is satisfied, a poll is planned for receiver R_i (if there is not one, already). With *wb,* upon losses, *sw* shrinks with the decrease of *sw.re* (and possibly *sw.aw*, too), which may cause the above condition to be satisfied for multiple receivers at once. As a result, receivers will be added to the Polling Table, and POLL packets sent to elicit responses from receivers with full buffer at every RTT (or so) until the window unblocks. Although the anti-nagging scheme may reduce the repeated polling, a given number of POLL packets will be sent and they add to the load of a congested network.

Additionally, as observed in the simulation experiments in Section 4.2, the blocking of the window (the tool of *wb*) is associated with an increase in network cost because more POLL

packets have to be sent.

To control the rate in which DATA/DATAPOLL/POLL packets are transmitted, it is necessary to dynamically alter the IPG. Recall that the IPG (Section 3.3) determines the minimum time that should elapse between any successive packet transmissions[4]. PRMP varies the transmission rate (the inverse of IPG) to apply congestion control.

The value of the IPG is varied in the same manner as in the *wb* scheme: multiplicative decrease of transmission rate (multiply the value of IPG) when congestion is detected, and additive increase of transmission rate (subtract from IPG) when no loss is observed.

Assuming that a window of L packets is transmitted and acknowledged, the mechanism additively increases the load/rate by 1 packet for the next RTT transmission cycle. Similar to increasing *sw.cwnd* by 1, the IPG is recomputed as "if $L+1$ packets were to be sent in the next RTT", so that more packets are sent within the same time interval. The end result is that the same number of packets is sent within a proportionally smaller period, increasing the transmission rate.

To emulate this behavior with a *gradual increase* after each of the L packets in the window becomes *acked* (as defined in Section 5.5.1), the sender computes the new transmission rate, nr, as

$$nr \leftarrow cr + \frac{1}{RTT_{max} \times L}$$

where cr is the current rate, and RTT_{max} is the current maximum RTT between the sender and its receivers, i.e., $RTT_{max} \leftarrow \max\{RTT_i \mid R_i \in \{*\}\}$. This change provides the expected additive increase in load.

The IPG varies within the interval $[IPG_{min}, IPG_{max}]$; IPG_{min} and IPG_{max} are values preset at start of the communication. When *rb* is being used, the user-configurable IPG works only as the initial value. For multiplicative decrease, at sign of congestion (a packet in *sw* becomes *nacked*), IPG is backed-off (doubled) using $IPG \leftarrow \min(IPG \times 2, RTT_{max})$.

[4]as noted in Section 4.1, with the exception of multiple unicast transmissions of the same packet, which are done without delay.

5.6 Session Control

The attempt to provide full-reliability can lead to blocking at the source, when a particular receiver persistently fails to respond to the polls sent to it. Without responses from a problematic receiver, the *sw*'s of all parents up to the source do not slide forward, preventing transmission of new data.

To guarantee progress, a parent removes the problematic receiver from its destination set $\{*\}$ if the receiver fails to respond to a given number of consecutive polls sent (i.e., after several RTO_i periods). It eventually informs its own parent of this removal and the information thus gets passed on upwards along the tree towards the source. The sending application can be informed by the source about the receiving applications that are not guaranteed to have received the full data contents.

In order to terminate a session, PRMP provides a `close()` downcall to the application level. The sending application is expected to use successive `write()` calls to send all data it wants to, and then request the termination of the session with `close()`. The source assembles and transmits the last data unit with any bytes left, i.e., that have not been transmitted yet (the last unit may be smaller than the previous ones). All packets transmitted from a parent to its children will thereafter carry the sequence number of the last data unit of the session (this is the *DP* value previously referred to). The sending application will remain blocked while the source coordinates the tear down of the session.

Receiving applications, on their turn, realize that the last data unit has been consumed (and its size) through ordinary `read()` calls (since receiving applications continuously consume data through `read()`). After that, a receiving application is expected to perform any required actions (typically closing a destination file) and then invoke `close()`.

After the *sending* application has used `close()`, the source keeps it blocked until reliability requirements are met. Flat PRMP offers two levels of reliability:

(**r**) all receivers are known to have *received* all packets; or

(**c**) all receiving applications are known to have used `close()`.

(**c**) provides stronger guarantees than (**r**) to the sending application since a receiving application

or node may crash between the reception of the last packet and processing of the corresponding data unit. In the hierarchic PRMP, each of the two reliability levels has two additional sub-levels, applicable to each parent:

(**f**) all *children* are known to have received all data (**f-r**) or have used `close()` (**f-c**);

(**h**) all *nodes in the subtree* are known to have received all data (**h-r**) or have used `close()` (**h-c**).

The paragraphs below discuss the implementation of the session termination scheme. An additional flag in responses, "RESP.*closed*", is employed to indicate to the parent (when polled) that the child is in the process of closing the session (it will keep the session control block for a given amount of time). A parent maintains a receiver set "*Closed*", initially empty, which indicates which of its children have closed. Whenever a RESP packet is received from R_i such that RESP.$closed = true$, R_i is added to the set (i.e., $Closed \leftarrow Closed \cup R_i$). When the parent node is the source, and its *Closed* set is complete (i.e., $S.Closed = \{*\}$), the sending application can be unblocked from `close()`.

In (**f-r**) case, the receiver will start reporting RESP.$close = true$ to its parent as soon as the last data unit, as well as all the previous ones, have been received (that is, $rw.ned = DP + 1$). In (**f-c**) case, each receiver containing a receiving application waits until its local receiving application uses `close()` before reporting RESP.$close = true$ (if the receiver does not have a receiving application it behaves like (**f-r**)).

In the cases of (**h-r**) and (**h-c**), the source depends on termination state being collated through the tree. A leaf receiver will perform like in (**f-r**) and (**f-c**), but an internal receiver R_s can only indicate RESP.$closed = true$ when all of its own children have reported that they have closed (that is, when $R_s.Closed = \{*\}$). Additionally, in (**f-c**) case, if there exists a local receiving application, the receiver needs to wait for the `close()` downcall as well. Note that due to the nature data packets are propagated, if a child receiver reports RESP.$closed = true$ (after receiving all data), it means that its parent must have also received all data.

Implementing all these reliability levels requires that a receiver does not remove the session

"control block" until the sender is known to have received at least one of its polling responses such that RESP.*closed* = *true*. After sending the first response with RESP.*closed* = *true*, the receiver starts a *session termination timer*. Every time the receiver is polled thereafter, it will send the same response and restart the timer. When this timer expires, the receiver erases the control block regarding the session.

Chapter 6

Prototyping & Simulation of Hierarchic PRMP

The multi-threaded, object-oriented architecture described in Chapter 4 defined a protocol node that sends and another that receives packets. In this chapter, that architecture is extended (in Section 6.1) to include nodes that both send and receive packets. This extended architecture is employed in the simulation experiments performed in the remainder of the chapter. In Section 6.2, the enhanced network simulation model is described; the hierarchic PRMP and FF are compared in Section 6.3; in Section 6.4 the anti-nagging mechanism is evaluated; Section 6.5 defines an "IMAGINARY" network configuration in which the experiments are based. The experiments performed with the window- and rate-based congestion control mechanisms of PRMP are discussed in Section 6.6. The last set of experiments performed, which is described in Section 6.7, provides a comparison between the flat and hierarchic versions of PRMP, for the IMAGINARY network configuration.

6.1 Protocol Architecture

The hierarchic PRMP presents little difference in terms of architecture in comparison to flat PRMP. In the latter, there were two types of nodes: sender and receivers. Receivers are fairly simple, implemented by a single thread, RM (Receiver Module). The sender embodied all

the mechanisms for error control, flow control, and implosion avoidance, and thus was more complex. Its architecture was multi-threaded with four threads which cooperated through two queues and five tables.

Recall that in the flat scheme architecture, the GM (Generator Module) interacts (at the sender) with the sending application to obtain data and enqueue it as data units in TxQ. In the hierarchic case, the above sender is the *source*, at the root of the PRMP tree; internal nodes do not have a sending application, and instead receive data packets from their parent through the RM thread. Therefore, in internal nodes, GM is replaced by RM (see Figure 6.1). Like GM, RM is restricted by flow control, and can only enqueue a new data unit *seq* for transmission if *seq* is "transmittable" (i.e., $seq \leq sw.re$).

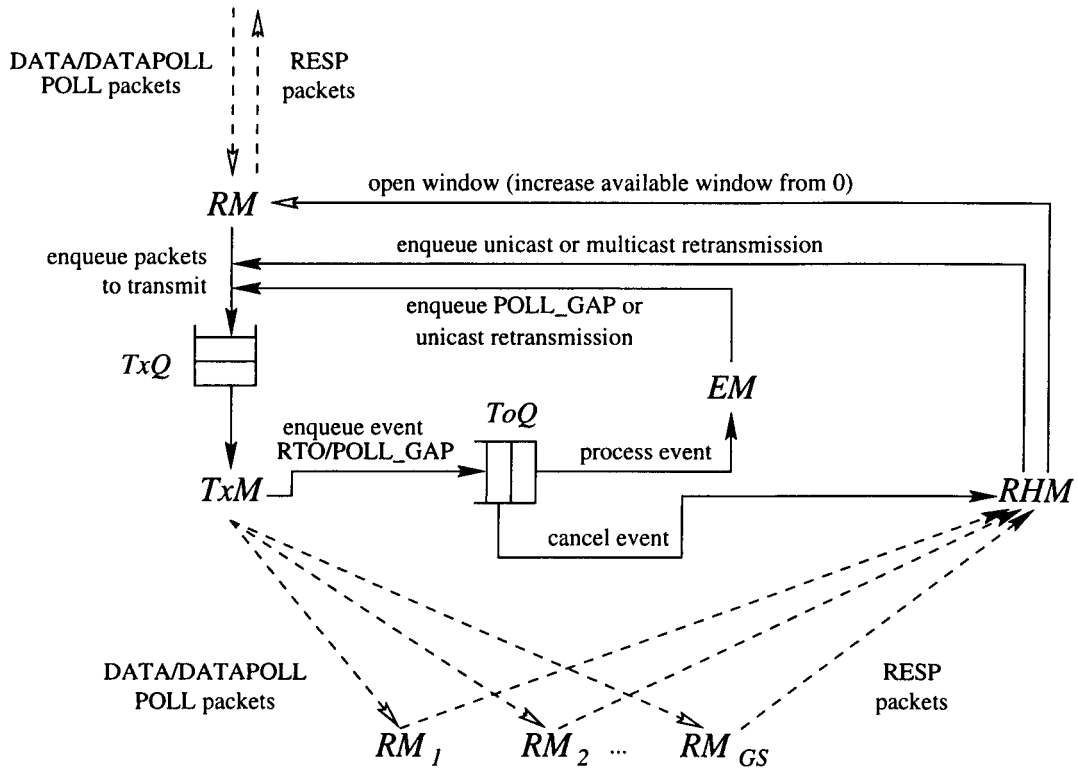


Figure 6.1: Architecture of an internal receiver node.

Recall that with window-based congestion control active, $rw.re$ can decrease, that is, the rw can shrink. Packets awaiting transmission in TxQ may become "unstransmittable": all entries tx such that tx.seq> $sw.re$ should not be transmitted or otherwise the congestion control mechanism cannot be effective in curbing congestion. Thus, before consuming the first entry of

TxQ, the TxM (Transmitter Module) verifies whether *seq* can be transmitted. If the first entry cannot be sent, then no other entry can. In this situation the TxM behaves as if the TxQ were empty: if there are planned pollings in the PPT, enqueue a poll gap event into the Timeout Queue, ToQ.

Two situations may cause TxM to unblock: (a) when the poll gap expires, the EM will enqueue a poll gap entry in TxQ and schedule TxM so that TxM can process it and send a POLL packet with due planned polls; (b) if RESP packets arrive, and RHM (Response Handler Module) can as a result increase *sw.re*, RHM schedules TxM so that it may be able to transmit new packets.

In terms of data structures, a new table is required at internal nodes[1] to record the time packets are forwarded (it is used by the internal node to determine the relative order between the forwarding of a packet and a poll response). Recall that Tx_{seq} represents the forwarding time of *seq*, equal to ∞ if *seq* has not been transmitted yet. The Transmission Table, TxT, records the transmission time of up to L packets in the interval [*sw.nea..sw.hs*]. The time in TxT[seq] is not updated when a tx.seq is enqueued in TxQ, but only when tx.seq is consumed and when the packet is actually transmitted. Otherwise, a response generated by a polling request which was transmitted after *seq* was enqueued would wrongly refer to (and negatively acknowledge) *seq*. The complete set of data structures for an internal receiver node is shown in Figure 6.2.

6.2 Enhanced Network Model

A prototype of hierarchic PRMP was implemented according to the protocol architecture described in Section 6.1, and allowed the analysis of hierarchic PRMP through simulation experiments. This section describes the *network model* employed in this new set of experiments. In Chapter 4, flat PRMP was evaluated and compared with the Full Feedback protocol using a simplified network model: each receiver maintained an *independent channel* with the sender, with three attributes: mean and deviation of propagation latency, and loss probability per packet. All kinds of packet losses except implosion were generated using a *random draw*. The RTTs were modelled using the *Normal distribution*.

[1] in practice, at all sending nodes.

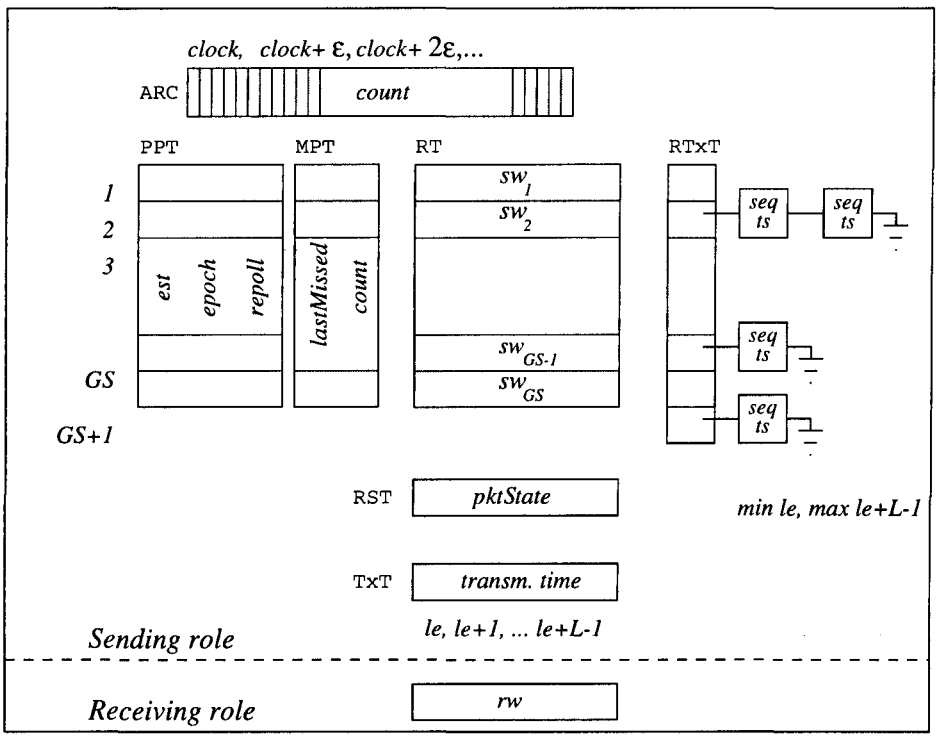


Figure 6.2: Structures used at sender and receivers.

The simplified model employed in Chapter 4 has the following limitations:

- the model is restricted to flat topologies;
- losses are independent and randomly generated, while loss correlation is required to better evaluate the error control scheme;
- RTTs are randomly generated, instead of fluctuate with the network load.

Therefore, a new network simulator[2] was designed to evaluate: (a) the hierarchic PRMP with different topologies and receiver allocations; (b) the poll planning and implosion avoidance scheme, in face of more realistic RTTs; (c) the error control mechanism, with correlated losses; and (d) the congestion control schemes proposed with topologies and routers. The tool provides packet traces, loss traces, queue occupation reports, and a handle for transport-level, protocol-dependent information tracing. Finally, it allows experiments to be performed with

[2]at the time this simulation work begun, the author considered using the network simulator "ns" [ns], but ns was too complex and lacked documentation.

congestion, by dynamically fluctuating the capacity of a router, making it temporarily behave as a bottleneck in the network.

The network is defined as a set of interconnected network nodes, each containing limited-size buffers (i.e., queues) and bounded processing rates (i.e., maximum rate at which elements are consumed from each queue). For any queue, a packet is only added to (the tail of) the queue if its maximum queue size will not be exceeded, otherwise the packet is dropped. Each queue is limited *both* by the amount of bytes it stores *and* the number of packets it contains. Each packet has a destination address, which may be a unicast or multicast address. Each node is uniquely identified by a "fictitious" network address (which is derived from the multicast tree hierarchy).

Packets that arrive at a node are first stored in a buffer, and then "routed" to one or more queues according to the packet headers (possibly to the upper layer of the local host). The buffer where packets arrive at and stay before being routed is the "`INCOMING`" queue (see Figure 6.3). Packets are consumed from the head of `INCOMING` in a FIFO basis with a maximum rate of $\frac{1}{T_{rout}}$. `INCOMING` limits the rate at which the network layer of a router or host can process incoming packets, including those generated by the upper layer.

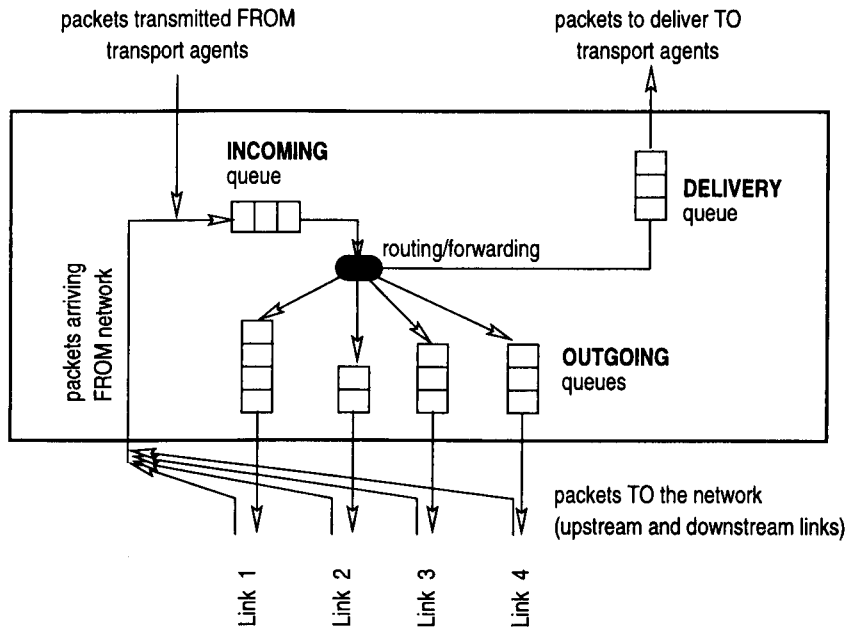


Figure 6.3: Schematic view of a simulated host.

If the node is a *host* and the packet is addressed to it (the uni/multicast address of the packet "matches" the id of the host), then the packet is routed (enqueued) into the "DELIVERY" queue. As its name suggests, the DELIVERY queue contains packets to be *delivered* to the local receiver. Recall that a receiver may have a sending role, a receiving role, or both. The *receiving* role of a node will continuously consume the first packet of type DATA, DATAPOLL, or POLL it finds in the DELIVERY queue, at a maximum rate of $\frac{1}{T_{pack}}$. The *sending* role of a node will continuously consume the first packet RESP it finds in DELIVERY, at a maximum rate of $\frac{1}{T_{resp}}$. These two rates are independent and correspond to the maximum speed at which the transport layer "agents" (in PRMP case, RM and RHM) will handle packets.

A multicast packet being consumed from INCOMING may be *replicated* into one or more of the "OUTGOING" queues; each OUTGOING queue is associated with a link (there is one OUTGOING queue for each link connecting the node). A packet is forwarded into one or more of the OUTGOING queues according to its destination address. Packets are consumed from each OUTGOING queue on a FIFO basis (no RED gateways [Floyd93] or *fair-queueing*) according to the bandwidth and latency associated with the corresponding link.

The simulated network can now be defined as "a set of network nodes, either hosts or routers, which are interconnected by communication links forming an arbitrary tree with the source at the root". The main difference between a host and a router is that the former may have a sender or receiver part of the PRMP tree, while the latter *only* routes packets. Both hosts and routers share the same conceptual routing abstraction. In hosts which contain a receiver, there may be a receiving application to consume the data (in the simulation results presented this was always the case); otherwise, the receiver would only route packets according to the hierarchic organization of the PRMP tree. Each communication link is point-to-point and bidirectional, and has a fixed propagation time (in ms), bandwidth (in bits/s) and corruption rate (in percentage of bytes).

Each network node is uniquely denoted using hierarchical addressing; the first letter of the name identifies the kind of node, 'h' for hosts and 'r' for routers. The root node, which has to be a host, is denoted as 'h0'. Its children are denoted as: 'h1' or 'r1'; 'h2' or 'r2', etc.; the children of, say, 'r1', are: 'h1,1' or 'r1,1'; 'h1,2' or 'r1,2', and so on.

Congestion and implosion can both be attributed to losses caused by buffer overflow (though implosion is also related to contention in shared-media networks). In the simulation experiments undertaken, packets could only be lost because of corruption or buffer overflow. In the former case, one of the bytes of a packet gets corrupted during its transmission through a link; at the receiving end of the link, the corruption is detected and the packet silently discarded.

In the latter case, a packet needs to join a queue which has already filled. This queue size restriction is applied in two ways: it limits the size of the queue *in bytes* (buffer size), as well as in *number of packets* (entries in the queue). If a new packet would exceed either limit, the packet is dropped. Each queue has associated a pair *bytes/packets* which describes its maximum capacity: Q_i for `INCOMING`, Q_d for `DELIVERY`, and Q_o for `OUTGOING`. The buffer overflow losses were classified in three types according to the type of packet and the node in which it occurred:

implosion feedback packets which are dropped at any node apart from the leaf nodes;

congestion non-feedback packets dropped at nodes without sender or receivers (routers or hosts acting as routers);

overrun data packets which are lost at the root (before being transmitted) or at the receiver's host (before being delivered), or feedback packets which are lost before being transmitted.

There is an additional input parameter associated with each host, T_{cons}. In hosts occupied by a receiver and a receiving application, data packets which become "consumable" at the receiver are passed in sequence to the receiving application. Assuming that the receiving application will always wish to consume data, the rate at which data units can be consumed by the receiving application is dictated by $\frac{1}{T_{cons}}$. This is particularly important to study scenarios in which flow control is to play a major part in the experiment.

The same output metrics defined for the simulation in Chapter 4, namely throughput T, network cost N, and implosion losses I, are used in the experiments reported in this chapter. T is the throughput (measured in Kbps) from the point of view of the sending application; it is determined as the amount of data provided by the sending application per time taken

to complete the reliable transmission (from the first packet to termination by the source). In the simulations with hierarchic PRMP, there are two variations of T, T_1 and T_2, depending on when the source terminates (according to reliability degrees defined in Section 5.6): T_1 refers to (**f-r**) and T_2, to (**h-c**). I is the number of packet losses caused by implosion. N is the relative network cost, calculated as the total number of packets exchanged per receiver per data unit. In this chapter, N is determined differently, to allow the heterogeneous set of distances (in hops) to weigh accordingly: the total number of packets ($totPkts$) now incorporates the number of hops traversed by each packet between sender and receivers. The resulting N will be equal to or greater than the N of Chapter 4, since a packet will have to traverse *at least* 1 link between a sender and a receiver in the tree.

6.3 Comparison between PRMP and FF

This section compares the hierarchic PRMP with the FF protocol (as in Chapter 4) using a tree topology in which *all nodes are hosts*. A group of 50 receivers ($GS = 50$) was chosen, and the 51 hosts were arbitrarily organized according to Figure 6.4. The figure shows, for each link, the propagation latency and the error percentage assigned to the link (the chance a byte gets corrupted when traversing the link). Table 6.1 shows the network parameters for that topology[3].

meaning	variable	value
INCOMING max sizes	Q_i	16,384 bytes/64 packets
OUTGOING max sizes	Q_o	8,192 bytes/64 packets
DELIVERY max sizes	Q_d	8,192 bytes/64 packets
Time to route packet	T_{rout}	3 ms (333 p/s)
Time to handle response packet	T_{resp}	4 ms (250 p/s)
Time to handle data/poll packet	T_{pack}	5 ms (200 p/s)
Time to consume data packet	T_{cons}	0 ms (∞ p/s)

Table 6.1: List of network parameters employed in the PRMP v. FF experiment.

The 50 FF receivers were allocated to the tree in no particular order; as FF does not impose any logical order among receivers, the allocation of receivers to hosts is irrelevant. In contrast,

[3]in the experiments involving the Full Feedback protocol, T_{cons} was set to 0, because flow control was not included in the protocol implementation.

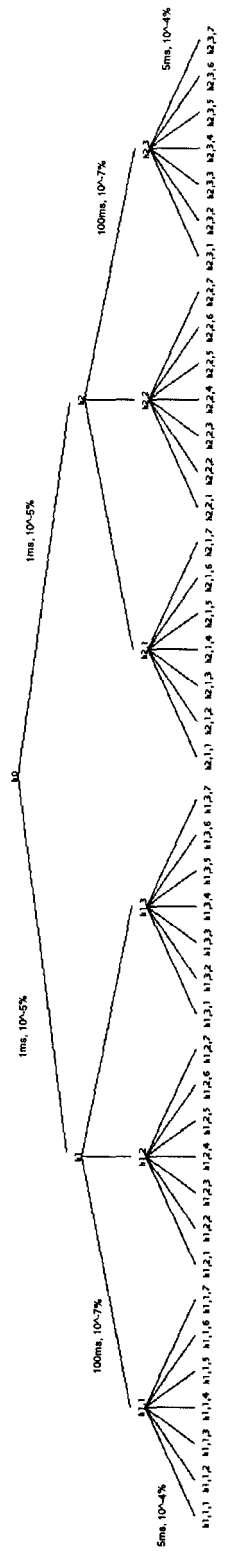


Figure 6.4: Network multicast tree employed in the PRMP v. FF experiment.

for PRMP, the logical tree *exactly matches* the physical tree. This represents the best case scenario for PRMP.

Both protocols transmitted 1MB of data using 500-byte data units ($unitSize = 500$, $DP = 2,000$), at a maximum rate of 100 packets/s ($IPG = 10$ms), and using a window of 32 KB (length $L = 64$). Additionally, PRMP employed epoch length $EL = 20$ms, $RR = 50$ responses/s, and $MTR = 20\%$. Both PRMP and FF protocols were equally tested using an "infinite window" of 2,000 packets ($L = 2,000$); the corresponding runs are denoted as PRMP-IW and FF-IW, respectively.

Table 6.2 shows the results obtained, with the values verified for I, T_1, T_2, and N. The number of losses, total and per type of loss, are also shown. Recall that congestion losses are those non-feedback packets discarded by nodes without a sender or a receiver; as in this experiment all nodes have a sender and a receiver, per definition there cannot be congestion losses (hence the column for congestion is omitted from the table). On the other hand, internal hosts are acting as network routers too; therefore, the buffer losses at these elements could be also regarded as congestion losses; congestion losses will appear masked as implosion or overrunning losses.

	Losses			Buffer Loss			
	Total	Corrupt	Buffer	I	Overr.	T_1/T_2	N
PRMP	53	53	0	0	0	399/385	1.09
PRMP-IW	47	47	0	0	0	399/385	1.09
FF	1,492,049	983	1,491,066	1,466,210	24,856	3	56.29
FF-IW	7,585,236	4,230	7,581,006	7,359,229	221,777	2	209.192

Table 6.2: Numerical results from experiment comparing PRMP to FF.

The numbers in the table show emphatically the difference between PRMP and FF: while the Full Feedback runs led to a very large number of implosion losses, which caused a "collapse" of the network, PRMP prevented *all* buffer losses in the two runs. The polling-based feedback scheme was effective in avoiding implosion in such a hierarchic configuration. Note that RR was set to 50 packets/s, which is 1/4 of the physical capacity of each host, 200 ($RR = \frac{ITR}{4}$).

Note also that the reduction in the number of implosion losses caused by window blocking in Full Feedback runs, as shown in Chapter 4, is repeated here: because FF has limited window

size, its actual transmission rate is lower than the one achieved by FF-IW, and so the feedback rate generated by receivers is smaller, causing fewer implosion losses.

In this scenario, for $DP = 2,000$ units, $unitSize = 500$ bytes, and $IPG = 10$ms, the best possible T_2 is given by:

$$T_{2_{opt}} = \frac{2,000 \times 500 \times 8}{2,000 \times 10 + 212} \Rightarrow 395 \text{ Kbps}$$

The 212 value corresponds to the the best (thus smallest) RTT_{max} possible between the source at 'h0' and any receiver at the bottom, such as 'h1,1,1'. The best RTT_{max} occurs when a packet-pair round trip finds all queues empty, and therefore are never delayed awaiting forwarding. So, in this case, RTT_{max} is the sum of link latencies: $(1 + 100 + 5) \times 2 \Rightarrow 212$.

In this network configuration without routers, with a perfect match between the *physical* and the *logical* multicast trees, the T_2 achieved by PRMP and PRMP-IW are very close to optimal (385 and 386 Kbps, respectively). Likewise, the N values for the PRMP runs are very close to the optimum N, 1, and were helped by the matched topology.

6.4 Evaluation of Anti-Nagging Mechanism

A specific network scenario was employed to evaluate the efficacy of the anti-nagging mechanism: as shown in Figure 6.5, there are only four network nodes in the tree, of which three are hosts. Host 'h0' contains the source, S; host 'h1,1' contains the single child of S, R_1; host 'h1,2' contains R_1's single child, $R_{1,1}$. The receiver $R_{1,1}$ behaves as the bottleneck in the data flow because its receiving application is very slow to consume the data received.

Recall that T_{cons} is the time it takes for the receiving application to consume a fixed-size unit of data from its local receiver. In this experiment T_{cons} is set to 100ms, so that the consumption becomes much slower than the rate in which packets are made available for consumption (up to 100 packet/s, since $IPG = 10$ms). The leaf receiver $R_{1,1}$ reports full buffers to R_1, which cannot slide its *sw* forward. Buffers at R_1 fill, and R_1 reports to S that it cannot receive any additional data packet. The parent S then keeps nagging R_1 because of $R_{1,1}$. Hence, overall $R_{1,1}$ will require extra POLLs from its parent R_1 because of flow control.

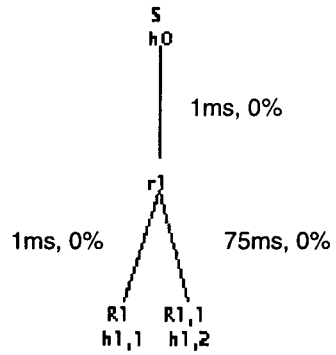


Figure 6.5: Network scenario in which the nagging parent syndrome occurs.

PRMP was run with and without the anti-nagging mechanism active. Table 6.3 presents the outputs, which are the total number of POLL packets exchanged, the network cost, N, and the throughput, T_2.

Note that T_{cons} = 100ms is the restricting factor in the maximum achievable throughput. Since a receiving application cannot consume more than 10 packets/s, the maximum T_2 will not exceed 10 packets/s, that is, 40 Kbps.

	POLL packets	N	T_2
without mechanism	36,033	22.01	39 Kbps
with mechanism on	9,193	8.59	39 Kbps

Table 6.3: Effectiveness of the mechanism to avoid the nagging parent syndrome.

The scheme to decrease the frequency in which R_1 is polled by S while R_1 waits for $R_{1,1}$ was effective in minimizing the nagging syndrome: Table 6.3 shows that the POLL packet overhead was reduced in 74% when the mechanism was active. This contributes to a lower network cost, as N was reduced with the mechanism from 22.01 to 8.59. It is clear from the column T_2 that the reduction in bandwidth achieved by the anti-nagging mechanism did not on the other hand introduce notable delays, since the mechanism did not affect the throughput of the protocol.

6.5 The IMAGINARY Tree Topology

The tree topology used in Section 6.3 may be unrealistic, because there were no network routers, and the corruption rates were higher for the links at the bottom. These factors favored PRMP and its hierarchical organization for two reasons: firstly, the tree with PRMP receivers perfectly

matched the network topology; secondly, more frequent losses at the *leaf* receivers allowed their parent receivers to recover their losses more promptly (than in FF).

A more "realistic" network configuration is described in this section, and used in the remainder of the experiments reported in this chapter. The *topology* of this new network configuration is based on the "imaginary tree" described in [Papadopoulos98], and will therefore be denoted as `IMAGINARY` configuration (see Figure 6.6). The propagation latencies of each link are shown in the figure; those omitted are equal to 1ms. The default corruption rate assigned to links was 10^{-4}% (i.e., percentage of bytes which get corrupted when traversing the link); some links have 0% of corruption losses, and these are indicated in Figure 6.6.

There are 34 hosts in the tree; the host at the root, 'h0', is to be occupied by the source (and sending application). The remaining 33 hosts are all placed at leaf positions in the tree; there are 14 routers.

The Figure 6.7 shows the allocation of the *logical* PRMP tree to the physical, network multicast tree of Figure 6.6. Note that in the tree of Figure 6.7, not a single internal receiver is located at a host node acting as a router. Given the way PRMP propagates data units (through its logical tree), there is an overhead in network cost and latency whenever an internal node forwards a data unit from one level to the next. For example, in Figure 6.7, when R_1 forwards a data unit to $R_{1,1}$, the packet traverses upstream the link to router '`r1,1`', and then downstream to $R_{1,1}$. This forwarding has an impact on T and N of PRMP, and is the price paid by PRMP for carring out the data forwarding itself.

Even though this forwarding process makes specially important the allocation of receivers to the network nodes (that is, the formation of the logical PRMP tree), the tree formation scheme is outside the scope of this thesis. The arbitrary allocation of receivers used in the `IMAGINARY` configuration is shown in Figure 6.7. For the Full Feedback protocol, its receivers do not follow any logical structure, and are sequentially allocated to host nodes (and thus the corresponding figure can be omitted).

Tables 6.4 and 6.5 show the input values employed in the `IMAGINARY` configuration, at network and protocol level, respectively.

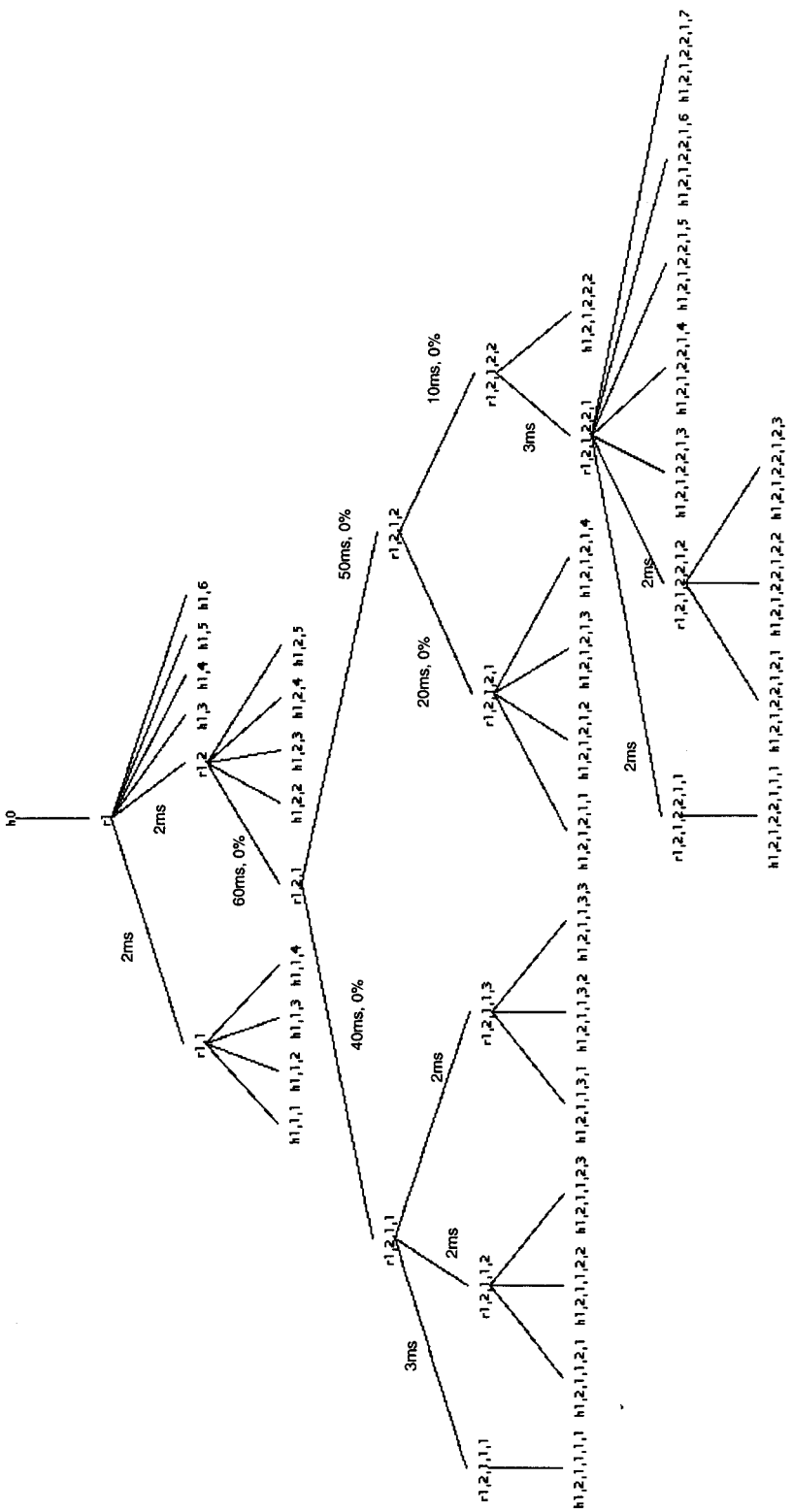


Figure 6.6: `IMAGINARY` multicast tree configuration.

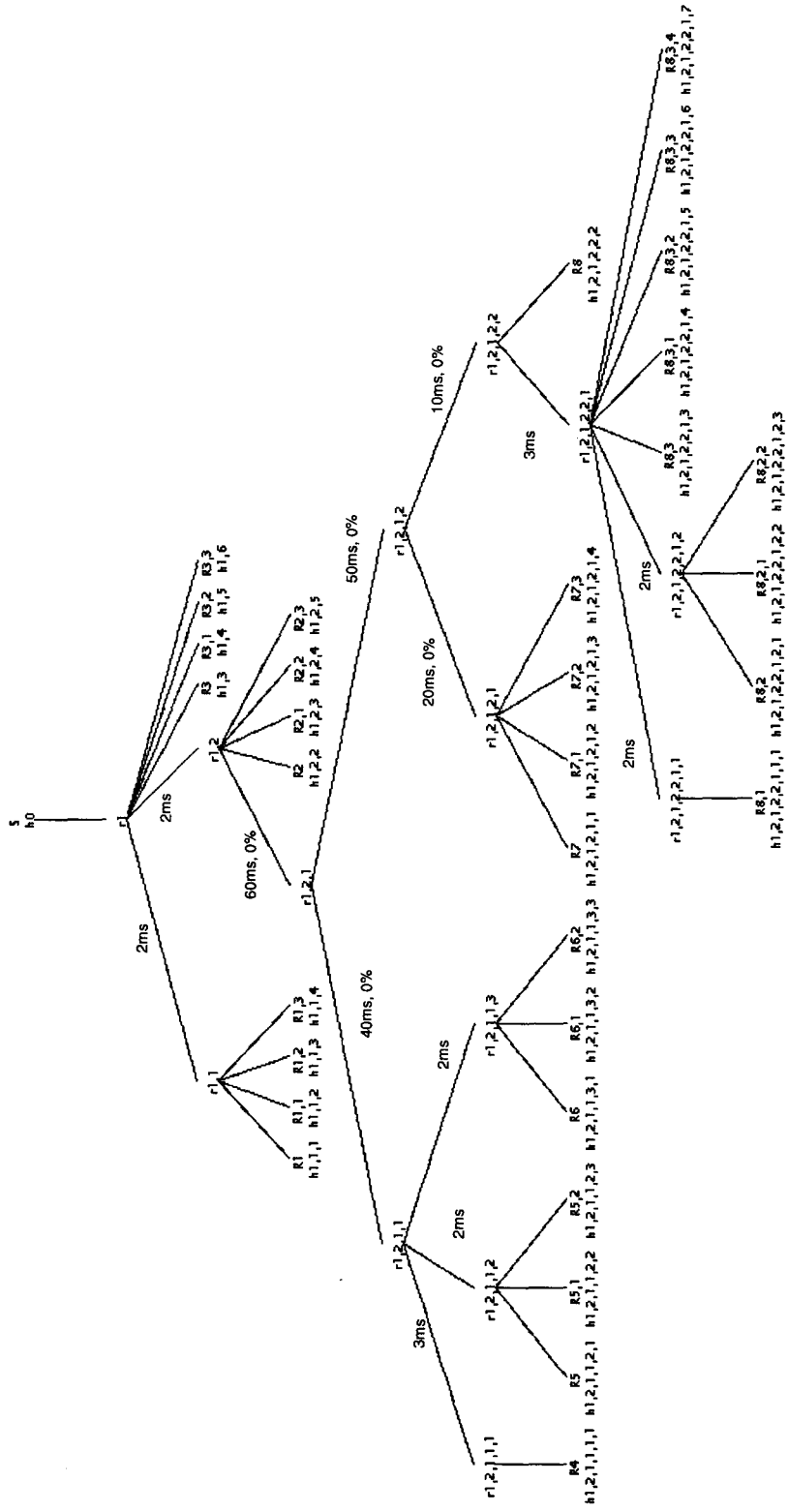


Figure 6.7: **IMAGINARY** multicast configuration with allocated PRMP source and receivers.

meaning	variable	value
INCOMING max sizes	Q_i	16,384 bytes/64 packets
OUTGOING max sizes	Q_o	8,192 bytes/64 packets
DELIVERY max sizes	Q_d	8,192 bytes/64 packets
Time to route packet	T_{rout}	2 ms (500 p/s)
Time to handle response packet	T_{resp}	4 ms (250 p/s)
Time to handle data/poll packet	T_{pack}	5 ms (200 p/s)
Time to consume data packet	T_{cons}	0 ms (∞ p/s)

Table 6.4: List of default network parameters employed in the IMAGINARY configuration.

Input	Variable Name	Value
window length	L	100 packets
data unit size	$unitSize$	500 bytes
transmission size	DP	4,000 packets
group size	GS	33 receivers
inter-packet gap	IPG	10ms
epoch length	EL	20ms
response rate	RR	100 RESP/s
uni v. multicast threshold	MTR	20%

Table 6.5: Protocol inputs used in the experiments with the IMAGINARY configuration.

6.6 Congestion Control Evaluation

This section describes an experiment to evaluate the dynamics of the congestion control mechanism in face of fluctuations of the network load. To isolate congestion losses, the corruption rate of all links was set to 0% (thus there were only buffer overflow losses). The source transmits 4,000 units of 500 bytes each.

An artificial load was induced into a router during the transmission, as follows. The first 799 packets are transmitted by S normally; at the time the packet $seq = 800$ is sent, the router 'r,1,2,1,2,2,1,2', as shown in Figure 6.8, has its load increased 4 times: $Q_i \leftarrow \frac{Q_i}{4}$, $Q_d \leftarrow \frac{Q_d}{4}$, and $Q_o \leftarrow \frac{Q_o}{4}$; further, $T_{rout} \leftarrow T_{rout} \times 4$, so that there are four times less buffers available in all, and packets take four times longer to be routed. All stored packets remain in the router and are forwarded normally; the new, shorter buffer size limit only applies to newly arrived packets.

The router remains the bottleneck until the source transmits the packet $seq = 1,300$, when the capacity goes back to the original level, and is kept normal during the remaining 2,700 data

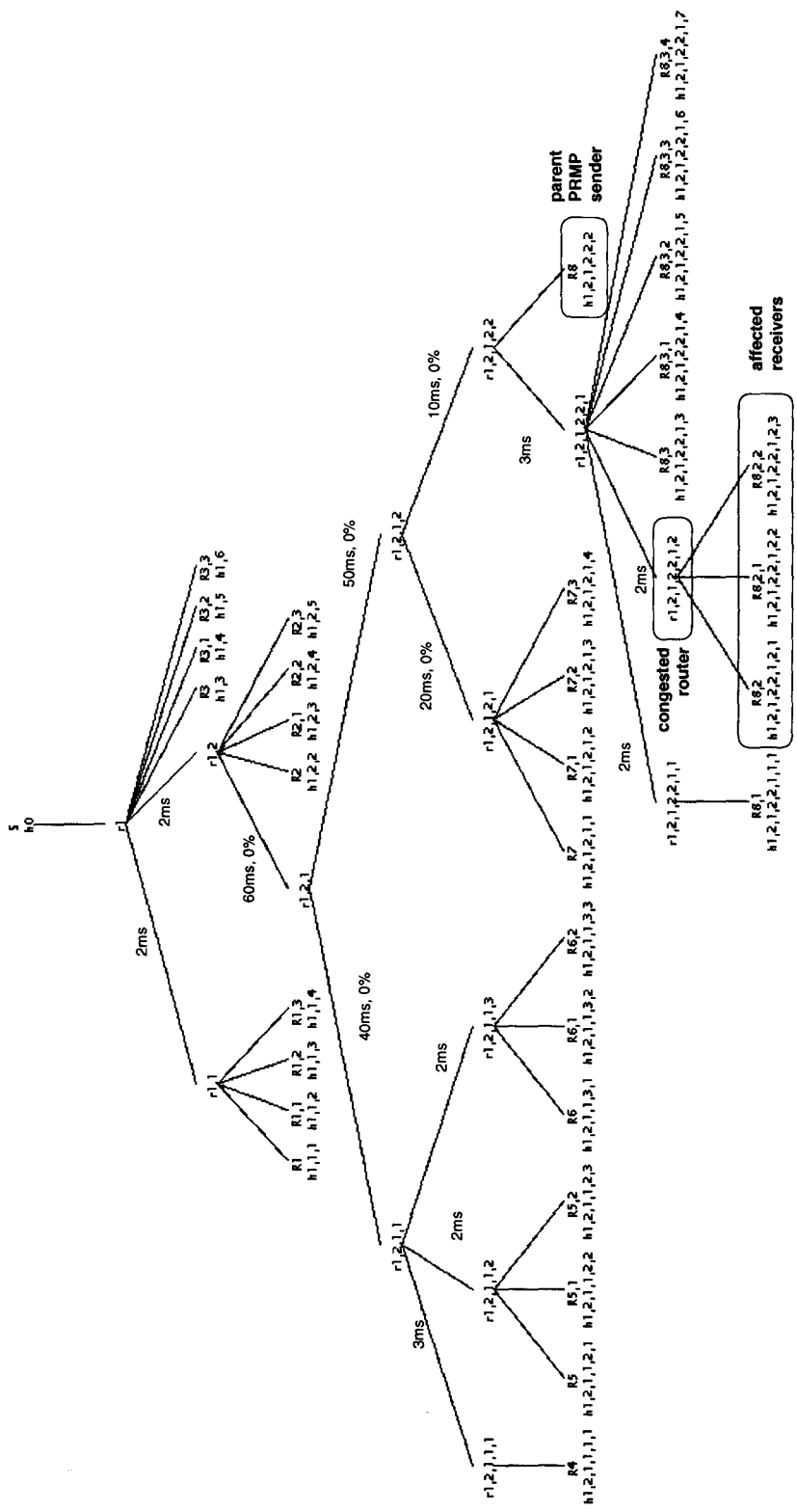


Figure 6.8: IMAGINARY multicast configuration with allocated PRMP source and receivers.

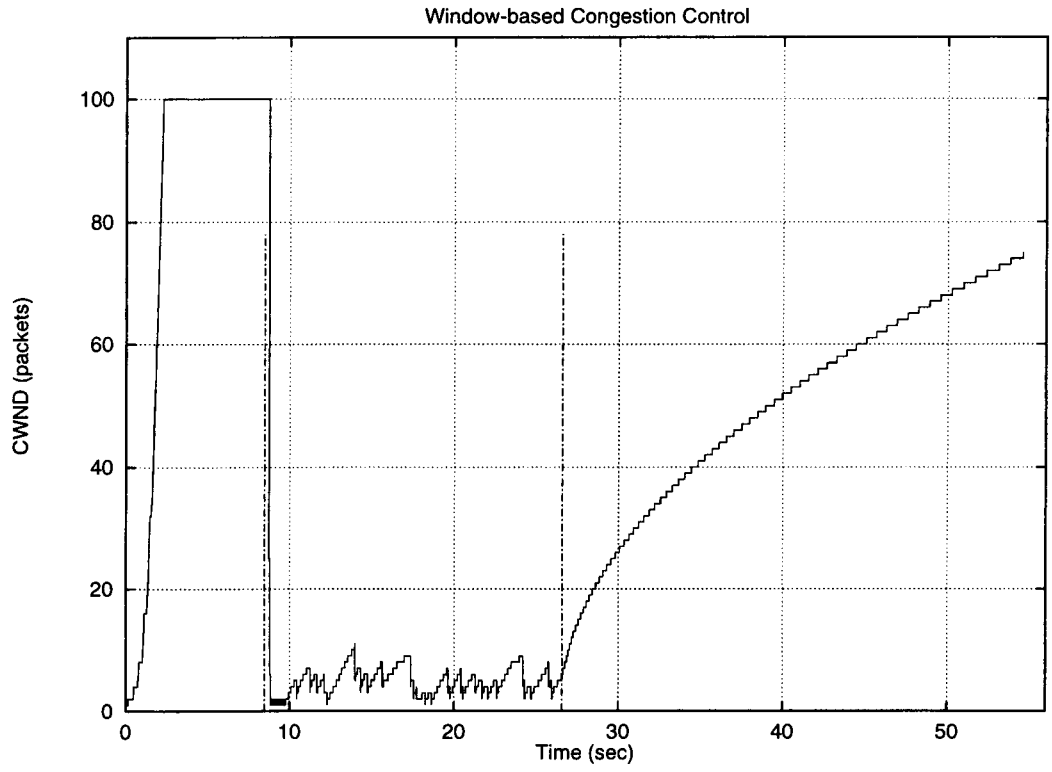
packets, i.e., until the end of the transfer. This experiment was run for both window-based (*wb*) and rate-based (*rb*) congestion control mechanisms, for the `IMAGINARY` configuration.

Figures 6.9.(a) and 6.9.(b) depict the *wb* congestion control mechanism in action, and show the fluctuation of its control variable, the congestion window, through time. The *sw.cwnd* shown in the figure is that of the sending role of R_8: R_8 is the parent of the receivers which will experience packet losses when congestion is induced. It is R_8 that forwards packets to $R_{8,2}$, which on its turn forwards data packets through the congested router to its own children, $R_{8,2,1}$ and $R_{8,2,2}$. Thus, $R_{8,2}$ will also detect and reduce the load to reduce congestion. Figure 6.9.(b) is a zoom into the time interval in which congestion was induced, and also indicates packets losses at the congested router.

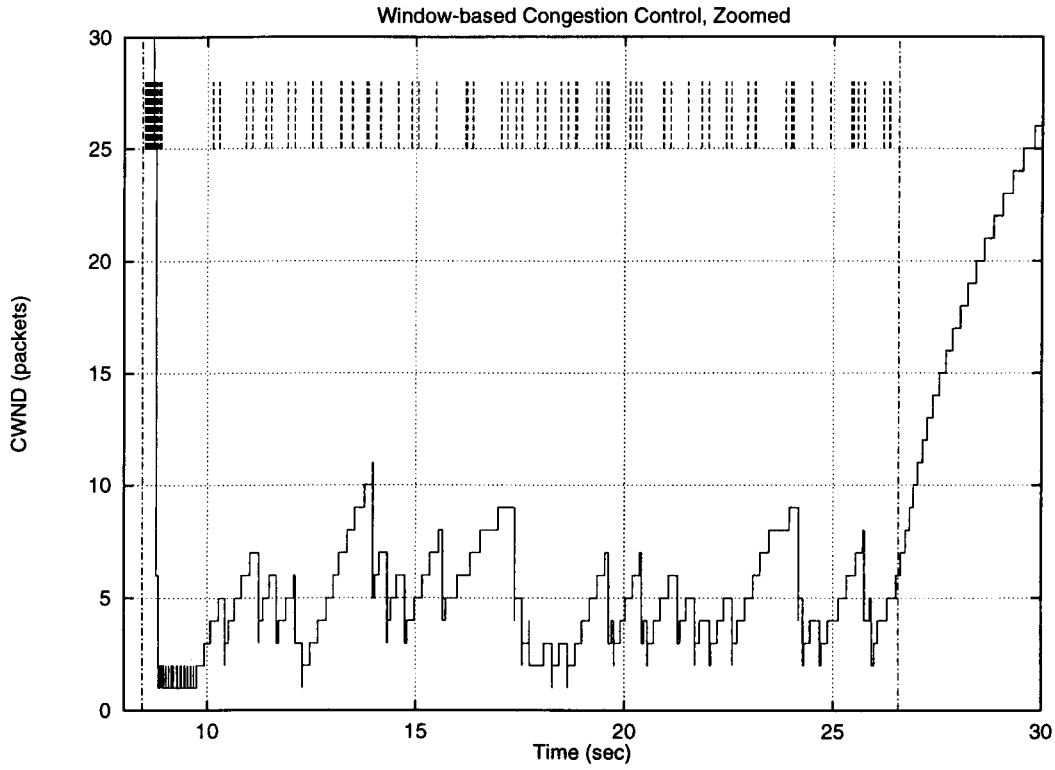
The mechanism starts in "slow-start": *sw.cwnd* increases exponentially from 1 until the limit L is reached, at around 2s. *sw.cwnd* remains unchanged until 8s, when the induced congestion at the router starts. Note that the concentration of vertical lines in Figure 6.9.(b) shows the large number of losses verified in the router. These multiple initial losses make *sw.cwnd* rapidly drop to 2 or 3 packets. While the router stays congested, *sw.cwnd* typically varied between 2 and 8 packets. Near 27s the router is relieved from congestion, and *sw.cwnd* starts growing with additive increase until the transmission ends, at time 57s. In this example, the transmission does not last long enough to allow the *sw.cwnd* of R_8 to reach its maximum, L.

Figures 6.10.(a) and 6.10.(b) show the variation in value of the control variable of the *rb* mechanism, the inter-packet transmission gap, through time. Like in the previous example, the congestion control shown refers to the sending role of R_8 (the value of IPG at R_8); 6.10.(b) is a zoom into the interval in which congestion was induced.

The "sawtooth" pattern which is typical of *sw.cwnd* is also present in the *rb* scheme. The IPG starts at the provided default of 10ms and remains equal to 10ms until congestion is induced at the router at around time 3s; note that congestion starts early in this example because *rb* does not employ a slow-start; instead, it starts transmitting using the default IPG = 10ms (IPG_{min} was set to 10ms). Therefore, S will transmit packet $seq = 800$ earlier, and thus induced congestion will start earlier than in the window-based scheme.

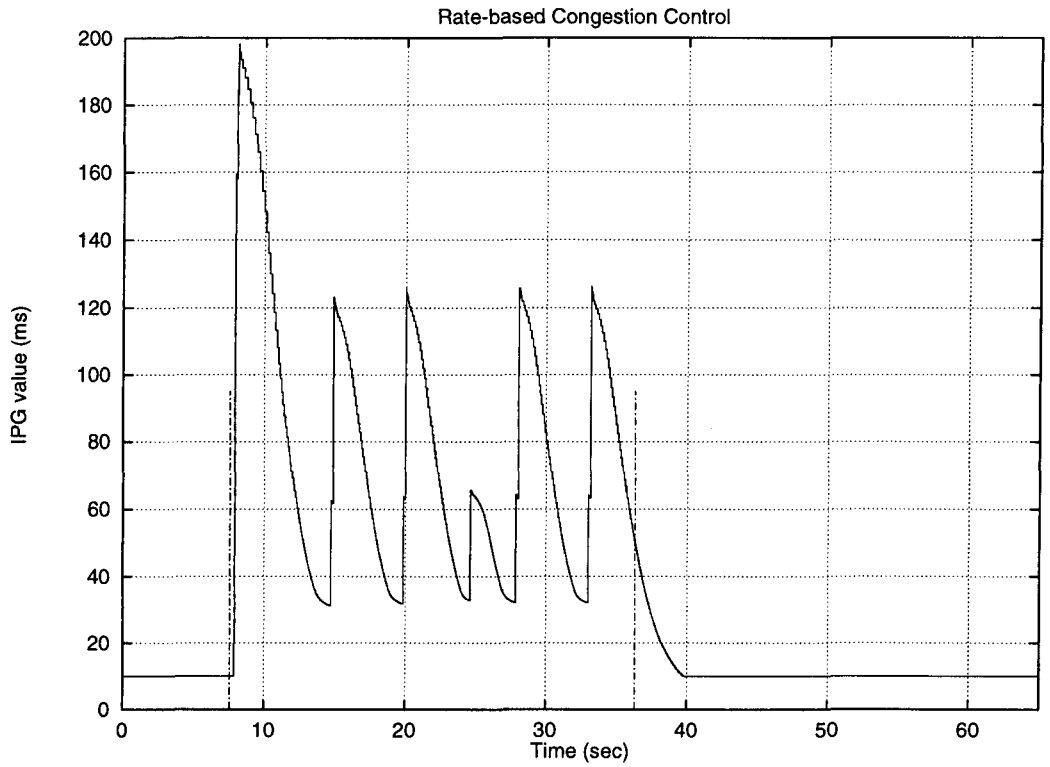


(a) complete

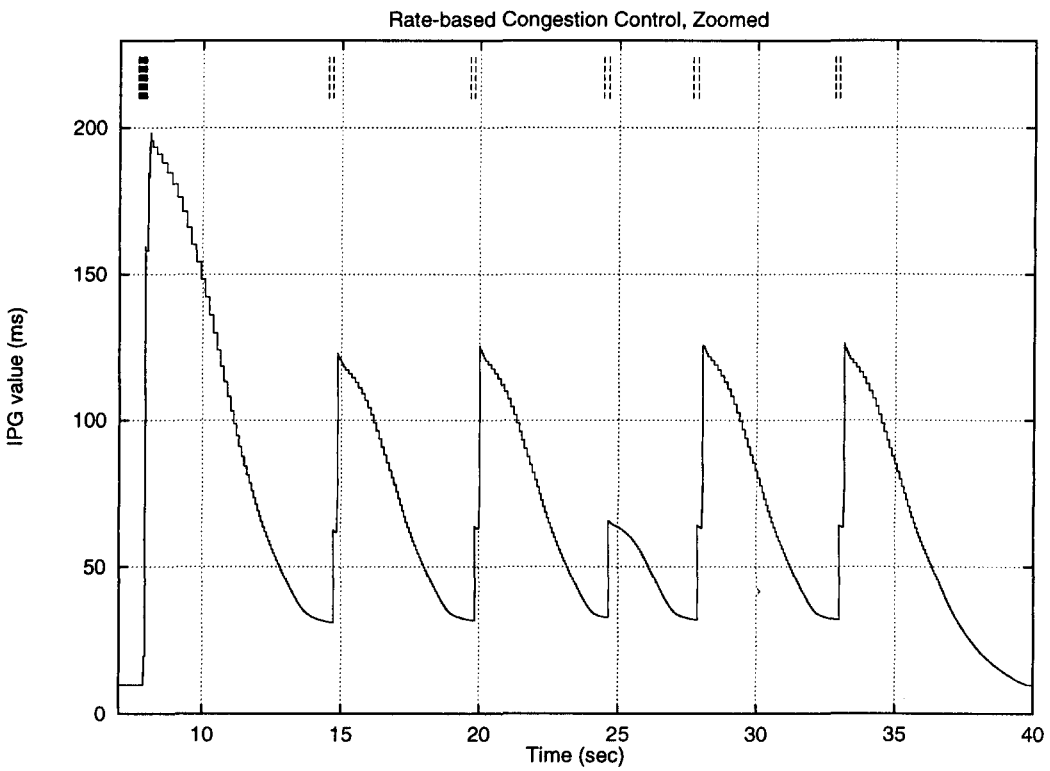


(b) zoomed

Figure 6.9: Variation of $R_8.sw.cwnd$ in time with induced congestion (packet losses are marked at the top in Figure (b)).



(a) complete



(b) zoomed

Figure 6.10: Variation of IPG in time with induced congestion; (packet losses are marked at the top in Figure (b)).

When induced congestion starts, there are multiple consecutive losses within a transmitted window (like in the window-based example), which makes the IPG jump to 200ms. Then the transmission rate is low enough to make losses stop, and the value of IPG starts decreasing. It decreases until IPG = 30ms, when the network capacity is exceeded and provokes 1 or 2 new packet losses. The pattern is repeated five times: quick decrease in rate, slow increase in rate, and eventually, losses. At around time 36s, the router is relieved from congestion, while the IPG is on decrease. There are no further losses, and at time 40s IPG reaches IPG_{min} that is, the maximum transmission rate set by the user.

Table 6.6 compares numeric results for three settings regarding congestion control. It shows the number of buffer losses experienced at the congested router during the entire transmission, as well as the throughputs and the number of POLL packets exchanged.

Congestion Control	buffer losses at router			T_1/T_2	POLLs
	total	congestion	impl.		
None	2,256	2,115	141	304/282	1,263
Window-Based	116	111	5	295/274	9,893
Rate-Based	41	37	5	244/241	1,598

Table 6.6: Numeric results obtained for transmissions using different congestion control mechanisms.

With either congestion control mechanism in place, it is evident the congestion mechanism of PRMP prevent congestion losses at the bottleneck router: 111 losses for *wb*, and 37 for *rb*, in contrast to 2,115 losses without any mechanism. There are few implosion losses in all 3 cases, but they were noticeably smaller with *wb* or *rb* active.

The throughput of PRMP is affected by the congestion control mechanisms: the protocol slows down, instead of trying to keep up with transmission. Therefore, T_1 and T_2 of "greedy" PRMP (*without* congestion control) result in slightly higher throughput, despite incurred losses.

Finally, note that the number of POLL packets required is substantially increased in the *wb* case. This poll overhead, as previously discussed, is due to the window blocking effect (see Section 4.2): the window blocks, one or more receivers need to be polled, and thus when their next poll plan is due, a POLL packet is sent to elicit the required feedback. Therefore, the rate-based mechanism may not tackle congestion so efficiently; however, POLL packets tend to be much smaller than data packets, and therefore do not contribute as much with an increase

in the load.

6.7 Flat v. Hierarchic

The final set of experiments provides a comparison between the basic, flat PRMP, and the extended, hierarchic PRMP. It is well-known that the hierarchic schemes offer potential gains in terms of scalability of reliable multicast protocols; this section reports experiments which quantify such gain for the same `IMAGINARY` configuration which has been employed in the previous sections. An additional, alternative case was studied where the corruption rate of all the links downstream of router '`r1,2,1,2,2,1`' were increased (as illustrated in Figure 6.11); this scenario was studied to highlight the potential gain stemming from local recovery when the losses are concentrated at lower levels of the tree. Its purpose was to evaluate the potential gains in T and N arising from the use of distributed error and flow control.

Recovery times can be reduced when recovery is provided by nearby agents, avoiding long delays in reporting a loss to the source and in obtaining a retransmission from it. To realize the gain of local recovery, however, losses need to be concentrated at the "bottom" levels of the tree. Further, for PRMP, the gain is only possible if the point of loss is between the source and a parent node: in this case, the child receivers (and their receivers too, if any) will profit from local recovery. To measure this advantage, the *average* and *maximum loss recovery times*, from the receivers' point of view, were also recorded. Table 6.7 summarizes results. The window-based congestion control scheme was used in both cases.

	normal		lossy	
	flat	*hierarchic*	*flat*	*hierarchic*
POLL packets	11,648	9,079	18,812	11,440
throughput T_1/T_2 (Kbps)	76/76	202/202	43/43	172/171
network cost N	3.816	3.853	5.518	4.134
avg. recovery latency	285	124	345	87
max. recovery latency	457	300	737	507

Table 6.7: Comparison between hierarchic and flat allocation of receivers.

First note that, as expected, the overall results are consistently better in the "normal" configuration than in the "lossy": less POLL packets were required, higher T achieved, lower

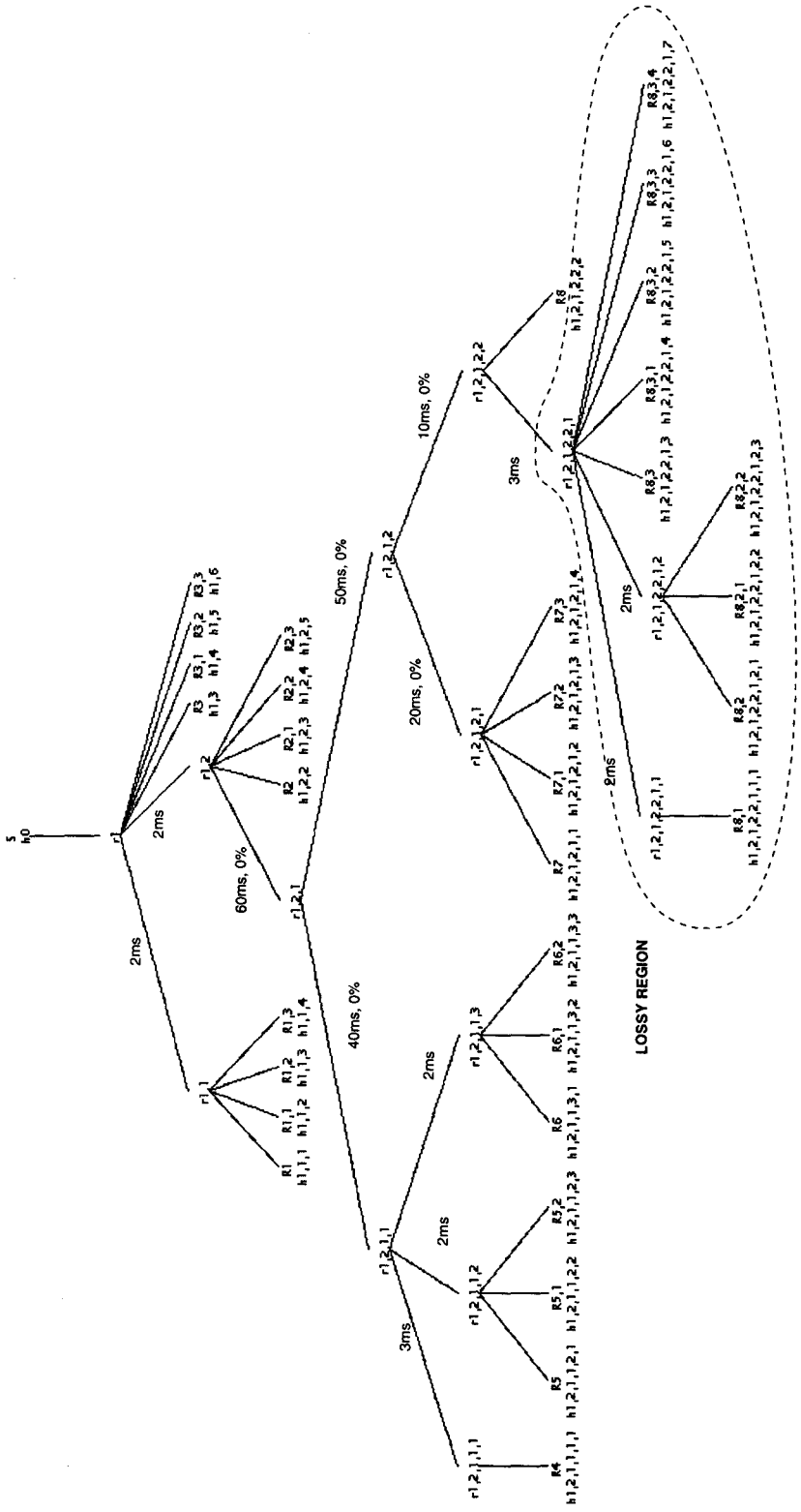


Figure 6.11: Scenario with some lossy links at lower levels of the tree

N, and faster loss recovery. In each configuration, in general the hierarchic PRMP exceeded the flat PRMP. The number of POLL packets required by hierarchic PRMP was around 22% lower (i.e., better) in "normal", and 39% lower in the "lossy" scenario; this is due to the fact that the latencies between a sender and a receiver in the hierarchic PRMP are much smaller, and thus sw blocks less often. This shows on the throughput, which is significantly superior in the hierarchic PRMP for both scenarios. Localized error, flow, and congestion control is the chief reason for such gains. The N of hierarchic was not substantially better than that of the flat case; this is probably because of the overhead caused by the "detour" of data packets which are forwarded through the PRMP tree. In the flat case, data packets flow downstream through the tree and never traverse the same link twice; in the hierarchic case, the data packets are forwarded at protocol level, which implies some overhead. Finally, both the average and the maximum recovery times recorded were better with the hierarchic PRMP: in the "normal" scenario it was around 2/3 of the corresponding value in the flat, while in the "lossy" scenario the advantage of local recovery was even more evident: the average recovery time for receivers with the hierarchic PRMP was 1/4 of that verified in the flat PRMP.

Chapter 7

Concluding Remarks

There has recently been substantial research in the field of scaleable reliable multicast protocols. One of the main issues discussed is the *implosion feedback* problem. It limits the scalability of reliable multicast protocols. Novel schemes for scaleable loss detection and recovery have been devised, resulting in the development of new reliable multicast protocols. Most of these protocols follow the same approach to solve the scalability limitation: *make the sender independent from receivers.*

This thesis follows a radically different approach, by tackling the scalability problem through a polling-based implosion avoidance mechanism. It shows that the initial assumptions about sender-initiated protocols (according to [Pingali94]) need not be true, and can be circumvented by controlling the rate of feedback returned by receivers. The scalability of the protocol described in this thesis, PRMP, and its mechanisms is further enhanced by structuring receivers according to a hierarchic organization (i.e., a tree-based scheme).

7.1 Synopsis

Multicast communication allows the efficient transmission of packets to potentially large sets of receivers. Packets are propagated through a multicast tree, and each packet crosses a given edge only once before reaching a receiver. The IP multicast architecture allows multicast transmission in the Internet in a scaleable manner, but such transmission is unreliable, and packet losses are likely to occur. For many applications, some degree of packet loss is tolerable;

examples are applications that broadcast events through the Mbone. For others, the contents transmitted by the sending application have to reach receiving applications *integrally*: receiving applications need to receive an *exact* copy of the data transmitted. This is known as fully-reliable multicasting.

In certain uses of multicast, the sending application transmits data unaware of receiving applications: the sending application "announces" in a session directory the starting time of a transmission and the "channel" used (the multicast group id); interested receivers join the group at the scheduled transmission time. This is like the event broadcasting example. In other cases, the sending application wishes to determine the *exact* list of receivers, and possibly to keep track of any changes in the membership which may occur during the transmission (known as sender-reliable multicasting). This is important, for example, whenever the sending application wishes to "bill" receiving applications for the contents transmitted.

This thesis sets out to design a fully-, sender-reliable multicast protocol that can scale well in terms of both number of receivers and their geographical distribution (wide-area networks, sparse distribution, heterogeneous latencies, etc.). The scalability limitation to be overcome by this protocol and others is the ACK-implosion problem. Implosion happens in reliable multicast when the amount of feedback packets returned by receivers exceeds the sender or network capacity, leading to packet losses and, hence, loss of throughput and increase in network cost.

To solve the ACK-implosion problem, a new class of protocols, the receiver-initiated protocols, has been designed. [Pingali94] has shown that traditional sender-initiated schemes, where the sender expects to receive an ACK packet from each receiver for every data packet multicast, scale poorly because of the ACK-implosion problem. Receiver-initiated protocols, in contrast, shift the initiative for loss detection and recovery to receivers: the sender is "passive" and after transmitting, it waits for receivers to request retransmissions. This makes the sender independent of the number of receivers, and allows a protocol of such a class to scale to very large groups.

However, receiver-initiated protocols have their limitations, too, and they result from the fact that the sender does not have control of or knowledge about receivers. Firstly, implosion is still possible, when losses are correlated and shared by large sets of receivers (i.e., NACK-

implosion). Secondly, as the sender does not receive positive confirmation of receipt from receivers, it does not know when it can "safely" release a data packet from its buffers. Therefore, it either cannot guarantee (within reasonably probability) full-reliability, or it has to store all data packets transmitted during the transmission and afterwards for an arbitrarily long time (i.e., rely on the "infinite buffer assumption"). Thirdly, because the sender does not know about receivers, heuristic inputs (e.g., maximum RTT) are often required; conservative values may need to be used, negatively affecting the protocol. Consequently, error control and flow control mechanisms suffer due to the lack of knowledge about receivers. Furthermore, because the design of sender and receiver initiated protocols differ radically, to "*add the knowledge about receivers*" to the sender of a receiver-initiated protocol will not automatically bring the benefits of having such knowledge at the sender.

This thesis develops an alternative approach to solve the scalability limitation of sender-initiated protocols: the use of a polling-based implosion avoidance mechanism. This idea is not new [Hughes94], although it has only been applied within the context of local-area networks. The polling-based implosion avoidance mechanism described in this thesis extends that concept: receivers are polled at *carefully-planned times*, so that despite heterogeneous sets of RTTs, the rate of feedback packets arriving at the sender does not exceed the sender or network's capacity. The sender can therefore remain active in the role of controlling the communication with receivers (which is achieved through a multicast sliding window). The design of PRMP is based on a *one-to-many sliding window mechanism*, and error, flow, congestion and session control all derive from it.

The resulting protocol is called PRMP: Polling-based Reliable Multicast Protocol. Its design process has been divided into two distinct stages: *flat* and *hierarchic*. The flat PRMP (Chapter 3) was developed so that high-performance multicasting could be delivered to "medium-sized" groups (e.g., hundreds of receivers). The hierarchic PRMP (Chapter 5) is the extension that improves the scalability both in terms of group size and geographical distribution of receivers.

In the flat PRMP, the sender directly communicates with all receivers. The protocol at the sender is scaleable due to its polling-feedback implosion avoidance mechanism. Its flow control mechanism prevents unnecessary packet losses caused by overrun receivers (the sender

has status about receivers and limits the transmission speed accordingly). The error control mechanism collects retransmission requests and judiciously decides between selective unicast and multicast retransmission (as discussed in Section 2.2.1, this is an important feature for the scalability of reliable multicast protocols). Furthermore, the error control mechanism prevents the unnecessary retransmissions that would arise from "obsolete NACKs". As shown in the simulation experiments, the end result is a scalable reliable multicast protocol which can deliver high throughput with low network cost.

However, the number of receivers in which the flat PRMP protocol can achieve such good results is limited. There are two factors that restrict the scalability of flat PRMP. The first factor is that the capacity of the sender and the network in dealing with feedback is *finite*, so that the rate of feedback packets available to the sender is limited. For a given network, there will be a group size that is sufficiently large to reach this limit; thereafter the response rate will become the bottleneck in the transmission. The second factor is that flat PRMP cannot explore the topology. In wide-area multicasting, the round-trip delays between sender and receivers can be large; tree-based or hierarchic protocols employ a "representative receiver" that is able to recover losses more quickly by retransmitting to nearby receivers. Additionally, the hierarchic organization allows the protocol to scale better: only a subset of receivers communicates directly with the sender, reducing the risk of implosion, and the amount of state kept at the sender.

The hierarchic PRMP extension was developed to increase the scalability of the flat PRMP. It has a fully-hierarchic nature, more so than other tree-based protocols: in PRMP, data packets are forwarded by the protocol receivers using multicast transmission between each two successive levels. This fully-hierarchic nature is reflected on the design of the protocol: error control, flow control, etc. are *fully distributed* among members of the group. To provide reliable multicast in a "network-conscious manner", two congestion control mechanisms were developed for PRMP: a window-based and a rate-based, both detecting congestion through aggregated packet status in the sliding window.

The simulation in Chapter 6 extended the comparison between PRMP and the Full Feedback protocol (FF) shown in Chapter 4. It confirmed the previous results: while FF scales poorly because of implosion losses, PRMP can efficiently prevent such losses and, with the help of

its hierarchic organization, achieve high throughput and low network cost. The benefits of distributed error and flow control are visible in the experiments performed with multicast trees, and hierarchic PRMP outperformed flat PRMP in all aspects apart from network cost, in which case they were on par. The network cost is not improved with hierarchic PRMP (over the flat one) due to the nature in which data packets are transmitted from the sender to receivers in the hierarchic PRMP: they are *forwarded by receivers*, overlapping with the more efficient network-level routing. The simulation experiments also showed that both the anti-nagging and congestion control mechanisms are effective in reducing the number of POLLs and congestion losses, respectively.

7.2 Contribution

Sender-initiated protocols were shown to be poorly scaleable due to the ACK-implosion problem. This thesis explored the idea of making sender-initiated schemes scalable through a polling-based, sender-controlled implosion-avoidance mechanism. It demonstrated the feasibility of this approach through the design of an efficient and scalable reliable multicast protocol. The mechanism behind the scalability of the protocol is a novel implosion avoidance mechanism based on the *timely planning of polls*.

The implosion avoidance allows the sender to maintain status about receivers; a novel one-to-many sliding window scheme has been devised. It introduces the concept of *receiver sets* and *aggregation of attributes*, and with these extended the abstraction of reliable unicast to reliable multicast. The design of novel error, flow, and congestion control mechanisms has been based on the aggregated attributes as well as receiver sets that are generated from the status stored in the sending window. As mentioned above, the error control mechanism of PRMP efficiently detects and recovers from packet losses; the flow control prevents overrunning of receivers; the congestion control reduces the load and substantially reduces the number of losses caused by buffer overflow at congested routers.

A multi-threaded architecture was designed to implement the resulting protocol, and a network tool to simulate and evaluate its implementation. The simulation experiments illustrated that the PRMP can achieve high throughput and low network cost regardless of the group size.

7.3 Future Work

This thesis described in detail the most important mechanisms of PRMP, including a prototype implementation; however, important issues regarding PRMP were not addressed. Both session and congestion controls require more work. In particular, with respect to session control, it is necessary to investigate the protocols and criteria behind the formation of the PRMP logical tree. This is an important issue for any tree-based reliable multicast protocol, but even more so for PRMP, considering its fully hierarchic structure.

Congestion control also requires more investigation, with evaluation of network scenarios involving multiple unicast and multicast flows. Fairness among flows has not been addressed. Further, the loss-based, congestion detection mechanism can be improved: packet losses may not be due to congestion; also, the scheme is reactive, not proactive, and only reacts when congestion has already started. Imminent congestion can be detected and avoided, for example, by keeping track of RTTs (e.g., *Tri-S* [Wang91]).

The PRMP protocol engine at the sender is complex. Even though the number of feedback packets transmitted to the sender is greatly reduced by the polling scheme, the work required to handle each response packet is substantially higher than with ordinary ACK packets (and so is the bandwidth). Additionally, the planning mechanism of PRMP requires *fine-grain* timers, so that it can uniformly implement a response rate through time. For example, if the transmission rate is low, the interval between the transmission of two successive polling requests increases, and with it the granularity of polling (i.e., receivers polled by a single packet); this makes undesirable peaks in the response rate more likely. Therefore, it would be interesting to investigate a scheme with two $IPGs$, one for transmission of data packets, and another for transmission of polling requests.

Finally, there are issues to be studied regarding the hierarchic structure of PRMP. Consider the rate at which data packets are made available for transmission at an internal node; if the parent is slow to transmit (e.g., due to a large IPG, or small window, etc.), it is possible that an internal node will receive a data packet, forward it immediately, and then block without further data to transmit. As there is no other data to transmit, the internal node sends POLL packets to elicit needed responses from its receivers; this may happen periodically, leading to

a notable increase in the number of POLL packets and hence in the network cost. This is particularly likely to happen in the rate-based congestion control mechanism. One potential alternative to be investigated is the multicast of data packets *from the source to all receivers*, as in other reliable multicast protocols.

Finally, the current prototype, which runs on top of a simulated network environment, needs to be implemented and tested in real networks. For the forwarding of data packets between successive levels, multiple IP multicast groups can be used. But before, a formal specification of the protocol architecture, including the identification of real-time constraints, is recommended.

Bibliography

[Ammar92] Ammar, M. H., Wu, L. R. "Improving the Performance of Point to Multi-Point ARQ Protocols through Destination Set Splitting, " Proceedings of IEEE INFOCOM '92, Florence, Italy, May 1992, pp 262-271.

[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

[Bhagwat94] P. Bhagwat, P. P. Mishra, S. Tripathi, "Effect of Topology on Performance of Reliable Multicast Communication", INFOCOM'94.

[Birman91] K. Birman, A. Schiper, P. Stephenson. "Lightweight Causal and Atomic Group Multicast", February 1991. ACM Transactions on Computer Systems, v.9, n.3, August 1991, pp 272-314.

[Birtwistle73] G.M.Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard, "Simula Begin", Petrocelli/Charter, New York, 1973, 391p.

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

[Cheung95] Cheung, S. Y., Ammar, M. H., "Using Destination Set Grouping to Improve the Performance of Window-controlled Multipoint Connections, " Computer Communications Journal, Vol. 19, 1996, pp723-736. (Also in, Proceedings of International Conference on Computer Communication and Networks , Las Vegas, Nevada, September, 1995, pp388-395.)

[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.

[DeLucia97] D. DeLucia, K. Obraczka, "Multicast Feedback Suppression Using Representatives", IEEE INFOCOM'97, Kobe, Japan, 17-11 April 1997.

[Floyd93] S. Floyd, V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, v.1, n.4, Aug. 1993, pp. 397-413.

[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.

[Holzmann91] G. J. Holzmann, "Design and Validation of Computer Protocols", Prentice-Hall Software Series, 1991, 500p.

[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.

[Jones91] M.W. Jones, S. Sorensen, S. Wilbur, "Protocol design for large group multicasting: the message distribution protocol", Computer Communication, v.14, n.5, June 1991.

[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

[Macedo94] R.J.A. Macedo. "Fault-Tolerant Group Communication Protocols For Asynchronous Systems". Ph.D. Thesis, Dept. of Computing Science, University of Newcastle upon Tyne, 1994.

[McCanne96] S. McCanne, V. Jacobson, M. Vetterli, "Receiver-driven Layered Multicast". ACM SIGCOMM, August 1996, Stanford, CA, pp. 117-130.

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

[Mitrani82] I. Mitrani, Simulation Techniques for Discrete Event Systems, Cambrisge Computer Science Texts 14. Cambridge University Press, 1982.

[Moser96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System," Communications of the ACM, April 1996.

[Nonnemacher97] J. Nonnemacher, E. W. Biersack, "Asynchronous Multicast Push: AMP", Proc. of ICCC'97 International Conference on Computer Communications, Cannes, France, Nov. 1997.

[Nonnemacher97b] J. Nonnemacher, E. W. Biersack, D. Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission", Proc. of ACM SIGCOMM'97, Cannes, France, Sept. 1997, pp.289-300.

[Nonnemacher97c] J. Nonnemacher, E. W. Biersack, D. Towsley, "How Bad is Reliable Multicast without Local Recovery?", Proc. of IEEE INFOCOM'98, San Francisco, USA, April 1998, pp.972-979.

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

[Papadopoulos95] C. Papadopoulos, G. Parulkar, "Implosion Control for Multipoint Applications", In Proc. of 10th Annual IEEE Workshop on Computer Communications, Rosario Resort, USA, Sept. 1995.

[Papadopoulos98] C. Papadopoulos, G. Parulkar, and G. Varghese, An Error Control Scheme for Large-Scale Multicast Applications, INFOCOM'98, San Francisco, 28 March-2nd. April 98.

[Paul94] S. Paul, K. Sabnani, and D. Kristol, "Multicast Transport Protocols for High-Speed Networks", Proc. of the IEEE Intl. Conf. on Network Protocols, p.4-14, 1994.

[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.

[Paxson97] V. Paxson, End-to-End Internet Packet Dynamics, IEEE INFOCOM'98, San Francisco, California, 29th March-2nd April 1998, pp.139-152.

[Peterson96] L. L. Peterson, B. S. Davie, "Computer Networks: A Systems Approach", Morgan Kaufmann Publishers, San Francisco, USA, 1996, 552p.

[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.

[Rajagopalan93] B. Rajagopalan, "A Mechanism for Scalable Concast Communication", Computer Communicatons, v.16, n.8, Aug. 1993.

[Saltzer84] J. H. Saltzer, D. P. Reed, D. D. Clark, "End-To-End Arguments in System Design". Transactions on Computers, v.2, n.4, pp. 277-288, 1984 .

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

[Speakman98] T. Speakman, D. Farinacci, S. Lin, A. Tweedly, Pretty Good Multicast Transport Protocol Specification, RFT Internet Draft, 8 January 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.

[Talpade95] R. Talpade, M. Ammar, "Single Connection Emulation: An Architecture for Providing a Reliable Multicast Transport Service", in Proceedings of the 15th IEEE Intl Conf on Distributed Computing Systems, Vancouver, June 1995.

[Towsley87] D. Towsley, S. Mithal, "A Selective Repeat ARQ Protocol for a Point-to-Multipoint Channel," IEEE INFOCOM'87, 6th Annual Conference, San Francisco, March 31 - April 2, 1987.

[Turletti94] T. Turletti, J.C. Bolot, I. Wakeman, "Scalable Feedback Control for Multicast Video Distribution in the Internet", In Proc. of SIGCOMM'94, London, 31 Aug-2 Sept, 1994.

[Vicisano98] L. Vicisano, L. Rizzo, J. Crowcroft, "TCP-like Congestion Control for Layered Multicast Data Transfer", Proc. of IEEE INFOCOM'98, San Francisco, USA, April 1998, pp.996-1003.

[Wang91] Zheng Wang and Jon Crowcroft, "A New Congestion Control Scheme: Slow Start and Search (Tri-S)," ACM Computer Communication Review, vol. 21, pp. 32-43, Jan. 1991.

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

[Yavatkar95] R. Yavatkar, J. Griffioen, M. Sudan, "A Reliable Dissemination for Interactive Collaborative Applications", ACM Multimedia'95.

[Yavatkar95b] R. Yavatkar, J. Griffioen, "Reliable Dissemination for Large-Scale Wide-Area Information Systems", Proceedings of the 3rd. IEEE Workshop on the

Architecture and Implementation of High Performance Communication Subsystems (HPCS'95), Aug. 1995.