

# An End-to-End Reliable Multicast Protocol

## Using Polling for Scalability

Marinho P. Barcellos\* and Paul D. Ezhilchelvan

Department of Computing Science, University of Newcastle upon Tyne

Newcastle upon Tyne, NE1 7RU - UK

{A.M.P.Barcellos@ncl.ac.uk, P.D.Ezhilchelvan@ncl.ac.uk}

Phone: +44-191-477-6553 - Fax: +44-191-222-8232

### Abstract

Reliable sender-based one-to-many protocols do not scale well due mainly to implosion caused by excessive rate of feedback packets arriving from receivers. We show that this problem can be circumvented by making the sender poll the receivers at carefully planned timing instants, so that the arrival rate of feedback packets is not large enough to cause implosion. We describe a generic end-to-end protocol which incorporates this polling mechanism together with error and flow control mechanisms, and an architecture that implements it. We analyse the behaviour of our protocol using simulations which indicate that our scheme can be effective in minimising losses due to implosion, and achieve high throughput with low network cost.

**Keywords:** Implosion, Reliable Multicast, Transport Protocols.

## 1 Introduction

The working principles behind reliable *sender-initiated* ([1]) one-to-one (*unicast*) protocols, such as TCP ([2]), are well-known. Extending them for one-to-many (*multicast*) communication brings scalability problems, in particular that of implosion. The *ack-implosion problem* ([3], [4]) is an acute shortage of resources caused by the volume and synchrony of acknowledgments, leading to packet losses and increase in network cost and latency.

One solution is to use the *receiver-initiated* approach. Protocols which take this approach, such as the Scalable Reliable Multicast, or SRM ([5]) and the Log-Based Reliable Multicast, or LBRM ([6]), generally

---

\* corresponding author; on leave from Universidade do Vale do Rio dos Sinos (UNISINOS), Sao Leopoldo, Brazil.

scale better because: (a) the processing burden is shifted to the receivers, and (b) feedback packets are used only as retransmission requests. By eliminating obligatory positive acknowledgements for each data packet, network cost is reduced and so is the risk of implosion. However, these protocols are not entirely free from implosion: when the same losses are experienced by many receivers, an “avalanche” of negative acknowledgements known as *NACK-implosion* ([7]) can occur. Further, the sender’s lack of knowledge about the status of receivers leads to error and flow control problems not encountered in sender-initiated protocols. For example, the sender may have to store transmitted packets longer than the application requires, and at any given time, it cannot guarantee how many packets are being successfully delivered and to how many receivers.

We develop in this paper a reliable multicast protocol that provides efficient error and flow control and has a mechanism to minimise implosion. The basic idea in minimizing implosion is to use polling ([8]): a receiver responds with a feedback packet only upon being requested or polled; the sender polls receivers at carefully planned timings so that the arrival rate of feedback packets is not large enough to cause implosion. We analysed the behaviour of our protocol using simulations which indicate that our scheme can be effective in minimising losses due to implosion.

Our protocol is developed in the context of a large number of receivers organised in a multicast tree with the sender at the root. (Similar to TMTP ([9]), RMTP ([10]), and LGC ([11]), we have chosen the tree structure because it allows faster error control and recovery through localised communication ([12]).) We assume that a receiver receives packets from, and sends its feedback to, its immediate parent in the tree. With its recursive structure for forward and reverse propagation in mind, in this paper we focus on a single unit of this recursion: an end-to-end protocol in which a non-leaf receiver or the sender (called generally the *parent*) reliably transmits to a set of  $NC$  receivers (the *child* nodes). *Reliable*, in this case, means that “at the end of transmission the parent knows that every member in its membership set has received all data packets transmitted”.

The following are assumed: *connection setup* and *termination* phases precede and succeed, respectively, a *transmission* phase, during which all transfer of data from the parent to the children occurs. Though no new member is allowed during the transmission, a child may *leave* the destination set, by own will, or be *disconnected* by the parent because of repeated absence of responses. Although aimed to work at transport level, the protocol is generic. The only requirement is an underlying point-to-point (unreliable) transmission mechanism; however, we assume a (unreliable) multipoint mechanism capable of efficiently<sup>1</sup> propagating copies of a packet to its destinations. An upper-layer at the sending host *produces* data to the sender-end of the protocol, which assembles packets and transmit them across the network to the receiver hosts; there, each receiver-end processes the packets, potentially returns responses, and orderly

---

<sup>1</sup>use of multicast routing as well as hardware-supported multicast.

delivers the data to the upper-level (packets are *consumed* by the upper-level).

The rest of this document is organised as follows. First, we begin by describing the main features of the protocol. Section 3 presents an overview of the protocol architecture. Simulation results are shown and discussed in Section 4. Section 5 concludes the paper.

## 2 Protocol Description

The protocol solves the problem of a parent node having to reliably transmit a number of data packets to *NC* children. Failures during transmission result in packets being lost or corrupted and discarded. To deal with packet losses, the parent keeps a copy of each transmitted packet in its buffer until a confirmation of receipt of that packet is obtained from all children (packet becomes *fully acknowledged*). In our protocol, a child indicates the receipt or non-receipt of packets only upon being explicitly requested to do so. The parent polls the children selectively to avoid an implosion of responses; in response to a polling request, a child positively acknowledges (or *acks* for short) the packets it has received so far and negatively acknowledges (or *nacks* for short) the packets which appear to be missing. There are four different types of packets that can be exchanged between the parent and the children: (a) DATA packets which contain only data; (b) POLL packets containing no data but only a polling request identifying a set of children to return a response; (c) DATA POLL packets, containing both data and a polling request; and, (d) RESP packets, returned by a child with feedback. Each data packet is assumed to have a sequential number (*seq*) which can uniquely identify the packet. (We assume *seq* is large enough to avoid problems that arise when sequence numbers are wrapped around and reused.) A data packet is said to be *earlier* than another data packet if the former has a smaller sequence number.

### 2.1 Sliding Windows and Poll Responses

The error and flow control of our protocol are based on a sliding window scheme. At the parent, data is assembled into fixed-size packets and transmitted by the protocol. As stated earlier, the parent keeps a copy of every transmitted data packet until it is acked by all children; for this purpose, the parent has a buffer that can accommodate  $S$  data packets. At a child, the upper-layer may be slow and consequently some packets available for consumption may remain *unconsumed*. Received packets remain *unconsumable* if an earlier packet has not been received. In order to store the unconsumed and the unconsumable packets, every child  $R_i$  is assumed to have a finite size buffer that can accommodate  $S$  data packets. The buffer size  $S$  is negotiated at connection setup, and assumed to remain constant during the transmission.

Each child keeps a receiving window  $W_i$  that is characterised by a *left edge*  $LE_i$ , a size  $S$ , which is also the size of the buffer, and the *highest received* sequence number  $HR_i$  from the parent ( $HR_i$  is set to the *seq*

of a packet received from the parent if  $seq > HR_i$ ).  $LE_i$  is the minimum between the sequence number of the earliest unconsumed packet in  $R_i$  and the sequence number of the earliest packet yet to be received by  $R_i$ . Thus  $LE_i$  refers to the smallest sequence number of the packet that is either waiting to be consumed or expected to be received.  $W_i$  is a boolean vector indexed by  $seq$ ,  $LE_i \leq seq < LE_i + S : W_i[seq]$  is true if  $R_i$  has received the data packet  $seq$ , or false otherwise.

The parent keeps a set of  $NC$  sending windows, one  $W_{p,i}$  for each child  $R_i$ .  $W_{p,i}$  is the parent's (latest) knowledge of  $W_i$  of  $R_i$ . Like  $W_i$ , it is characterised by a left edge, denoted as  $LE_{p,i}$ , and size  $S$ .  $LE_{p,i}$  and  $HR_{p,i}$  are the parent's knowledge of  $LE_i$  and  $HR_i$ , respectively. For  $seq$ ,  $LE_{p,i} \leq seq < LE_{p,i} + S$ ,  $W_{p,i}[seq]$  indicates the parent's knowledge of whether  $R_i$  has received the data packet  $seq$ ; it is initially set to false. Finally, the parent keeps the variable  $HS$  to record the largest  $seq$  of data packets multicast so far, and thus applies to all  $W_{p,i}$ .

When the parent sends a polling request, it includes the following information: (a) a timestamp denoted as  $POLL.TS$ , which is the time when the  $POLL/DATAPOLL$  was sent; (b) the set of children denoted as  $POLL.CLD$  that are being requested by this poll to respond; and, (c) a sequence number  $POLL.SEQ$  which indicates the  $seq$  of the data packet that was sent just before or with this poll (that is,  $POLL.SEQ$  will be  $HS$ ). When child  $R_i$  receives a poll, it checks whether its id is part of  $POLL.CLD$ . If so, the child responds by sending a packet  $RESP$  to the parent containing (a)  $RESP.W$ , which is the copy of its receiving window ( $RESP.W.LE$  contains the value of  $LE_i$ ); (b)  $RESP.W.HR$ , the value of  $HR_i$ ; (c) a timestamp  $RESP.TS$ , and (d) a sequence number  $RESP.SEQ$ . The values of  $RESP.TS$  and  $RESP.SEQ$  are the same as the  $POLL.TS$  and  $POLL.SEQ$ , respectively. The  $RESP.TS$  enables the parent to distinguish earlier responses from later ones, since it is possible to send more than one polling request without advancing  $HS$  (i.e., with no new data being transmitted between successive poll requests). It is also used by the parent to estimate the round trip time (RTT) between itself and the child:  $RTT$  measured is the arrival time of  $RESP$  at the parent minus the  $RESP.TS$ .

When the parent receives a  $RESP$  packet from  $R_i$ , it updates its variables related to  $R_i$ :  $LE_{p,i} \leftarrow \max\{LE_{p,i}, RESP.W.LE\}$ ,  $HR_{p,i} \leftarrow \max\{HR_{p,i}, RESP.W.HR\}$  and only then, for all  $seq$ ,  $RESP.W.LE \leq seq \leq RESP.W.HR$ ,  $W_{p,i}[seq] \leftarrow W_{p,i}[seq] \vee RESP.W[seq]$ . From  $W_{p,i}$ , the parent can infer that  $R_i$  has received all data packets with  $seq$ ,  $seq < LE_{p,i}$  or  $W_{p,i}[seq] = true$ , and not received packets with  $seq$ ,  $seq \leq HR_{p,i}$  and  $W_{p,i}[seq] = false$ ; all packets with  $seq$ ,  $HR_{p,i} < seq \leq HS$  are in transit and have not reached  $R_i$ . Based on these inferences, the parent detects and handles packet losses.

A snapshot of windows at the children and the parent nodes at a given time is shown in Figure 1;  $S$  is assumed to be 10.  $LE_1$  is the  $seq$  of the earliest missing packet in  $R_1$  (#100), which prevents packets #101 – 102 from being made available for consumption.  $HR_1 = 104$  means that  $R_1$  does not know whether the packets from #105 onwards are yet to be transmitted by the parent or have already been

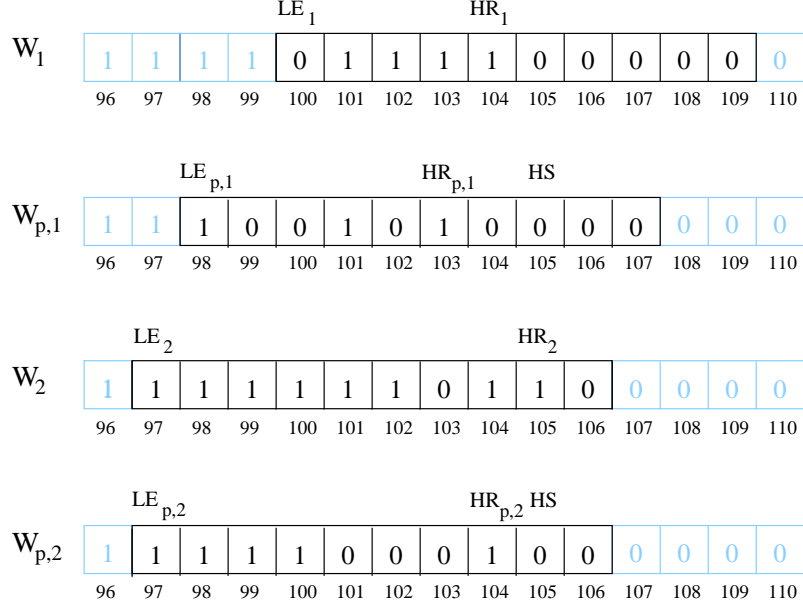


Figure 1: Example of the sliding windows.

transmitted. Let  $RESP_1$  and  $RESP_2$  be the last responses which the parent has received from  $R_1$  and  $R_2$ , respectively. Comparing  $W_{p,1}$  with  $W_1$  indicates that ever since  $R_1$  has sent  $RESP_1$  it has received packets #99, #102, and #104, and its  $W_1$  has slid (to the right) by two packets due to the consumption of packets #98 and #99. Similarly,  $W_{p,2}$  and  $W_2$  indicate that after sending  $RESP_2$ ,  $R_2$  has received packets #101, #102, and #105, and the window has not slid. The consumption at  $R_2$  seems rather slow because the packets #97 – 102 (see  $W_2$ ) are consumable but remain unconsumed. Finally,  $HS$ , which cannot be smaller than  $\max\{HR_1, HR_2\}$ , is taken to be 105 in the figure.

## 2.2 Flow Control

Our protocol employs a window-based flow control mechanism: a sliding window is used to determine how many new packets can be safely multicast without causing buffer overflow at the children. Since packets are transmitted to all children, the child with the smallest number of free buffer spaces will determine the number of new transmissions.

The parent determines the *effective window* ( $EW_{p,i}$ ) for each child  $R_i$ , where  $EW_{p,i}$  denotes the number of new packets child  $R_i$  can take without buffer overflow:  $EW_{p,i} \leftarrow (LE_{p,i} + S) - (HS + 1)$ . If  $EW_{p,i} > 0$ ,  $R_i$  has space in its buffer to receive at least  $EW_{p,i}$  new packets (child  $R_i$  can receive more than  $EW_{p,i}$  packets without buffer overflow if some packets were consumed while its poll response was being transmitted to the parent.) Since the parent has to wait for the slowest children,  $EW_p$ ,  $EW_p \leftarrow \min\{EW_{p,i} \mid \forall i : 1 \leq i \leq NC\}$ , new packets are transmitted. We define a *sending window*  $W_p$  for the parent with  $LE_p \leftarrow \min\{LE_{p,i} \mid 1 \leq i \leq NC\}$  and size  $S$ . When  $LE_p + S = (HS + 1)$ ,  $EW_p$  is

zero and the  $W_p$  is said to be *closed*, blocking transmissions of new data from the parent.

Referring to Figure 1, we provide an example to illustrate how the scheme with multiple windows regulates new transmissions by the parent and avoids buffer overflow at the children. As  $LE_{p,1} = 98$  and  $HS = 105$ ,  $EW_{p,1} = (98 + 10) - (105 + 1) = 2$ , and as  $LE_{p,2} = 97$ ,  $EW_{p,2} = (97 + 10) - (105 + 1) = 1$ . After  $R_i$  has transmitted the last response ( $RESP_1$ ),  $W_1$  has slid by two packets; so,  $R_1$  can receive 4 new packets #106 – 109 (more than  $EW_{p,1}$ ).  $R_2$  depicts the worst case when  $LE_{p,2} = LE_2$  and  $W_2$  has not slid since the last response was sent. The flow control scheme caters for this worst case:  $EW_p = \min\{1, 2\} = 1$  and the parent can transmit only a single packet, #106.

In addition to the window-based scheme, the protocol allows the user to set a maximum transmission rate by establishing an inter-packet gap ( $IPG$ ), which we define as the minimum interval to be observed between the transmission of any two packets by the parent.

### 2.3 Polling Mechanism

A sender-initiated unicast reliable protocol, such as TCP ([2]), typically triggers one acknowledgement per one or two DATA packets transmitted. A reliable multicast protocol, such as [13] and [1], results roughly in one acknowledgement per DATA packet per receiver. Expressing these in polling terms, the parent can be regarded to include 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 implosion. To avoid implosion, our protocol is designed to request only a selected set of children 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 child not being polled during the transmission of a (finite) number of data packets; so, naturally, a response from a child will ack not just a single packet but will ack/nack all packets received/missed between successive poll requests.

Polling requests cause response packets to be sent to the parent. 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. We thus define the maximum “allowable” arrival rate of incoming responses as the *implosion threshold rate*, or  $ITR$  for short. Though  $ITR$  cannot be known precisely, we assume that it can be estimated with reasonable accuracy (see [6]).

To avoid losses by implosion, the protocol controls the arrival rates and timings of response packets returned by children. The mechanism aims at implementing a given *response rate* ( $RR$ ), which is an input value of the protocol. The lower is the  $RR$ , the fewer will be the 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 acks from all children; hence the parent may be blocked (because of closed  $W_p$ ) longer from making new transmissions. Thus, a smaller  $RR$  may also result in smaller throughput. To seek a balance

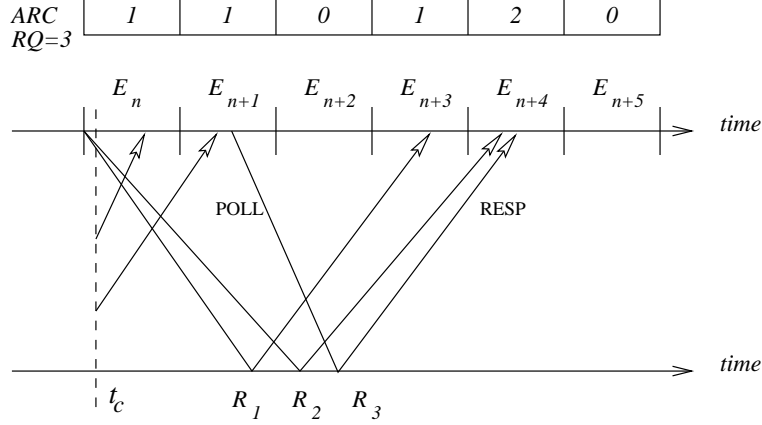


Figure 2: Polling scheme based on allocation of responses in time.

between throughput and implosion losses, it is preferable to have  $RR$  equal to  $ITR$  (in section 4, we show that the throughput is not significantly affected when  $RR$  is set to slightly underestimated values of  $ITR$ ). In order to keep the arrival rate of responses equal to  $RR$ , the mechanism controls the arrival times of responses by *planning ahead* the time when every given child should be requested to respond. This mechanism is explained in more detail below.

Time is divided into *epochs*, time intervals of fixed-length  $\varepsilon$ , where  $\varepsilon = k \times IPG$  for some  $k \geq 1$ . Epochs are denoted as  $E_n$ , with  $n = 0, 1, \dots$ . A *response quota*, denoted as  $RQ$  and calculated as  $RQ \leftarrow \lfloor RR \times \varepsilon \rfloor$ , is initially allocated to each epoch. The polling mechanism estimates the arrival time of responses using its estimate of RTT, and schedules the transmission of polling requests such that at most  $RQ$  responses are expected to be received during any epoch. A vector, called *Anticipated Response Count*, or *ARC* for short, is maintained to keep track of the number of responses which have been planned to arrive in an epoch.  $ARC$  is indexed by the epoch number;  $ARC[n]$  is initialised to 0 and incremented by 1 if the response for a planned poll is expected to arrive during  $E_n$ . In Figure 2,  $t_c$  is the current time and  $RQ$  is 3.  $ARC[n+2] = ARC[n+5] = 0$ , since no poll response is expected during  $E_{n+2}$  and  $E_{n+5}$ ;  $ARC[n+4] = 2$  as two responses are expected to arrive in  $E_{n+4}$ .

The time to send a poll request to  $R_i$  is planned as follows: (i) assuming that a polling request can be sent immediately<sup>2</sup>, find the earliest epoch  $E_n$  such that  $E_n$  contains or follows the time  $clock + RTT_i$  (where  $clock$  represents the current clock value and  $RTT_i$  the RTT estimate for  $R_i$ ) and  $ARC[n] < RQ$ ; (ii) increment  $ARC[n]$  by 1; (iii) assign the *estimated sending time* ( $est_i$ ) of the polling request to  $R_i$  to be  $est_i = \max \{clock, n \times \varepsilon - RTT_i\}$ . There is at most one polling planned ahead for each child, and this information is kept in a table called the *planned polling table*, or *PPT*. The planning of a poll for a given child  $R_i$  happens in any one of three situations: (a) just before a DATA packet is to be transmitted to

<sup>2</sup>How this assumption is met is described later.

$R_i^3$  and  $R_i$  is not in  $PPT$ ; (b)  $R_i$  reports a buffer full of unconsumed packets ( $W_{p,i}[seq] = true$  for all  $seq : LE_{p,i} \leq seq < LE_{p,i} + S$ ); and (c) when  $R_i$  appears not to have returned its response for a polling request (more details in section 2.4).

The transmission of a polling request is carried out according to the timing information in  $PPT$ . There are two types of situations which could lead to the examination of  $PPT$  for scheduled poll timings. In the first case,  $PPT$  is examined just before a DATA packet is to be sent. The set of children with  $est_i \leq clock$  is removed from  $PPT$  and piggybacked onto the ongoing DATA packet. The second case refers to the situations where there is no DATA packet to be sent or an available packet cannot be sent due to closed  $W_p$ . In that case, the next polling time ( $NPT$ ) is estimated to be one  $IPG$  plus the minimum of the  $est_i$  in  $PPT$ . At  $NPT$ , a POLL packet is sent requesting the set of children with  $est_i \leq NPT$ . When no data packet can be sent, successive  $NPTs$  will be at least one  $IPG$  apart. This ensures that only one POLL packet is used to request all children whose  $est$  are between  $NPT - IPG$  and  $NPT$ . If it ever becomes possible to send a DATA packet between  $NPT - IPG$  and  $NPT$ , then the scheduled poll at  $NPT$  is cancelled, and a new  $NPT$  is computed if  $PPT$  is not empty and there is no data to be sent. Observe that in both cases the polling mechanism allows up to one  $IPG$  to elapse between the scheduled and actual transmission times of a poll 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: (a) it allows a polling request planned for  $est_i$  to be sent anytime between  $est_i$  and  $est_i + IPG$ ; (b) the processing loads at the host CPU may cause the time between successive transmission of polling requests exceed  $IPG$ ; this may further increase the difference between  $est_i$  and actual transmission times; (c) 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 (a), (b) and (c), as well as by the values adopted as  $\varepsilon$ . Using a higher value for  $\varepsilon$  (and thus higher  $RQ$ ) results in fewer but larger epochs, decreasing the probability of a response arriving outside the expected epoch. However, with greater  $RQ$ , then some responses may be lost if all expected responses in a given epoch arrive *en masse* at the same point in that epoch.

## 2.4 Handling Absent Poll Responses

Polling requests and responses can be lost during transmission. So, to avoid waiting for ever to receive a response from a given polled child  $R_i$ , the parent waits on a retransmission timeout ( $RTO_i$ ). The parent

---

<sup>3</sup>If  $est_i$  turns out to be the current time, then a polling request to  $R_i$  is included in the DATA packet which then becomes a DATAPOLL packet.



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 transmitting additional packets.

When no response is received from a polled child within the  $RTO_i$ , the parent declares the response to be *absent*. The absence of a poll response from a given child can be due to one of the following: (a) a transient failure caused the polling request to be lost or discarded; (b) the response transmitted by the child did not reach the parent due to a transient failure; (c) the  $RTO_i$  is too small and the response is still in transit; (d) the child has become permanently disconnected or failed. It is impossible for the parent to know what exactly is the underlying cause for an absent response. The protocol deals with a suspected absent response by *repolling* the child and waiting (on timeout) for a response; this repolling is repeated for a finite number of times. If the underlying cause is (a), (b) or (c), then it is hoped that the parent 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 acked/nacked by that poll response. If a child  $R_i$  fails to respond within  $RTO_i$ , then  $R_i$  requires a repoll as soon as possible. This is because the earlier  $R_i$  is made to respond, the sooner a transmitted packet will get fully acked and be removed from the buffer. 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, the RTT to the child is used to estimate the epoch  $E_m$  in which the response would be received if the polling request to  $R_i$  were to be sent now (i.e.,  $est_i \leftarrow clock$ ). (c1) If  $ARC[m] < RQ$  then (a1)  $ARC[m]$  is increased by 1. Else (if  $ARC[m] = RQ$ ), then check (c2) if there is another child  $R_j$  in  $PPT$  which is not in REPOLL and from which a response is expected during  $E_m$ . (Recall that  $PPT$  contains information about poll requests that have not yet been transmitted.) If (c2) is true, then (a2)  $R_i$  “steals” the response quota from  $R_j$  (i.e., a poll request is sent to  $R_i$  at the earliest possible time), and  $R_j$  is re-scheduled following the normal scheduling procedure described in section 2.3. If both conditions (c1) and (c2) are false for  $m$ , then repeat checking (c1) and (c2), in this order, for  $m + 1, m + 2, \dots$  until either condition is satisfied for some  $m'$ ,  $m' > m$ . Set  $est_i \leftarrow m' \times \varepsilon - RTT_i$  and then apply action (a1) or (a2), if the condition to become true was (c1) or (c2), respectively.

Our protocol assumes that if no response is received for a given number of consecutive polls<sup>4</sup>, then the underlying cause is (d) and the non-responsive node is removed from the child set. Removing a persistently non-responsive receiver node from the set of children relieves the parent from having to wait for acks from that node; this may open  $W_p$  and allow the parent to transmit new packets. Once a child is removed, any packet received from it is ignored.

---

<sup>4</sup>This number is a user configurable, protocol variable.

## 2.5 Data Loss Recovery

The parent node detects the loss of the packets it transmitted through the poll responses sent by the children. The scheme employed by the parent to recover from these losses involves retransmission of packets which can be done via multicast or selective unicasts. The way the scheme operates can influence the system throughput, network load, and the number of packets processed by a child. In this subsection, we explain the rationale behind the design choices we have made for our protocol.

We will first describe 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 children, since the same packet will be unicast multiple times. For example, if a data packet is propagated from the parent to the children through a multicast routing tree, the loss of a data packet near the parent would probably lead to the loss of  $seq$  in a large percentage of the children at lower levels of the tree. So, if a packet is lost by more than one child, it may be advantageous to retransmit it via multicast.

The second scheme pessimistically assumes that if a packet is nacked by one child then it will be nacked by many other children as well. Based on this assumption, the packet  $seq$  is retransmitted via multicast soon after a nack for  $seq$  is received. When the number of children that share the same loss is likely to be large, this scheme speeds up recovery for those children whose nacks have not yet reached the parent. On the other hand, resending a packet to a child that already has the packet, incurs unnecessary network load, and processing cost for that child. In the extreme case, a single lossy child 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 [6] as the “crying baby” problem.

The third scheme uses a wait-and-see approach to minimise 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 children. Different criteria can be used to limit the waiting (e.g., a fixed interval) during which nacks are collected. In general, the longer is the collection time, the more appropriate will be the decision made for recovery, and hence the fewer will be the unnecessary packets transmitted.

Our protocol employs both the first and the third schemes, in the following manner. The third scheme is put in operation as soon as the parent receives the first nack for a given packet  $seq$ . The mechanism waits collecting nacks; if the percentage of children (out of  $NC$ ) that has nacked  $seq$  reaches or exceeds the *multicast threshold ratio* ( $MTR$ ), the packet is multicast. The multicast, if carried out, will terminate the waiting. If the multicast cannot be done, the waiting will terminate after each child either has acked/nacked  $seq$  or is in `RE POLL` for having failed to respond to a poll with `POLL.SEQ ≥ seq` (recovery

is not delayed waiting for non-responsive children); this will be then followed by unicasting  $seq$  to each child that has nacked  $seq$ . After a multicast or a series of unicasts is carried out, the workings of the third scheme terminate; the first scheme then becomes operative and will be in force until  $seq$  becomes fully acked. The first scheme is more appropriate after the third scheme because the number of children which still require retransmissions of  $seq$  is likely to be small. Below we describe the mechanism in detail.

### 2.5.1 The recovery algorithm

Suppose that the parent multicasts (for the first time) a packet with number  $seq$  to all children. Within the parent node, the variable  $status(seq)$  records the status of a transmitted packet, and is initially set to `NO_NACK`; the set variable  $ALL$  contains all receiver nodes which the parent regards as children. The parent computes the set  $acked(seq)$  as the set of all children from which a response has been received acking  $seq$ , or more formally:

$$acked(seq) \leftarrow \{R_i \in ALL \mid seq < LE_{p,i} \vee W_{p,i}[seq]\}$$

Consider a child  $R_i$  in the set  $ALL - acked(seq)$ . Regarding the responsive behaviour of child  $R_i$  towards polls sent with `POLL.SEQ`  $\geq seq$ , the child  $R_i$  can be regarded by the parent to be in one of the following situations: (a) it has nacked  $seq$  in response to some poll; (b) its response has been judged to be absent for all the polls sent; or, (c) a response has not yet been received from  $R_i$  regarding  $seq$ . In the last case, the parent waits to decide between (a) and (b). If  $R_i$  is in situation (a), it is grouped in the set that represents the set of all children that have nacked the packet  $seq$ ,  $nacked(seq)$ . More formally,

$$nacked(seq) \leftarrow \{R_i \in ALL \mid \neg W_{p,i}[seq] \wedge seq \leq HR_{p,i}\}$$

If  $R_i$  is in case (b), the parent will either repoll or decide to remove child  $R_i$  from its set  $ALL$ . The set of all children that have been repolled at least once and appear not to have responded to any of the polls sent with `POLL.SEQ`  $\geq seq$  are categorised as a set  $repolled(seq)$ . To formally define  $repolled(seq)$ , we will use the predicate  $sent(POLL, i)$  that becomes true only if the parent has sent a polling request `POLL` to  $R_i$ , and define a set  $absent(POLL)$  that contains the children whose responses for `POLL` are judged to be absent:

$$repolled(seq) \leftarrow \{R_i \in ALL \mid \forall POLL: sent(POLL, i) \wedge POLL.SEQ \geq seq : i \in absent(POLL)\}$$

Once  $nacked(seq)$  becomes non-empty,  $status(seq)$  changes from `NO_NACK` to `COLLECTION` and the recovery mechanism starts collecting nacks until the following condition becomes true:

$$|nacked(seq)| \geq MTR \times NC \vee acked(seq) \cup nacked(seq) \cup repolled(seq) = ALL$$

Once this condition is true (which may occur immediately), the parent retransmits. If  $|nacked(seq)| \geq MTR \times NC$ , then the packet  $seq$  is multicast to all children and the retransmission time is recorded in a table called the *retransmission table* ( $RTxT$ ). This table is indexed by two parameters:  $[cid, seq]$  where  $cid$  is the id of the child to which the packet was retransmitted and  $seq$  is the sequence number

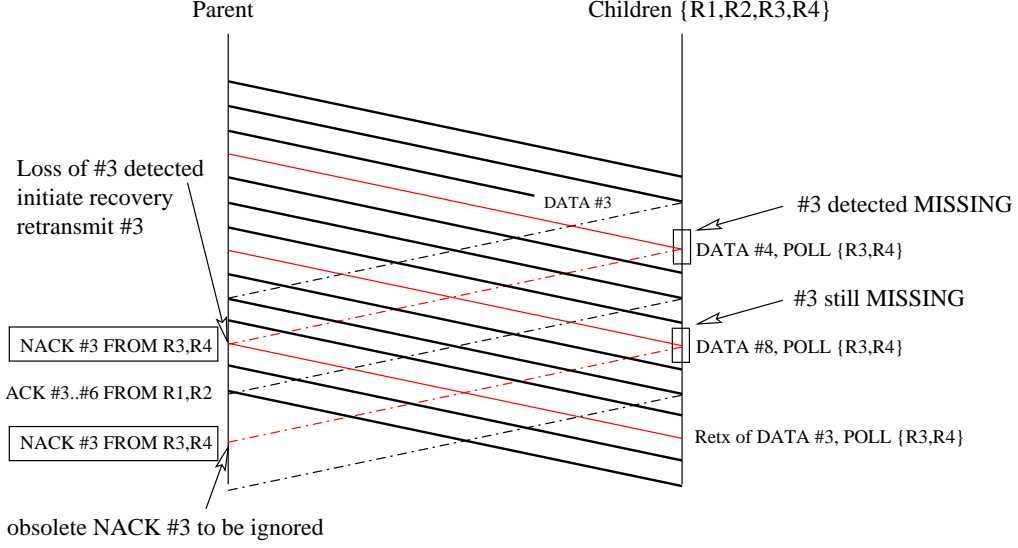


Figure 3: Example of an obsolete nack.

of the retransmitted packet. The table entry  $RTxT[cid, seq]$ , if exists, indicates the latest time when the packet  $seq$  was retransmitted to the child indicated by  $cid$ . (If  $RTxT$  has no entry for  $[cid, seq]$ , then that would mean that the packet has not yet been retransmitted to  $cid$ .) After making the multicast retransmission, the parent enters in  $RTxT$  the time of retransmission for every child in  $nacked(seq) \cup repolled(seq)$ . If the above condition became true with  $|nacked(seq)| < MTR \times NC$ , then, for each child in  $nacked(seq)$ , the packet  $seq$  is unicast and the time of transmission is entered in the  $RTxT$ . After the packet  $seq$  is retransmitted, the parent regards the recovery status of the packet to be RETRANSMITTED, i.e.,  $status(seq) \leftarrow \text{RETRANSMITTED}$ .

Suppose that the parent has received a nack for packet  $seq$  from  $R_i$ . After it has retransmitted the packet to  $R_i$ , it should regard all the nacks that precede the retransmission as *obsolete*, avoiding redundant retransmissions. To distinguish meaningful nacks from obsolete ones, the timestamp  $RESP.Ts$  is compared with the corresponding timestamp recorded in  $RTxT[cid, seq]$ , if any; if there exists the entry  $RTxT[cid, seq]$  and  $RESP.Ts < RTxT[cid, seq]$ , then the nack is deemed obsolete and thus ignored. Figure 3 depicts an example of an obsolete nack. A parent with four children  $\{R_1, R_2, R_3, R_4\}$  is assumed and the time moves from top to bottom; the flows of DATA, DATAPOLL, and RESP packets are indicated by solid, light, and broken lines, respectively. The DATA#3 is not received by children  $R_3$  and  $R_4$ , and they nack DATA#3 in their responses to the POLL sent along with DATA#4. When these responses arrive, DATA#3 is re-multicast (assume  $MTR \leq 50\%$ ) just after DATA#11 is multicast. After this retransmission, the parent receives nacks for packet DATA#3 from  $R_3$  and  $R_4$  which were sent in response to the POLL sent with DATA#8.  $RESP.Ts$  of these nacks will be smaller than the retransmission time recorded for  $[R_3, seq]$  and  $[R_4, seq]$ . So, the parent will identify the second nacks for DATA#3 as obsolete and discard them.

Recall that the parent removes from the set  $ALL$  any child that appears not to have responded to a

given, user-specified number of consecutive polls. Hence  $repoll(seq)$  will become empty eventually. We assume that a functioning child which nacks  $seq$  one or more times will receive the packet  $seq$  after a finite number of recovery attempts by the parent. We present below the algorithm for recovering  $seq$  which will be executed whenever the parent receives or detects the absence of RESP,  $RESP.SEQ \geq seq$ , and if  $acked(seq) \subset ALL$ :

```

case  $status(seq)$  {
  NO_NACK:
    IF ( $nacked(seq) \neq \{\}$ ) {STATUS( $seq$ )  $\leftarrow$  COLLECTION;}
  COLLECTION:
    if ( $|nacked(seq)| \geq MTR \times NC$ ) {
      MULTICAST PACKET  $seq$ ; STATUS( $seq$ )  $\leftarrow$  RETRANSMITTED;
      for EACH  $R_i \in nacked(seq) \cup repolled(seq)$  {  $RTxT[i, seq] \leftarrow retxTime$  }
    } else if ( $acked(seq) \cup nacked(seq) \cup repolled(seq) = ALL$ ) {
      for EACH  $R_i \in nacked(seq)$  { UNICAST PACKET TO  $R_i$ ;  $RTxT[i, seq] \leftarrow retxTime$ }
      STATUS( $seq$ )  $\leftarrow$  RETRANSMITTED;
    } break;
  RETRANSMITTED:
    if (RECEIVED RESP FROM  $R_i$  WITH A NON-OBSOLETE NACK) {
      UNICAST PACKET TO  $R_i$ ;  $RTxT[i, seq] \leftarrow retxTime$ ;
    }
}

```

### 3 Protocol Architecture

In this section we describe an architecture for implementing our protocol. Our description refers to three kinds of components: *modules*, *queues* and *tables*. Modules are the only active entities (*threads*), and are scheduled non-preemptively (one thread can schedule one or more threads to be run at the moment or later).

Two queues are employed, namely, a *transmission queue* (TXQ) and a *timeout queue* (TOQ). To avoid unnecessary memory copies, only a reference to the data packet (sequence  $seq$ ), not its contents, circulates through queues and tables in the system. TXQ contains packets to be transmitted/retransmitted; packets in the ascending order of their  $seq$ , so that the policy of giving priority to retransmissions over ordinary transmissions is implemented. TOQ contains an entry TO for every scheduled asynchronous event. Most entries are retransmission timeouts (*RTO*) used to limit waiting for responses from polled children. For

every POLL sent, an entry TO is made with TO.TS, TO.SEQ, and TO.CLD set to POLL.TS, POLL.SEQ, and POLL.CLD respectively. The timeout expiry time TO.EXP is set to  $\max\{RTO_i \mid \forall i : i \in \text{TO.CLD}\}$ . Also, whenever  $NPT$  is determined an entry is made with  $\text{TO.EXP} \leftarrow NPT$ . Entries are ordered as per TO.EXP. The system alarm is set for TO.EXP of the first entry in TOQ.

There are five main tables used at the parent node: *planned polling table* (PPT), *response table* (RT), *retransmission table* (RTXT), *recovery status table* (RST), and *missing poll table* (MPT). PPT records for each child  $R_i$  the  $est_i$  (which will be 0 if no poll is planned), the number of the epoch during which the response is expected to arrive, and the priority which will be high if  $R_i$  is in repoll. RT consists of  $NC$  windows  $W_{p,i}$ . RTXT (realisation of  $RTxT$  in section 2.5) records for each  $R_i$  the latest transmission time of every packet that has been unicast to  $R_i$ . It also contains a special entry to record the latest time of every multicast retransmission.

Recall that a nack from  $R_i$  for packet  $seq$  is to be processed depending on whether: (a) it is/is not the first nack to be received for  $seq$ ; (b) it is a meaningful/obsolete nack if the packet has been retransmitted to  $R_i$ . RST is implemented to efficiently provide the variable  $status(seq)$  described earlier. Only if the status is RETRANSMITTED, RTXT is searched to see whether the received nack is meaningful or not.

The MPT records for each  $R_i$  the number of consecutive polling requests for which the responses from  $R_i$  are found to be absent; if this count is not 0 it also records POLL.TS of the last polling request for which the response was absent. Whenever a RESP from  $R_i$  is received and if the count in MPT is not 0, then it is modified depending on RESP.TS and the timestamp recorded in MPT for  $R_i$ : the count is decremented by 1 if these timestamps are equal; it is set to 0 or 1 if RESP.TS is larger or smaller, respectively.

Besides these five tables, there is the ARC object used for planning poll timings. Since at most one poll is planned for a child, the ARC size is  $NC + \lceil \text{RTT\_MAX} / \varepsilon \rceil$ , where RTT\_MAX is the maximum expected  $RTT$  to the children. Since RTT\_MAX cannot be safely predicted, the ARC size can dynamically increase if the measured  $RTT$ s increase; in such case, the number of entries in ARC will be incremented according to a step function.

We will now describe the modules (threads) at the parent which are: *generator module* (GM), *transmitter module* (TXM), *response handler module* (RHM), and the *event manager* (EM); at the receiver's side, there is a single module, called *receiver* (RXM). Figure 4 illustrates the interaction between these modules. The GM's role is to take continuous data from the upper-level, divide it into fixed-size data packets, and queue them for transmission at TXQ. It queues a new packet only if  $EW_p > 0$ . If  $EW_p = 0$  and if there is one (or more)  $W_{p,i}$  full of unconsumed packets, then GM will arrange a polling request to all such  $R_i$ . It enqueues in TXQ an empty DATA packet (with  $seq = HS$ ) to be sent to all  $R_i$ . TXM will transform this packet into a POLL packet (see below).

TXM consumes the packets from TXQ and transmits them. While TXQ is non-empty, it executes the

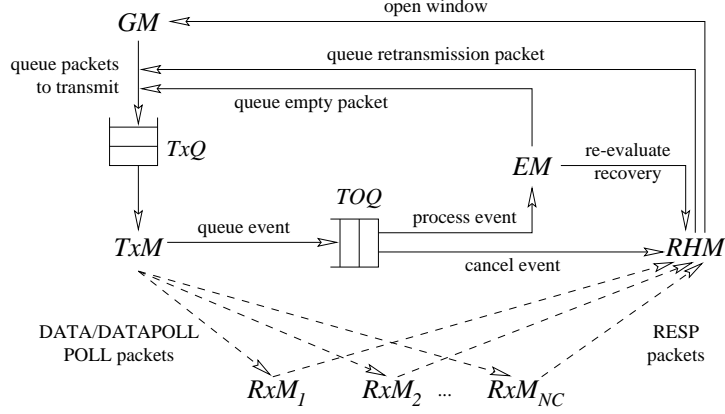


Figure 4: Overall structure of the protocol machine.

following sequence of actions: (i) take the first packet from TXQ; (ii) make sure at least one *IPG* has elapsed since the last transmission; (iii) plan polling for each child which is to receive this packet and does not have a planned polling in PPT; (iv) generate a polling set POLLSET according to PPT contents and piggyback POLLSET into the packet if POLLSET is not empty; if both POLLSET and DATA are empty, then the packet is discarded; (v) assign the appropriate type to the packet; (vi) transmit the packet either through multicast or multiple unicasts, by comparing the size of the destination set with  $MTR \times NC$ ; (vii) if the packet is of type POLL/DATAPOLL, add an entry to the TOQ with appropriate expiry time. If the TXQ becomes empty and PPT is not empty, then TXM computes *NPT* and makes an “*NPT*-entry” in TOQ.

The RHM handles all the feedback from children and coordinates loss detection and recovery. The cycle executed by the response handler is as follows: (i) wait for an incoming RESP packet; (ii) read a RESP packet available from the lower layer; (iii) scan the TOQ and remove the child  $R_i$  that sent RESP from TO.CLD of all TO with  $TO.TS \leq RESP.TS$ ; remove any TO with empty TO.CLD from TOQ; (iv) update  $W_{p,i}$  of RT, and MPT if required; (v) for each  $seq$ ,  $LE_p \leq seq \leq RESP.W.HR$ , carry out the recovery action for  $seq$  (as described in section 2.5.1): update RST if necessary, and identify whether a retransmission (via multicast or unicast) is necessary. If a retransmission is necessary, queue the packet in TXQ.

The EM sets the alarm for TO.EXP of the head of the TOQ. When the alarm expires, EM inspects the event in the entry at the head of TOQ. If this is a *RTO* event for a polling request POLL sent earlier, then MPT is updated for each child in TO.CLD; for every  $seq$ ,  $LE_p \leq seq \leq POLL.SEQ$ , carry out the recovery actions as described in (v) above (by invoking a method exported by RHM). If TO is about an *NPT*-event, TXQ is examined. If TXQ is non-empty, nothing is done since the next transmission of data will carry the necessary poll request; if TXQ is empty, EM adds an empty DATA packet to be sent to all  $R_i$  with  $est_i \leq clock$ .

The module  $RMX_i$  executes the following loop at  $R_i$ : (i) waits for packets from the lower layer; (ii)

Connection Type	Latency ( $L$ )	Latency Std. Dev ( $SD$ )	Error $Err$ (%)
LAN	1.5 ms	0.08	1
INTERLAN	5 ms	0.5	1
WAN	75 ms	15	10

Table 1: General properties assumed for kinds of connections.

takes available packet; (iii) updates  $W_i$ ; (iv) checks the bit in the POLLSET of the received packet to see if a response is requested, and if so, return a RESP packet to the parent.

## 4 Simulation Results

To study the behaviour of our protocol we carry out simulation experiments under various settings. Further, to perform a comparative analysis we also simulate the sender-initiated reliable multicast protocol described in [1]. The main characteristics of this protocol are as follows: (a) it employs a sliding window scheme with selective retransmission (i.e., no go-back-N); (b) receivers acknowledge every packet received; (c) loss detection is timeout-based, and recovery via global retransmissions. We chose this protocol because it was used by [1] for comparing sender-initiated and receiver-initiated schemes. We call this protocol *full feedback* or FF for short, and ours, *polling feedback* or PF for short. We have conducted a series of experiments for both these protocols by varying the group size and considering two different network configurations.

To present the configurations considered, we first characterise three basic types of connections between a child and the parent, namely: LAN, INTERLAN, and WAN. Each kind of connection is characterised by a set of three attributes: propagation latency mean  $L$ , latency standard deviation (to emulate jittering)  $SD$ , and the percentage error rate  $Err$ . The values we associate with each type are listed in table 1. For each type we consider a network configuration in which all children are connected to the parent by connections of that type. In addition, we consider a HYBRID configuration that contains all types of connections; at least  $\lfloor NC/3 \rfloor$  children are connected to the parent by connections of a given type ( $NC \geq 3$  in this configuration). The simulation results obtained were almost identical for LAN and INTERLAN configurations, and similar for WAN and HYBRID configurations. For space reasons, we will only present the results for LAN and HYBRID configurations.

Because of the well-known impact of the window size (to “keep the pipe full”) on throughput, we tested both PF and FF for two large window sizes ( $S$ ): (a) 64 packets, and (b) infinity, i.e.,  $S$  equals to the number of packets to be transmitted. When the window size is set to infinity, the polling feedback and full feedback protocols are respectively denoted as PF-IW and FF-IW. Further, to assess the impact of implosion losses, we run the FF-IW protocol for an infinite implosion threshold rate ( $ITR = \infty$ ), and this protocol version is called FF-IW-IT. Note that none of the FF versions has implosion control mechanism;



however, when the window size is small, it could indirectly limit the number of losses due to implosion, for the following reason: when the sending window at the parent closes, preventing new transmissions, the transmission rate decreases and also does the response rate. With a smaller response rate, fewer packets will be lost due to implosion.

For the two network configurations and the five protocol versions, we have measured the following values for different group sizes: the throughput  $T$ , the relative network cost  $N$ , and the relative number of implosion losses  $I$ . Let the amount of data to be transferred be  $D$ , the packet size be  $P$  (both measured in bytes), and  $DP$  be the number of packets to be transmitted:  $DP = \lceil D/P \rceil$ . Let  $\Delta t$  be the period of time (in ms) between the transmission of the first DATA packet and the moment all packets become fully acked (both events occurring at the parent), the throughput is calculated as  $T = DP/\Delta t$ , in packets/ms. The theoretical optimum value for  $T$  is close to the user-specified maximum transmission rate ( $1/IPG$  packets/ms), and achieving that can be restricted by network bandwidth and window size.  $N$  is calculated as the total number of packets exchanged  $TP$  per child per application packet, i.e.,  $N = TP/(NC \times DP)$ ; the ideal value for  $N$  is  $DP/(DP + 1)$  (one ack per child is required at the end of transmission).  $I$  is measured as the ratio of total implosion losses to  $NC \times DP$ . The desired value for  $I$  is 0, i.e., no losses due to implosion.

For a given set of parameters, we run between ten and twenty simulation runs, and the graphs shown here have a percentage offset of under  $\pm 4.9\%$  with a confidence level of 95% ([14]). The multi-thread support of the Simula language ([15]) was used to implement the threads mentioned in the section 3.

We simulated the implosion losses in the following manner: we define a buffer at the parent for storing the incoming responses. An incoming packet is stored in the buffer if there is space, otherwise it is discarded. At every  $1/ITR$  a packet is consumed from the buffer so long as the buffer is not empty. We assume a buffer size of 16 packets and  $ITR = 1500$  responses/sec (based on measurements in [6]). We also assume: (a) ready supply of data by the upper level at the parent; (b) *hungry* upper-level at children, with immediate consumption of consumable data; (c) a top transmission rate of 1,000 packets/sec ( $IPG = 1\text{ms}$ ); (d)  $RR = ITR$ ; (e) epoch length  $\varepsilon = 10\text{ms}$ ; (f) transmission of 1 MB of data in 1,000 packets ( $DP = 1000$ ) of 1 KB each; (g) RTTs are modeled using latencies randomly generated according to the *Normal distribution*; (h) packet losses due to causes other than implosion are modelled by a statistical draw using *Err*; therefore, losses are assumed to be mutually independent; (i) the probability of loss *Err* is applied equally to all types of packets; (j)  $MTR = 20\%$  (applicable only for PF and PF-IW). A lower *MTR* value would increase  $N$ , and also  $T$ ; we chose a small *MTR* with network cost in mind.

In Figure 5 we show the throughput for each of the five protocol runs in the LAN configuration. First we note that the graphs for FF and FF-IW are identical; this means that the chosen window size ( $S = 64$ ) is so large compared to *RTT*, *IPG*, and *Err* that when the time for transmitting the  $i + 64^{th}$  data packet

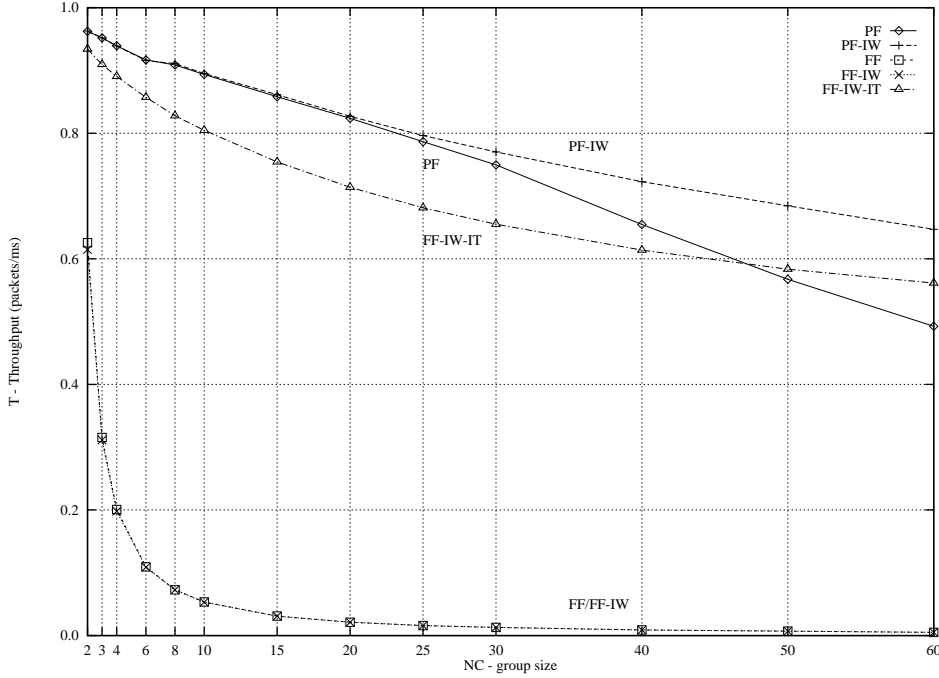


Figure 5: Throughput  $T$  in application packets/ms in the LAN configuration.

comes, the  $i^{th}$  packet has already been fully acked (i.e.,  $W_p$  never closes despite losses and recovery). It is clear from the graph that the throughput of FF and FF-IW degrades quickly as  $NC$  increases. The reasons are: (r1) the probability of a given data multicast not reaching at least one child increases with  $NC$ ; and (r2) implosion losses which increase with  $NC$ . Both (r1) and (r2) tend to increase the time it takes to get a packet fully acked. The graph for FF-IW-IT (which does not suffer from implosion) indicates that the poor scalability of FF/FF-IW is mainly due to (r2). The decrease in  $T$  for FF-IW-IT can only be due to (r1).

The effect of polling in containing implosion losses can be seen by comparing PF/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  $15 \leq NC \leq 40$ . This gain in  $T$  can be attributed to the “nack advantage” of PF protocols: the parent can detect a lost packet by receiving a nack within one RTT after the concerned child was sent a poll request. The FF protocol does not use nacks and relies on the absence of acks for loss detection. So the parent can detect a packet loss only at the expiry of the corresponding  $RTO$ . Note that both PF and FF protocols calculate  $RTO$  as twice the largest estimated  $RTT_i$ , for all  $R_i$  from which responses are expected. Hence when a small number of responses are lost **and** when poll requests are sent frequently, using nacks can help achieve faster error detection and recovery, and thus better throughput.

The  $T$  of PF is the same as that of PF-IW up to a point ( $NC = 8$ ) and starts decreasing thereafter, due to the following reason. In the PF protocols, as  $NC$  increases beyond a certain value, it takes longer to poll all children. Hence, the time to get a packet fully acked increases, and with a “small” window, the

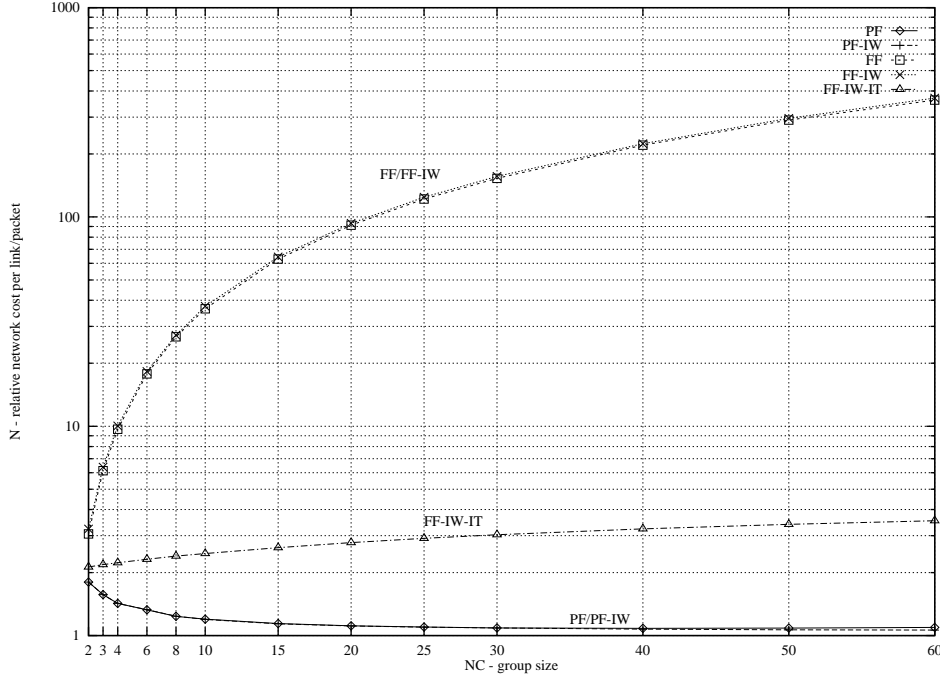


Figure 6: Relative network cost  $N$  in the LAN configuration.

transmission of new packets can get blocked more often.

The throughput gain achieved by PF and PF-IW is **not** at the expense of increased network cost, as illustrated in the Figure 6. Here again the graphs for FF and FF-IW are identical, indicating that the window size had no impact. The harmful effect of implosion losses on  $N$  can be seen from the gap between FF-IW and FF-IW-IT. The  $N$  for FF-IW-IT increases with  $NC$  because more losses are experienced as  $NC$  increases (due to the reason (r1) stated above), requiring more retransmissions. Since retransmissions are global multicasts, the cost per child increases. The relative network cost  $N$  for PF/PF-IW becomes suboptimal (close to 1) for  $NC \geq 10$ , because (a) the polling scheme reduces the amount of feedback from children, and thus decreases  $TP$ ; (b) the mechanism collects nacks to decide whether a multicast or unicasts can provide cost-effective recovery; and (c) it is also optimistic, in the sense that it does not interpret the absence of response as an implicit nack for those packets that could have been referred in the missing response (see section 2.4).

We have performed experiments to measure the value of  $I$  for the five protocols. The values for PF/PF-IW in both configurations were close to 0. For example, the values of  $I$  for the PF protocol in the LAN configuration are 0.01 for  $NC = 20$  and 0.007 for  $NC = 60$ . The  $I$  values in the HYBRID configuration are shown in Figure 7. The graphs for FF and FF-IW show that the chosen window size limits (unlike in the LAN configuration) the number of implosion losses, and that the difference between FF and FF-IW increases with  $NC$ . For the LAN configuration, the window size had no impact and the values for both FF and FF-IW were similar to those for FF-IW in the HYBRID configuration. Hence, the graphs  $I$  vs.  $NC$

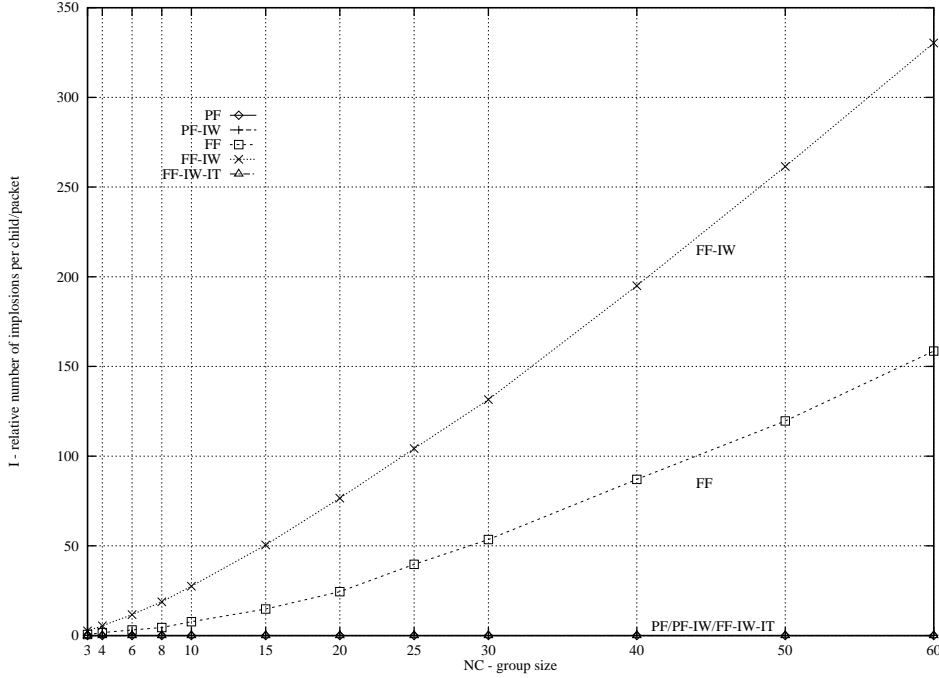


Figure 7: Relative implosion losses ( $I$ ) in the HYBRID configuration.

for the LAN configuration are not shown.

Figure 8 shows the throughput values for the five protocols in the HYBRID configuration. Due to the infinite window, FF-IW performs better than FF for small  $NC$  values. This advantage, however, becomes a disadvantage as  $NC$  increases, because of increased implosion losses. Hence, the performance of FF-IW rapidly becomes poor. The protocol FF provides consistently poor performance due to implosion losses and finite window. The performance of FF-IW-IT and PF-IW are similar. Note that the latter does not outperform the former like in the LAN configuration. We believe this could be attributed to higher  $Err$  values of WAN connections which can cause more RESP packets to be lost during transmission. This suppresses the nack advantage described earlier. PF performs better than FF/FF-IW due to negligible losses, and worse than PF-IW due to finite size window.

The relative network cost  $N$  for the HYBRID configuration is shown in Figure 9 (with  $N$  in log scale). As indicated in Figure 7, the chosen window size is small enough to limit implosion losses for FF and hence FF exhibits a lower cost compared to FF-IW. Observe that the differences in  $N$  for FF-IW and FF increase with  $NC$ , just like the differences in  $I$  increased in Figure 7. The  $N$  values for PF, PF-IW, and FF-IW-IT are close to 1, but not so close as they were in the LAN configuration. Further, there is a noticeable difference between PF-IW and PF for all values of  $NC$ . In PF-IW,  $W_p$  never closes and hence poll requests are guaranteed to be piggybacked with DATA packets (as DATAPOLL packets), except after all DATA packets have been transmitted. In PF, however, explicit POLL packets have to be sent when no DATA can be sent due to closed  $W_p$ . The effect of using explicit POLL packets on  $N$  is less pronounced in the

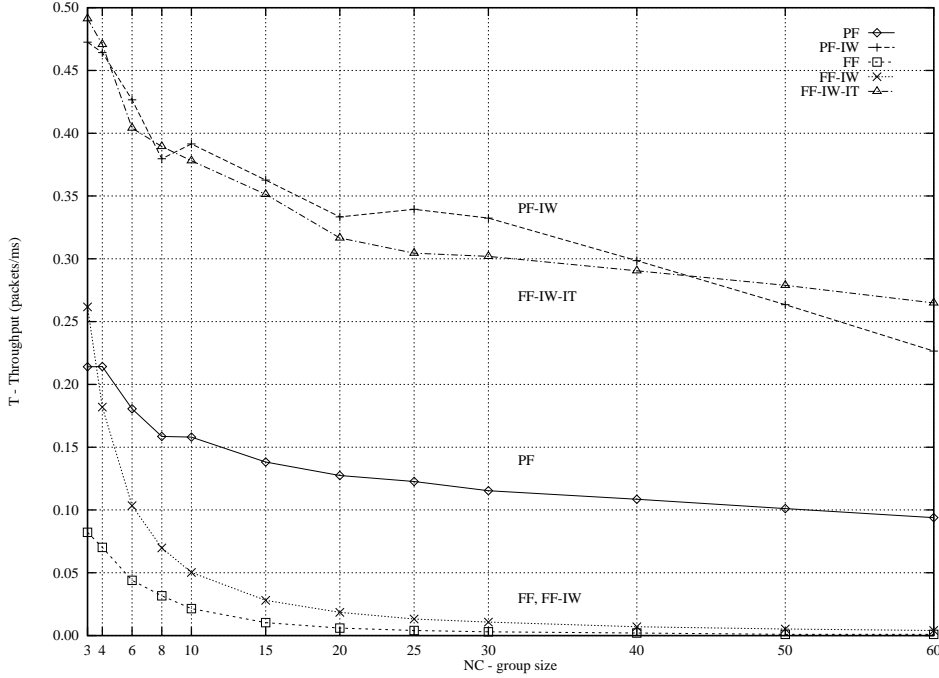


Figure 8: Throughput  $T$  in application packets/ms in the HYBRID configuration.

LAN configuration, where PF costs very marginally more than PF-IW for large values of  $NC$  ( $NC \geq 40$ ).

In all simulation results shown,  $RR$  is set to  $ITR$ . In our final set of experiments, we vary  $RR$  and assess the effect on  $T$  and  $N$  in LAN configuration for a large group (we fixed  $NC = 60$  children). Increasing  $RR$  has two major effects: (e1) poll requests to any given child are sent more frequently; hence a packet is likely to be acked sooner; (e2) as  $RR$  exceeds  $ITR$ , more response packets will be lost due to implosion, delaying loss recovery and packet acknowledgement. Note that (e1) and (e2) compete in promoting and suppressing nack advantage of PF protocols, respectively. Referring to Figure 10, the throughput for PF increases rapidly as  $RR$  approaches 1000, due to (e1); for  $1000 < RR \leq 1500$ , the rate of increase is reduced probably because (e2) comes into effect even before  $RR$  approaches  $ITR$ . Note that  $RR$  can be exceeded in certain epochs, due to uncertainties caused by the jitter (see section 2.3). For values of  $RR$  larger than  $ITR$ , (e2) is more dominant and  $T$  falls. Due to the nack advantage, PF-IW outperforms FF-IW-IT during  $600 \leq RR \leq 1500$ ; the  $T$  for PF-IW remains more or less flat and close to that of FF-IW-IT, for larger values of  $RR$ . This indicates that (e1) and (e2) cancel each other and little gain in  $T$  can be made by polling and using nacks when  $RR > ITR$ . The network cost  $N$  for PF and PF-IW is less than 50% of that for FF-IW-IT. The  $N$  for PF-IW increases with  $RR$  due to (e2). In PF, the use of explicit POLL packets sent to unblock  $W_p$  increases  $N$  further. The graphs for PF in Figure 10 and 11 indicate that  $RR$  should be set to an accurate estimate of  $ITR$  for maximum  $T$ . If  $ITR$  cannot be estimated accurately, an underestimation is preferable to overestimation.

Recall that our tests do not take into consideration the speed of the upper-level, since we assume

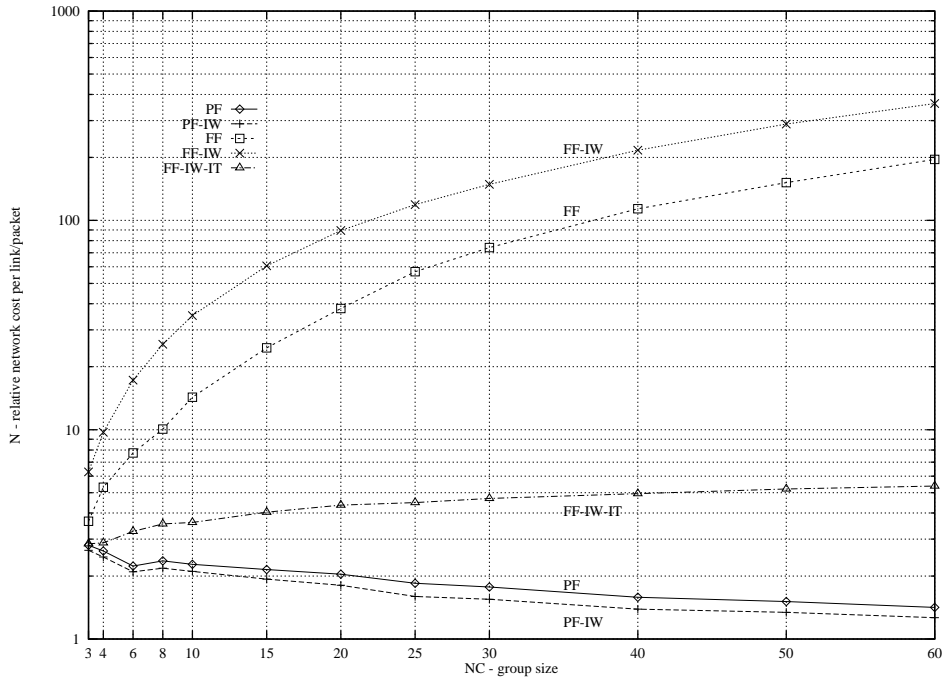


Figure 9: Relative network cost  $N$  in the HYBRID configuration.

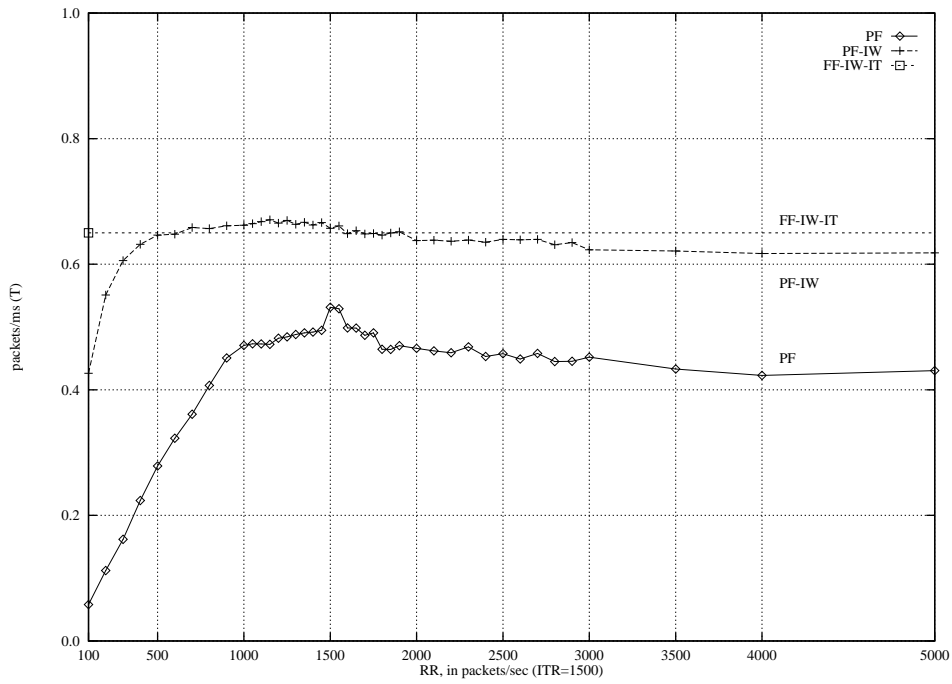


Figure 10: Effect of the  $RR$  value in the throughput  $T$  in the LAN configuration.

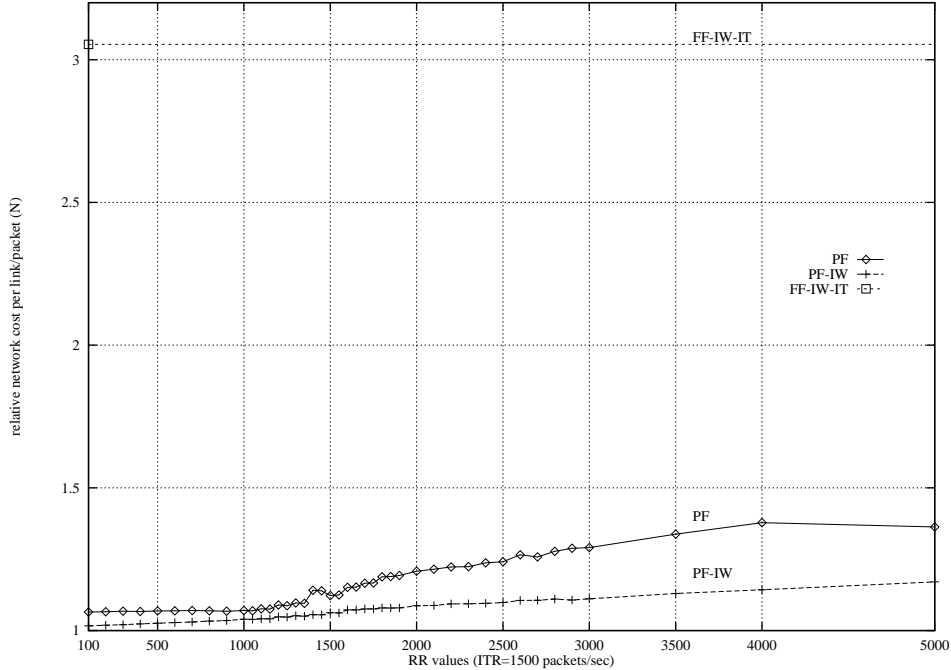


Figure 11: Effect of the  $RR$  value in the relative network cost  $N$  in the LAN configuration.

prompt production of data at the parent and “hungry” consumers (of data) at the children. This might cause a decrease in throughput for both PF and FF protocols, depending on how slow is the upper-level in producing or consuming data. We are currently analysing the effect of flow control in the protocol behaviour.

## 5 Concluding Remarks

The idea of using polling to avoid implosion is not new ([8]). In our paper we explore and extend this idea to develop a scheme to minimise implosion losses. The essence of our mechanism scheme is that the parent controls the arrival rate of responses by using information that is available or computable within the parent node. RTT delays between children can widely differ, though fluctuations in a given RTT can work against our scheme. We have described our protocol and the architecture to realise it, along with other mechanisms for error and flow control. Simulation analysis indicates that our scheme is indeed effective in reducing implosion losses. Although the processing per feedback packet is increased at the parent node, the amount of feedback packets can be immensely reduced. This in turn indicates that it is possible to develop scaleable protocols using the sender-based approach.

We envisage extending this work in two major directions. First, we plan to model and assess the processing cost at the parent and children nodes. Second, we will extend the single-level, one-to- $NC$  multicast protocol described here to the full multiple-level tree version (called PRMP- POLLING-BASED

RELIABLE MULTICAST PROTOCOL). We will investigate flow and congestion control strategies (such as [16]), and analyse the performance of PRMP through simulations. We also intend to analyse a variation of PRMP, where the sender multicasts the packets directly to all receivers and feedback packets are sent only to its immediate parent in the tree.

## Acknowledgements

Our thanks are due to Dr. Larry Hughes, who introduced us to the subject of polling-based implosion avoidance, while he was on sabbatical in Newcastle University. The paper also benefited from many discussions with Prof. Isi Mitrani regarding simulations. The financial support for the first author provided by CAPES (Brazil) is also *acked*.

## References

- [1] 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.
- [2] W. R. Stevens, “TCP/IP Illustrated, Vol. 1: The Protocols”. Chapter 21: TCP Timeout and Retransmission, Addison-Wesley Professional Computing Series, Addison-Wesley, 1994.
- [3] 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.
- [4] J. Crowcroft, K. Paliwoda, “A Multicast Transport Protocol”, ACM SIGCOMM’88, Stanford, 16-19 Aug. 1988.
- [5] 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.
- [6] 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.
- [7] M. Grossglauser, “Optimal Deterministic Timeouts for Reliable Scalable Multicast”, IEEE INFOCOM’96, San Francisco, California, March 1996.



- [8] L. Hughes, M. Thomson, "Implosion-Avoidance Protocols for Reliable Group Communications", Proc. of 19th Conf. on Local Computer Networks, Minneapolis, Minnesota, October 1994.
- [9] R. Yavatkar, J. Griffioen, M. Sudan, "A Reliable Dissemination for Interactive Collaborative Applications", ACM Multimedia'95.
- [10] J. Lin, S. Paul, "RMTP: A Reliable Multicast Transport Protocol", IEEE INFOCOM'96, March 1996, pp.1414-1424
- [11] 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.
- [12] 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.
- [13] 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.
- [14] I. Mitrani, Simulation Techniques for Discrete Event Systems, Cambridge Computer Science Texts 14. Cambridge University Press, 1982.
- [15] G.M.Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard, "Simula Begin", Petrocelli/Charter, New York, 1973, 391p.
- [16] V. Jacobson, "Congestion Avoidance and Control", Computer Communication Review, vol.18, no.4, pp.314-329, Aug. 1988.