Newcastle University e-prints

Date deposited: 4th April 2011

Version of file: Author final

Peer Review Status: Peer reviewed

Citation for item:

Further information on publisher website:
http://www.springerlink.com

Publisher’s copyright statement:
The original publication is available at www.springerlink.com at the following link:
http://dx.doi.org/10.1007/978-3-540-46703-8_11

Always use the definitive version when citing.

Use Policy:
The full-text may be used and/or reproduced and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not for profit purposes provided that:

- A full bibliographic reference is made to the original source
- A link is made to the metadata record in Newcastle E-prints
- The full text is not changed in any way.

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.
A Multicast Transport Protocol for Reliable Group Applications

Congue Liu,1 Paul D Ezihchelvan2 and Marinho Barcellos3
1Guangzhou Communications Institute, Guangzhou, 510310, P.R.China.
Congue.Liu@hw.com.cn
2University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK.
Paul.Ezihchelvan@newcastle.ac.uk
3UNISINOS, Sao Leopoldo, RS, 93022-000, Brazil.
Marinho@exatas.unisinos.ciee.br

Abstract. This paper presents a transport-level multicast protocol that is useful for building fault-tolerant group-based applications. It provides (i) reliable, end-to-end message delivery, and (ii) a failure suspector service wherein best efforts are made to avoid mistakes. This service can facilitate an efficient, higher level implementation of group membership service which does not capriciously exclude a functioning and connected member from the membership set. The protocol has mechanisms for flow- and implosion-control, and for recovering from packet losses. Through simulations, its performance is studied for both homogeneous and heterogeneous network configurations. The results are very encouraging.

1 Introduction

Building group-based applications capable of tolerating site crashes and network partitions has been under investigation for several years. Useful, programming paradigms such as view synchrony and virtual synchrony (VS) have been specified [1]. An efficient implementation of these abstractions can be obtained if the following low-level services are available: (ser1) multics from a group member (known as the sender) are received by other members (called receivers) in the sent order and with no message duplication and omission; and (ser2) if a receiver crashes or detaches due to partition, the sender is issued a failure-suspicion notice over that receiver.

A suspicion cannot always be correct as a slow, overloaded site cannot be distinguished from a crashed or detached one. So, false suspicions are admitted and are even acted upon as the only way to ensure liveness in applications. However, raising false suspicions must be avoided as much as possible. This, we believe, can be better achieved in a cost-effective manner if ser2 is built in the same level as where ser1 can be efficiently provided. The reasons for this are as follows. The sender’s suspicion becomes false when the timeout it employs to suspect a receiver failure becomes smaller than the round trip time (rtt) between itself and the receiver. That is, the sender’s ability to make correct suspicions depends solely on the sender having an accurate estimate of rtt. The rtt of course varies
with the network load and congestion levels. A multicast protocol that provides end-to-end reliability (ser1 above) will be constantly estimating up-to-date rtt's for detecting packet losses. Given that the quality of service offered by ser2 depends on the accuracy of the rtt estimates being used, it seems only natural to build both ser1 and ser2 together at the same level.

Systems that implement VS abstractions tend to build ser1 using standard, low-level communication services and then implement ser2 at a higher level (typically using pings). For example, Horus[2] builds on UDP/IP multicast to obtain end-to-end reliability; Newtop[3] and Phoenix[4] use multiple TCP unicasts; and Transis[5] employs Trans protocol[6] designed for broadcast mediums. This is probably because many well-known, end-to-end, reliable multicast protocols in the literature do not provide ser2 as they are also designed with scalability in mind. Consequently, they strive to relieve the sender from the burden of keeping excessive state information regarding receivers, in particular from the task of having to ensure that every functioning receiver receives the transmitted packets. In the receiver-initiated protocols, such as SRM[7], LBRM[8], receivers are responsible for detecting and recovering from packet losses; so, the sender cannot know which receivers are receiving the transmitted packets and which ones have crashed or detached. In the sender-initiated protocols, such as RMTP[9], the sender plays an active role in loss recovery only when receivers inform it of packet losses; it is not made aware of the entire receiver set (for reasons of scalability) and hence it cannot suspect crashed or disconnected receivers.

We here present a sender-initiated, transport-level multicast protocol that provides both ser1 and ser2 for a generic network topology. Our protocol is designed exploiting two characteristics of fault-tolerant group applications: the group size is usually small, so the sender can afford to keep a fair amount of state information per receiver; secondly, the sender knows the full membership of the receiver set. With this knowledge, it retransmits a packet until each receiver sends an ack (i.e., acknowledgement) for the packet or gets suspected.

A common problem to be tackled in the sender-initiated approach is the ack-implosion [10], [11] which is an acute shortage of resources caused by the volume and synchrony of incoming acks at the sender, resulting in ack losses and hence increased network cost caused by unnecessary retransmissions. Our protocol employs an effective mechanism to control implosion, and also provides flow control. Its design is motivated by the performance study of our earlier protocol PRMP [12], [13]. Compared to PRMP, the protocol presented here is simple and hence easier to implement; incurs far less computational overhead at the sender and less network cost; it however achieves a smaller throughput. Its performance and cost-effectiveness are demonstrated through simulations on both homogeneous and heterogeneous networks.

We assume the following system context: the transmission phase, during which the data packets are transferred from the sender to receivers, is preceded by the connection setup phase. The transmission phase is the focus of the paper and the connection setup is assumed to help accomplish an unreliable multicast
service that efficiently propagates the sender's data packets to the connected destinations. This unreliable service may be achieved through a series of unicast transmissions or a single IP multicast. (Both cases are simulated.) A receiver can unicast to the sender, can crash or detach from the sender. The rest of this paper is organized as follows. Section 2 describes the design and mechanisms of the protocol. Simulation results are given in Section 3 which are followed by concluding remarks in section 4.

2 Protocol Description

2.1 Overview

The protocol solves the problem of a non-faulty sender having to reliably transmit a number of data packets to a destination set of N receivers. Failures during transmission result in packets being dropped or corrupted and discarded. To deal with packet losses, the sender must keep a copy of each transmitted packet in its buffer and keep retransmitting a packet until an acknowledgement for that packet is received from every receiver. However, no acknowledgement can be received from a crashed or detached receiver. So, to ensure termination, the sender retransmits a given packet only for some specified number of times. A receiver that has not indicated the receipt or non-receipt of the packet after all these retransmissions is suspected to have failed. Once a packet is acknowledged by all unsuspected receivers, the sender regards it to have been fully acknowledged and removes it from the buffer.

Receivers indicate the receipt/non-receipt of a packet through their responses unicast to the sender. Simultaneous arrival of many such responses can cause implosion and the sender may miss receiving some of them. To minimise implosion losses, the sender sends Feedback Information (FI for short) to receivers at regular intervals called the Cycle. FIs essentially tell receivers when to unicast their responses. When a receiver receives an FI, it delays for a certain amount of time and then sends a response once every Period. (Both the delay and the Period are indicated in the FI.) This continues till another FI is received which initiates (after an indicated delay) another episode of periodic responses from the receiver.

The sender estimates the delay time (specific to a given receiver) and the Period (common to all receivers) based on its estimation of rtt's for the receivers. If this estimation is, and remains, accurate, the sender must receive one response from each receiver during the Period and the time gap between the arrival times of any two consecutive responses will be constant. In other words, the sender should receive the responses in a non-bursty manner and at a rate that cannot cause implosion, provided that the sender can precisely estimate rtt's and that the rtt's change very little over time. In practice, however, rtt's seldom remain constant even during short periods of time. The protocol copes with the variations in rtt in the following manner: (a) the sender obtains a fresh estimate of rtt every
time it receives a response from a receiver and thereby its rtt estimates are kept reasonably up to date; and, (b) it uses the most recent rtt estimates to compute FI parameters at the start of every Cycle and receivers employ a mechanism to tolerate small fluctuations of rtt which might occur during Cycle.

The sender observes a retransmission timeout (RTO) before it retransmits a packet that is considered to be lost during previous transmissions. RTO is fixed based on the sender's estimate of rtt so that premature retransmissions are avoided. Further, a retransmission can be either a multicast or a series of selective unicasts, depending on which option is deemed economical in terms of message cost.

The protocol employs a sliding window scheme for flow control and for detecting packet losses. It involves the use of three types of packets which can be exchanged between the sender and the receivers: (a) DATA packets are sent by the sender and contain a sequence number seq that uniquely identifies a packet; (b) FI packets, also sent by the sender, contain a unique sequence number F1seq; and, (c) response packets, RESP, sent by a receiver indicate to the sender which data packets have been received and which ones have not been. (We assume seq/F1seq is large enough to avoid problems that arise when sequence number are wrapped around and reused.) A DATA packet (FI packet respectively) is said to be earlier than another DATA packet (FI packet respectively) if the former has a smaller sequence number.

2.2 Protocol Details

2.2.1 Sliding Window

Packet loss detection and flow control in our protocol are based on a sliding window scheme with fixed-size data packets. Both the sender and receivers employ a buffer of size S packets, negotiated at connection setup, and kept constant during the whole transmission. The protocol, at the receivers, is required to sequentially deliver the data packets received. At a receiver, a data packet is said to be unconsumable if an earlier packet has not been received. Further, the upper-layer may be slow and consequently some packets ready for consumption may remain unconsumed.

Each receiver keeps a receiving window W_{ij} that is characterised by buffer size S, a left edge LE_{ij}, and the highest received sequence number HR_{ij} from the sender. LE_{ij} is the minimum between the sequence number of the earliest unconsumed packet in R_{ij} and the sequence number of the earliest packet yet to be received by R_{ij}. Thus LE_{ij} refers to the smallest sequence number of the packet that is either waiting to be consumed or expected to be received. W_{ij} is a boolean vector indexed by seq, LE_{ij} \leq seq \leq LE_{ij} + S - 1: W_{ij}[seq] is true if R_{ij} has received the data packet seq, or false otherwise. HR_{ij} is set to seq of a data packet received from the sender if seq > HR_{ij}.

The sender keeps a set of NC sending windows, one W_{pi} for each receiver. W_{pi,j} is the sender's (latest) knowledge of W_{ij} of R_{ij}. Like W_{ij}, it is characterised by size
$S$, a left edge, denoted as $LE_{p,i}$ and the highest received sequence number $HR_{p,i}$. $LE_{p,i}$ and $HR_{p,i}$ are the sender's knowledge of $LE_i$ and $HR_i$, respectively. For the data packet $seq$, $LE_{p,i} \leq seq \leq LE_{p,i} + S - 1$, $W_{p,i}[seq]$ indicates the sender's knowledge of whether $R_i$ has received $seq$; it is initially set to false. Finally, the sender keeps the variable $HS$ to record the largest $seq$ of data packets it has multicast so far.

When it is time to respond, $R_i$ sends a $RESP$ packet to the sender containing (a) $RESPW$, which is the copy of its receiving window; (b) $RESPW.LE$ which contains the value of $LE_i$; (c) $RESPW.HR$, the value of $HR_i$; and, (d) a timestamp $RESPTs$ which is used by the sender to estimate the Round-Trip time ($RTT$ for short).

When the sender receives a $RESP$ packet from $R_i$, it updates its variables related to $R_i$, $LE_{p,i} \leftarrow \max\{LE_{p,i}, RESPW.LE\}$, $HR_{p,i} \leftarrow \max\{HR_{p,i}, RESPW.HR\}$ and then, for all $seq$, $RESPW.LE \leq seq \leq RESPW.HR$, $W_{p,i}[seq] \leftarrow W_{p,i}[seq] \vee RESPW[seq]$. From $W_{p,i}$, the sender can infer that $R_i$ has received all data packets with $seq, seq < LE_{p,i}$ or $W_{p,i}[seq] = true$, and missed those 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 probably still in transit towards $R_i$. Based on these inferences, the sender detects packet losses.

### 2.2.2 Flow Control

Our protocol employs window-based flow control. Since data packets are multicast to the entire destination set, the receiver with the smallest number of free buffer spaces determines the number of new packets which can be multicast. The sender determines the effective window ($EW_{p,i}$) for each $R_i$, where $EW_{p,i}$ denotes the number of new packets $R_i$ can take without buffer overflow: $EW_{p,i} \leftarrow (LE_{p,i} + S) - (HS + 1)$. Without causing buffer overflow at any of the receivers, $EW_{p}, EW_{p} \leftarrow \min\{EW_{p,i} | \forall 1 \leq i \leq NC\}$, new packets can be multicast. When $EW_{p}$ is zero, the sender is said to be blocked. 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 is the minimum interval that must elapse between two successive transmissions from the sender.

### 2.2.3 Feedback Information (FI) and Related Parameters

The sender divides time into epochs of fixed length $\varepsilon$ which is known to all receivers. Epochs are denoted as $E_n$, with $n = 0, 1, 2...$; $E_n$ is the time interval between $n \times \varepsilon$ and $(n + 1) \times \varepsilon$.

An $FI$ packet multicast by the sender contains: (a) an array $Rdel$ with $Rdel[i]$, $1 \leq i \leq NC$, indicating the delay $R_i$ should observe before sending its first response after it receives $FI$; (b) $Period$, which is the time interval that should (ideally) elapse between the arrival of two successive responses from any given receiver. $Period$ is identical to all receivers and is expressed as the number of epochs; (c) $ts$, which indicates the sender’s local time when the $FI$ packet is
sent; and, (d) \( FI_{seq} \), which indicates the sequence number of the \( FI \) packet and helps a receiver to detect duplicate \( FI \). The sender periodically sends a new \( FI \) to receivers and this period is called \( Cycle \), \( Cycle = k \times \text{Period} \) for some \( k \gg 1 \).

![Diagram of FI packets and timing](image)

Figure 2.1. Receiver response times and \( FI \) parameters.

Figure 2.1 shows two successive \( FI \) packets, \( FI_1 \) and \( FI_2 \), from the sender. These packets are received by \( R_i \) at (local) times \( t_1 \) and \( t_2 \) respectively, and contain parameters \( \{Rdel_1, \text{Period}_1, FI_{seq} = 1, ts_1\} \) and \( \{Rdel_2, \text{Period}_2, FI_{seq} = 2, ts_2\} \) respectively. \( R_i \) sends its first response as per \( FI_1 \) at time \( t_1 + Rdel_1[i] \) and subsequently once every \( \text{Period}_1 \). This continues until \( t_2 + Rdel_2[i] \) when \( R_i \) sends its first response as per \( FI_2 \). After \( t_2 + Rdel_2[i] \), \( R_i \) responds once every \( \text{Period}_2 \) until it begins to respond as per the third \( FI \) packet it may receive later. The cycle thus repeats.

### 2.2.4 Sender’s Estimation of \( FI \) Parameters

If the rate of responses arriving at the sender exceeds a given threshold, losses can be expected due to implosion. This threshold is called the implosion threshold (ITR). It depends on the buffer space and the processing capacity currently available within the sender node, and hence may change if, for example, a new multicast session is opened or a concurrent one is closed. To avoid losses by implosion, the protocol controls the arrival rates and arrival timings of response packets using an input value response rate (RR). Thus, RR is the protocol’s knowledge of ITR. It should not exceed ITR if implosion is to be avoided; if it is too small compared to ITR then the sender is processing the receiver responses below its capacity and this will decrease the throughput. So, the objective should be to have RR tracking ITR. In our simulation study, we analyse the effect of RR exceeding ITR on implosion losses.

The sender computes \( FI \) parameters to meet two objectives. First, it aims to receive within every epoch the maximum number of responses permitted by RR. That is, it plans to receive \( RQ \) responses, \( RQ \leftarrow \lfloor RR \times \varepsilon \rfloor \). Secondly, within
Period, it plans to receive one response from every receiver. So, $\text{Period} = \lfloor NC/RQ \rfloor \times \varepsilon$. It computes $\text{Rdel}$ such that these objectives are met.

For now, assume that the sender knows $\text{RTT}$ for every receiver (see section 2.2.6). It orders $\text{RTT}$ values in non-increasing manner: $\text{RTT}_1 \geq \text{RTT}_2 \geq \ldots \geq \text{RTT}_{NC}$. Let $n$ be the smallest integer such that $n \times \varepsilon - \text{RTT}_1 > \text{current time}$. Say $t = n \times \varepsilon - \text{RTT}_1$. The sender is to multicast the $FI$ packet at its clock time $t$. It plans the first response (as per the $FI$ to be sent at $t$) from $R_1$ to arrive at $n \times \varepsilon$. So $\text{Rdel}[1]$ is set to zero. The first response from $R_2$ (as per the $FI$ to be sent at $t$) is expected to arrive at $n \times \varepsilon + \varepsilon / RQ$. So, upon receiving the $FI$ (to be sent at $t$), $R_2$ should be instructed to delay sending its first response by $\text{Rdel}[2] = n \times \varepsilon + \varepsilon / RQ - (t + \text{RTT}_2)$. In general, $\text{Rdel}[i] = n \times \varepsilon + (i - 1) \times \varepsilon / RQ - (t + \text{RTT}_i)$. Figure 2.2 depicts the rationale behind the sender’s estimation of $\text{Rdel}$ for $NC = 3$, $RQ = 1$, and $\text{RTT}_1 > \text{RTT}_2 > \text{RTT}_3$.

By estimating and sending new $FI$ for every $Cyde$, the sender accounts for changes occurred in $\text{RTT}$ and $RR$ during the past $Cyde$, and also for any changes in the receiver set due to exclusion of a receiver that is suspected to have crashed or disconnected from it. Further, receivers are programmed to cope with small fluctuations of $\text{RTT}$ around the sender’s $\text{RTT}$ estimate used at the beginning of $Cyde$ (described in section 2.2.7). Thus, maximum effort is being made for responses to arrive at the sender within the planned timing interval. Observe that the receivers need not know the value of $Cyde$ used by the sender. So, the sender can be made more responsive to changes in $\text{RTT}$, $RR$ or in receiver set by sending new $FI$ immediately after it detects these changes (instead of waiting for any remaining part of $Cyde$ to be over).
2.2.5 Reducing Redundant Transmissions

When the sender sends no new data packets for a long period (say, due to being blocked), successive responses from a receiver could be identical, and some of these responses may well be redundant. To save bandwidth, Ri sends no response until Wj changes, if it has already sent x identical responses in succession; we assume that at least one of these x responses will reach the sender. If q is the probability that a given packet can be lost due to network error, then the probability that at least one of x packets would reach the sender will be \((1 - q^x)\). The sender also multicasts a given FI packet x times, with an interval Period between two successive FI transmissions. Note that FI parameters need not be recomputed before retransmission due to the chosen interval Period between two successive transmissions of a given FI.

2.2.6 RTT (round trip time) Estimation

The sender and receivers use their own clocks which need not be synchronised i.e., the clocks may read different values at any given time. To deal with this clock asynchrony, Ri maintains a clock-difference counter C_Diff. Recall that the field ts in a DATA or FI packet indicates the time when that DATA or FI packet respectively is sent according to the sender’s clock. Whenever Ri receives a DATA or FI packet, it sets C_Diff to the local clock time when the packet is received minus the value of ts in the received packet. When Ri is to send a RESP packet at its clock time T_RESP, it computes RESP.ts to be T_RESP = C_Diff. When the sender receives a RESP packet from Ri at its clock time clock, it computes RTT_i = clock - RESP.ts.

Figure 2.3 explains the rationale behind the sender’s estimation of RTT_i and assumes that the last packet Ri received before it decides to send RESP at T_RESP is an FI packet. As per the figure, RTT_i = t_1 + t_3. Since C_Diff = T_FI - F_I.ts, RESP.ts = T_RESP - C_Diff = F_I.ts + t_2. So, RTT_i = clock - RESP.ts = t_1 + t_3.

Note that the sender keeps no state information for RTT_i estimation and obtains a fresh estimate of RTT_i for every RESP it receives from Ri.

![Figure 2.3. Estimation of RTT.](image-url)
2.2.7 Handling absent responses and lost packets

Both data packets and response packets can be lost during transmission. To avoid waiting for ever on receiving a response from a given \( R_i \), the sender also waits on a retransmission timeout (RTO) after having transmitted a given packet. \( RTO \leftarrow \max\{RTT_i | 1 \leq i \leq N\} + x \times Period \). A receiver that neither ACKs nor NACKs the transmitted packet during RTO is regarded to be an absentee for that packet. The packet is retransmitted to every absentee for a maximum number of times, with each transmission followed by waiting for a multiplicatively increased RTO to see whether \( RESP \) can be received. (This maximum number and the multiplicative factor can be specified as parameters of the protocol.) A receiver that remains as an absentee despite all these retransmissions is removed from the receiver set. This removal is indicated to the upper layer in a failure – suspicion exception. Note that removing a persistently non-responsive receiver from the set of receivers is necessary to prevent the sender from being indefinitely blocked (see flow-control, section 2.2.2). Once a receiver is removed, any response received subsequently from it is ignored.

The explanation behind RTO estimation and failure suspicion are as follows. Recall that \( R_i \) gets a chance to send its \( RESP \) once every Period. So, in the absence of failures, the first \( RESP \) sent by \( R_i \) after receiving a given packet, should reach the sender within \( RTT_i + Period \). Since at least one of \( x \) consecutive responses sent by \( R_i \) is likely to succeed, the sender waits for \( (RTT_i + x \times Period) \) for every \( R_i \). The sender retransmits a packet to an absentee receiver for a specified, maximum number of times. We assume that if the absentee is functioning and connected to the sender, at least one of the attempts will succeed; if all attempts fail, the absentee is suspected to be crashed/detached and is removed from the receiver set.

Changes in \( RTT \) during Cycle can cause a receiver’s response to arrive earlier or later than the planned time. To deal with small \( RTT \) changes during Cycle, \( R_i \) employs a second clock-difference counter \( C_{Diff} FI \) which is updated only when an \( FI \) is received: \( C_{Diff} FI = T_{F_i} - F_i.t_s \), where \( T_{F_i} \) is the local time when \( R_i \) received \( F_i \). Note that receiving an \( FI \) makes \( C_{Diff} \) and \( C_{Diff} FI \) have the same value, and subsequent arrival of a \( DATA \) packet may make \( C_{Diff} \) differ from \( C_{Diff} FI \). After \( R_i \) has updated \( C_{Diff} \) following the arrival of a \( DATA \) packet, it computes \( T_{adj} = C_{Diff} - C_{Diff} FI \). \( T_{adj} \) indicates half of the increase in \( RTT \) since \( R_i \) received the last \( FI \), and hence approximately half the amount of increase in \( RTT \) over the \( RTT_i \) estimate used by the sender in computing \( FI \) parameters.

Soon after sending a given \( RESP \), \( R_i \) sets \( T_{send} \) to the time when it has to send the next planned \( RES \); it sets \( T_{send} = T_{send} - T_{adj} \) if \( T_{send} - T_{adj} \) is larger than the current time. The \( RES \) planned to be sent at \( T_{send} \) is actually sent at \( T_{send} \), and new values for \( T_{send} \) and \( T_{send} \) are computed after \( RES \) is sent at \( T_{send} \). \( T_{send} - T_{adj} \) is not a future time means that the fluctuations of \( RTT \), since the sender last sent its \( FI \) are so large that compensation is not possible.

Next we discuss how the sender deals with the packets known to be lost. After
transmitting a packet seq at local time t, the sender performs the first recovery in one the following two ways. (i) if the number of receivers that indicate NACK for seq reaches or exceeds the multicast threshold MTR × NC, the packet seq is multicast; or (ii) if, at local time t + RTO, the number of absentees and the receivers that NACK seq reaches or exceeds MTR × NC, seq is multicast; otherwise, seq is unicast to each absentee and to every NACKing receiver. After the first recovery, seq is only unicast to any receiver that still remains as an absentee or keeps NACKing. This is because the number of receivers requiring retransmission after the first recovery is expected to be small.

3 Simulation Results

To evaluate the performance of our protocol, we carried out simulation experiments under various settings. These experiments are undertaken on two different network topologies: single-level tree and multi-level tree. In the single level case, each receiver is connected directly to the sender and the sender’s multicast is realised through multiple unicasts - one for each receiver. In the second case, a receiver is connected to the sender either directly or via multicast-enabled routers and the sender uses IP multicast. In both cases, a receiver addresses its RESP packets to the sender; the applications at receivers are assumed to be message-hungry: the received packets are consumed as soon as they become ready for consumption. The results of multi-level topology are omitted for space reasons but can be seen in section 3.2 of the full paper [15].

Three parameters are evaluated: throughput T, relative network cost N, and relative implosion losses I. If D bytes of data are to be transmitted, and the packet size is P, the number of data packets to be transmitted, DP, can be defined as: $DP = \lfloor D/P \rfloor$. 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 acknowledged (both events occurring at the sender), the throughput is calculated as $T = DP/\Delta t$, in packets/ms. N is calculated as the total number of packets exchanged (TP) per receiver per data packet, i.e., $N = TP/(NC \times DP)$; the ideal value for N is $(DP + 1)/DP$ (at least one ack per receiver 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.

In the network model we use in the experiments, losses are assumed to be independent. Each packet has a destination address, which may be a unicast or multicast address. Each receiver is uniquely identified by a “fictitious” network address. The implosion losses are simulated in the following manner: an incoming response is to be stored in an incoming queue (IQ) before being processed; it is considered to be lost if there is no space in IQ when it arrives. The size of IQ is 64 packets.
3.1 Simulation on Single-Level Tree

Series of experiments were conducted for different values of NC. The network was modeled as a set of channels directly connecting the sender and receivers. Three types of channels are considered: short, medium and long. Each channel type is characterised by a set of three attributes: normally distributed propagation latency with mean $L$, latency standard deviation (to emulate jittering) $SD$, and the percentage error rate $Err$. The values associated with each type are listed in Table 3.1. Two network configurations are defined: (a) LAN in which all channels are of type short; (b) HYBRID (HYB in figures) where at least $[NC/3]$ receivers are connected to the sender by channels of a given type.

<table>
<thead>
<tr>
<th>Channel Type</th>
<th>$L$ (ms)</th>
<th>$SD$ (ms)</th>
<th>$Err$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>1.5</td>
<td>0.08</td>
<td>1%</td>
</tr>
<tr>
<td>medium</td>
<td>5</td>
<td>0.5</td>
<td>1%</td>
</tr>
<tr>
<td>long</td>
<td>75</td>
<td>15</td>
<td>10%</td>
</tr>
</tbody>
</table>

Table 3.1. General properties for three channel types.

<table>
<thead>
<tr>
<th>Input</th>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data unit size</td>
<td>unitSize</td>
<td>1000 bytes</td>
</tr>
<tr>
<td>transmission size</td>
<td>$DP$</td>
<td>1000 packets</td>
</tr>
<tr>
<td>inter-packet gap</td>
<td>$IPG$</td>
<td>1 ms</td>
</tr>
<tr>
<td>epoch length</td>
<td>$\varepsilon$</td>
<td>10 ms</td>
</tr>
<tr>
<td>response rate</td>
<td>$RR$</td>
<td>1500 RESP / s</td>
</tr>
<tr>
<td>uni vs multicast threshold</td>
<td>$MTR$</td>
<td>20%</td>
</tr>
<tr>
<td>number of same responses</td>
<td>$x$</td>
<td>2 RESP$^s$</td>
</tr>
<tr>
<td>FI Cycle</td>
<td>$Cycle$</td>
<td>10 Periods</td>
</tr>
</tbody>
</table>

Table 3.2. Protocol inputs used in single-level tree configuration.

To assess the impact of window size ($S$) on performance, we employ two values for $S$: 64 packets, and 1000 packets, the latter being large enough to represent the window of infinite size. Table 3.2 shows the value of all input parameters used in the experiments. Unless specified otherwise, $RR = ITR$. From table 3.2, $RQ = [RR \times \varepsilon] = 15$. $Period = \lfloor NC/RQ \rfloor$ epochs.

3.1.1 Comparison with Full-Feedback Protocol

To illustrate the relative performance of our protocol, we compare it with the “standard” sender-initiated protocol – Full-Feedback protocol (or FF for short). FF is a multicast extension of TCP and the details can be seen in [14]. Its main characteristics are: (a) it employs a sliding window scheme with selective
retransmission (i.e., no go-back-N); (b) receivers instantly acknowledge every packet they receive; (c) loss detection is timeout-based, and loss recovery via global multicasts (i.e., $MTR \times NC$ is 1).

![Figure 3.1. Relative implosion rate, $I$.](image1)

![Figure 3.2. Throughput, $T$.](image2)
Figures 3.1, 3.2 and 3.3 show the simulation results of $I$, $T$, $N$, respectively. (A marked point in all the graphs is the average taken over 6 experiments.) The graphs named FW-XXX and IW-XXX indicate the performance of our protocol with finite and infinite widow size respectively, in the network configuration XXX which can be either LAN or HYB; the ones named FF-FW-XXX and FF-IW-XXX indicate the corresponding performance of FF protocol. These figures indicate that our protocol performs much better than FF in all three parameters concerned. With near-zero implosion losses in our protocol (see fig. 3.1), packets get fully acknowledged sooner, resulting in higher throughput as shown in figure 3.2 by the (widening) gap between the graphs of our protocol and FF. For both protocols $T$ decreases as $NC$ increases because the probability of a given multicast not reaching at least one receiver (thus requiring recovery) increases with $NC$.

![Graph showing relative network cost, $N$.]

The relative throughput gains in our protocol are not achieved at the expense of increased $N$, as illustrated in figure 3.3. With our protocol, $N$ decreases with increasing of $NC$ while FF has the opposite tendency. This is due to two reasons: our implosion control mechanism reduces the amount of responses sent by receivers, while in FF receivers ack immediately after receiving a packet; FF employs only multicasts for lost packet recovery whereas we decide judiciously between multicast and unicast. So, the total number of packets used in FF increases more compared to the increase in $NC \times DP$. 

$\text{Figure 3.3. Relative network cost, } N.$
3.1.2 Performance in Hybrid Network

The graphs named FW-HYB and IW-HYB of figures 3.4 and 3.5 show respectively $T$ and $N$ of our protocol in the hybrid network configuration. (The graphs MAX _ T and IW-HYB-NOERR will be discussed in the next subsection.) Note the gaps between FW and IW in both figures. FW provides less than half the throughput of IW, because in FW transmission of data packets is restricted by window size, while in IW data can be continuously transmitted at a rate of $1/IPG$. As for difference in $N$ between FW and IW, $T$ in IW is much higher than $T$ in FW. That means the time needed to transmit same amount of data packets is much shorter than in IW. So more FI packets and RESP packets are used, hence, increasing $N$.

Simulation results also show that the value of $x$ chosen affects the performance of the protocol. When $x$ is equal to 10, both $T$ and $N$ are worse than those when $x$ is 2. So, the smaller $x$, the better is the performance. In our experiments, the worst $Err$ is 10%. Using the formula discussed in section 2.2.5, $x = 2$ means that there is at least 99% chance that at least 1 out of 2 responses from a connected receiver reaches the sender.

![Figure 3.4. Throughput T in the hybrid configuration.](image)

3.1.3 Impact of IQ size and RR on implosion Losses

In all the experiments presented so far, the implosion losses were nearly zero when IQ size is 64 packets. To evaluate the impact of IQ size on implosion,
we fixed it to 3 and chose parameters that would maximise the possibility of implosion: we considered the hybrid case, where the variation in RTT is higher which provides larger scope for the FI parameters computed at the beginning of Cyde to become incorrect, thus causing the responses to arrive outside the planned interval; infinite window (IW) was assumed so that the sender is never blocked due to flow control and hence it is subject to implosion possibility to the maximum extent.

![Relative network cost graph](image)

**Figure 3.5. Relative network cost N in the hybrid configuration.**

Until NC = 20, the implosion loss was zero and thereafter it became non-zero but still negligibly small. The largest I we observed was 7e − 5 for NC = 30, and the average I (over six experiments) for NC = 30 was 1.667e − 5. This near-zero I can be attributed to (i) the effectiveness of our implosion control mechanism, and (ii) the RR used in the estimation of FI parameters is the same as ITR. In practice, however, ITR cannot be accurately estimated and can vary with time. These experiments nevertheless lead us to conclude that if the RR used for estimating FI parameters is accurate, our implosion control mechanism transforms the system, at least for small NC, into a system of infinite IQ size which can suffer no implosion. However, the mechanism, unlike the system with infinite IQ, extracts its price in terms of reduction in T and increase in N. We estimate this cost by fixing Err = 0%. That is, any loss that occurs can only be due to implosion. The IW-HYB-NOERR graphs in figures 3.4 and 3.5 show respectively T and N of our protocol for IQ size = 3 packets, Err = 0%, IW in the hybrid case.
In a system with an infinite-length IQ, the maximum throughput achieved when $Err = 0\%$ is $DP/DeltaT$, where $DeltaT =$ time for the sender to multicast all packets + time to get ack for all packets from all unsuspected receivers. The first term is $DP \times IPG$ (due to IW), and the second term is $2 \times RTT_{max}$ if we assume that each receiver collectively acks/nacks only after receiving an end-of-transmission packet from the sender. Thus the maximum $T$ achievable in a loss- and implosion-free environment is shown as $MAX_T$ in fig. 3.4, and $N$ in such an environment is nearly 1. The IW-HYB-NOERR case indicates the cost of our implosion control scheme: the smallest $T$ is about 50% of $MAX_T$ and the maximum $N$ is just below 1.9.

Observe the "humps" in Figure 3.5: $N$ increases with $NC$, during the first interval $3 \leq NC \leq 15$, falls sharply during the second interval $15 < NC \leq 30$ and falls again during interval $30 < NC \leq 45$. The explanation for this lies in the value chosen for the jitter of long channels ($SD = 15$) and the value of Period during these intervals which is 10 ms, 20 ms, 30 ms respectively. Note that Period does not change with $NC$ during a given interval, and that $RTO = \max\{RTT_i\}, 1 \leq i \leq NC\} + x \times Period$. So, the smaller the Period, the smaller is RTO and hence the larger is the probability that a response is sent along a long channel reaches the sender after RTO. Responses that do not arrive before RTO trigger retransmission -thus increasing $N$. For small values of $NC$ in the first interval, $N$ increases sharply. This is because: when at least $MTR \times NC$, $MTR \times NC = NC/5$, of $NC/3$ receivers that are connected by long channels cause RTO timeouts for a given packet, the packet is multicast to all receivers.

Our final set of experiments analysed the effect on $I$ due to the variation between $RR$ (used in estimating $FI$ parameters) and $ITR$: the value $RR$ should ideally track. Note that when $ITR$ is larger than $RR$ used, it does not lead to implosion as buffer spaces are under utilised. So, with buffer size = 1, we fixed $RR$ at 1.5 packets/ms and estimated $I$ by varying $ITR$ for $NC = 30$. The values of $I$ observed were 0.00213 and 0.00047 when $ITR$ was 0.5 and 1.00, respectively.

4 Conclusions

We have presented a transport-level multicast protocol that provides (i) reliable, end-to-end message delivery; and (ii) a failure suspecter service wherein the best efforts are made to minimise mistakes. The simulations indicate that objective (i) is met with good throughput and low network cost; the implosion control mechanism employed effectively minimises implosion losses. As the sender estimates up to date round trip times ($rtts$) for every arrival of a response from receivers, it is aware of $rtt$ variations and hence can make fewer mistakes while suspecting a receiver failure. This is very useful for building a group membership service that does not remove a functioning member capriciously. Building a failure-suspecter at the lower-level and, thereby, facilitating efficient provision of fault-tolerance at the application levels are a novelty of this paper.
The Trans protocol [9], like ours, also provides some basic services at the lower level which are useful for building important higher level services: it includes (at the data-link level) some useful (ack) information onto the delivered packets which is used for efficient message ordering at higher levels; we differ in the type of low-level service we chose to provide. The Reliable Multicast Transport Service of [16] provides a service similar to ser2 which helps building a membership service at the higher level; but the emphasis there is to transparently extend unicast TCP into multicast TCP. Recall that when a receiver receives no transmissions from the sender, our protocol causes the receiver to stop unicasting its responses to the sender. This feature not only helps save bandwidth but also allows the protocol to be easily extended to deal with the sender crash - an issue that is not addressed here. Other future work planned is three-fold: incorporating congestion control, extending the protocol for $n \rightarrow m$ context, and implementing the extended version as the underlying service for our group management system [6].

Acknowledgements

The authors thank the anonymous reviewers for their many useful comments. Thanks are also due to Richard Achmatowicz and Martin Beetz for their patience and technical support.

References


