Efficient TCP-like Multicast Support for Group Communication Systems

Marinho P. Barcellos
Hisham H. Muhammad

André Detsch
Guilherme B. Bedin

PIP/CA - Programa de Pós-Graduação em Computação Aplicada
C6 - Centro de Ciências Exatas e Tecnológicas
UNISINOS - Universidade do Vale do Rio dos Sinos
São Leopoldo/RS, CEP 93022-000 - Brazil
http://www.inf.unisinos.br/~marinho

Abstract

The availability of a "TCP-like multicast service" is a common assumption among group communication protocols. This assumption has been inefficiently satisfied through either multiple TCP connections or broadcasting. In this paper, we specify the requirements for such TCP-like multicast service, and then present a scalable reliable multicast protocol, called PRMP, that satisfies these requirements. PRMP allows a group communication protocol to provide high-level services efficiently and scalably for fault tolerant applications. We describe the architecture used to implement PRMP in Java, and how this implementation was used to perform a set of practical experiments simulating group communication. Experimental results show that PRMP outperforms two other alternatives, while still presenting less network cost.

keywords: reliable multicast, group communication, TCP.

1 Introduction

Building group communication systems capable of tolerating site crashes and network partitions has been under investigation for several years. Useful, programming paradigms such as virtual synchrony (VS) have been specified ([1]). An efficient implementation of these abstractions can be obtained if an underlying one-to-many "TCP-like multicast service"[1] is available. Historically, such service has been implemented through multiple TCP streams (e.g., Newtop [8], [15] and Phoenix [9]). Some other systems, like Transis ([7], [5]), are based on unreliable link-layer multicast/broadcast. Neither approach scales well, wasting network

[1]although we use the term "service", one could employ "layer" or "abstraction" as well.

bandwidth and host resources, since broadcasting is limited to LANs, and multiple TCP connections require many copies of the same packet to be redundantly transmitted over the same communication channels. For communication over the Internet, this is unaffordable.

Group communication in the Internet is only feasible through IP multicast (see [6]). IP multicast is capable of efficient and scalable packet delivery: packets are routed through multiple networks according to a multicast tree, built using a multicast routing protocol (e.g., DVMRP or PIM). A packet only traverses an edge once. Using IP multicast, a peer node may send packets to a given network address (IP class D), without having to (or being able to) determine the exact membership of the set. Several group communication protocols, such as Horus ([17]), claim to be able to take advantage of IP multicast, even though scalability issues such as feedback implosion are not addressed ([4]).

In this paper, we describe a "scalable reliable multicast protocol" called Polling-based Reliable Multicast Protocol (PRMP), which efficiently implements the TCP-like multicast service required by group communication systems. Note that our purpose is *not* to propose a new *group communication protocol*, which is more efficient, more scalable, or both, but instead to support other group communication systems by providing the often required TCP-like multicast service they assume to exist.

The rest of this paper is organized as follows. In Section 2, we introduce the concept of the TCP-like multicast service by defining the set of requirements. In Section 3 we discuss the PRMP controls and how they are used to satisfy the requirements. Experimental evaluation results are given in Section 4, which are followed by concluding remarks in Section 5.

2 TCP-like Multicast

As already mentioned, the feasibility of group communication protocols in the Internet depends on the use of IP multicast, for IP multicast is required to efficiently distribute packets along a tree of routers and receivers. Unfortunately, IP multicast may silently drop packets along the network. Further, packets may be duplicated, arrive of out order, or else suffer extreme delays. A packet drop may result in many receivers experiencing a given loss (see [21] for a study on spatial and temporal loss correlation in the Mbone). To deal with such events, basic Error Control techniques may be used. The simplest approach to loss recovery is a *multicast stop-and-wait* scheme, which works as follows. The *sender* sends a packet using IP multicast; upon receipt, each *receiver* unicasts a positive acknowledgment (ACK) to the sender. The sender waits on a timer, expecting one ACK from each of the receivers (as long as the sender knows who are the receivers). If all ACKs arrive within expected time, the sender can transmit a new data packet. Otherwise, the timer expires and the sender re-transmits the packet via multicast. This protocol is inefficient because it transmits a single packet per round-trip-time (RTT). A multicast sliding window ([2]) can be added to such protocol, allowing the sender to transmit several packets before waiting for any ACKs. A protocol where the sender sets a timer and expects ACKs from receivers is called *sender-initiated* ([20]). One such example is the Single Connection Emulation (SCE), an attempt to hide multicast behind TCP interface [19]. Sender-initiated protocols do not scale well because of the well-known problem of feedback implosion. Later in the paper, we show in practice how badly this kind of protocol performs (see "FF" in Section 5).

Newer protocols have been devised with scalability in mind, reducing the amount of feedback required. Several Implosion Control schemes have been introduced (see [16] for some examples). In general, these newer protocols attempt to make throughput, cost and (sender) state independent of group size. Because error recovery responsibility lies with the receivers, they are often called *receiver-initiated* ([20]). The design of the protocol trades off *reliability* (and performance) for *scalability.* There is no mechanism to keep track of membership: the sender remains *unaware* of how many receivers exist and their identity. Receivers join the group by subscribing to the proper IP multicast address. Network topology is harnessed through hierarchy: receivers are logically organized so that some act as re-transmitters for nearby receivers (local recovery). This model is highly scalable, and has been successfully used for one-to-many bulk dissemination of data and "semi-reliable" transmission of multimedia contents. However, because no state about receivers is maintained at the sender, receiver-initiated protocols cannot provide "end-to-end reliability" that is expected from the TCP-like multicast service required by group communication protocols. Further, [12] shows that these protocols require infinite buffers in order to prevent deadlocks.

Below, we identify a set of requirements for the TCP-like multicast service, based on what TCP connections offer and what group protocols assume to be available. The requirements are:

(r1) the protocol must control the level of feedback sent by receivers so to avoid feedback implosion;

(r2) end-to-end "reliable" transmission of a datagrams (or bytes of a stream) to multiple receivers, detecting packet losses and recovering from them in order to mask omissions, duplications and re-orderings;

(r3) initial multicast group setup and control, detecting host crashes and persistent network partitions and mapping them as "connection breaks";

(r4) manage *finite* buffers at sender and receivers, also preventing unnecessary packet losses due to receiver overrun; and in the case communication takes place over Wide Area Network (WAN) links,

(r5) help alleviating congestion in bottleneck routers within the network in a way that is "TCP-friendly").

The above requirements[2] make a difficult task to provide a TCP-like multicast service. IP multicast was introduced around 1990 ([6]), but still nowadays all of the above requirements for reliable multicast are yet to be satisfied in the same protocol. This is mainly because most research on *scalable* reliable multicast has attempted to come up with protocols that will scale to *very large* number of receivers, and whose performance and cost are independent of group size. To achieve this, the sender cannot control membership. So, most research on the field of scalable reliable multicast protocols have adopted receiver-initiated models, for the sake of improved scalability. Below we discuss why such protocols fail to satisfy the above requirements.

[2]Multimedia applications exhibit a requirement which is *not* covered by TCP: soft real-time delivery.

First, receiver-initiated protocols tend to satisfy (r1), as they avoid implosion through NACK-based Error Control, Forward Error Correction and/or hierarchy. However, as shown in [12], they can only satisfy (r2) as long as that they have *infinite* buffers and session time. They do not satisfy (r3), because even though some protocols might count with a "loose" Session Control (e.g., SRM [10]), the (unreliable) group membership must remain hidden at the IP multicast layer. They also do not satisfy (r4), since the sender cannot *safely predict* buffer availability at receivers (Flow Control in this case fluctuates the sending rate to attempt to reduce overrun losses). Finally, it is hard to satisfy properly requirement (r5), because TCP Congestion Control is sender-initiated (and since this is fundamental for Internet stability, new schemes for congestion control for reliable multicast are being intensely investigated).

We conclude defining a TCP-like multicast service as a one-to-many protocol that would fulfill all the requirements above.

3 PRMP Protocol Controls

PRMP has been described elsewhere ([2], [3], [13]); in this section, we provide an overall view and elaborate on how PRMP meets the five requirements through its corresponding controls.

Data is placed in data packets and transmitted via IP multicast to receivers. Sender and receivers keep a sliding window, called *send* and *receive window,* respectively; the sender marks in the send window which data packets have been positively ACKed and by which receivers, according to the feedback (response packets) it receives. To avoid feedback implosion, the sender uses polling to control the amount of feedback generated by receivers: only when prompted by a *polling request*, a receiver can unicast a *response* containing its status (ACKing and NACKing several data packets at once). When a response arrives at the sender, the send window is updated, and loss detection and recovery takes place. The sender detects data packet losses through NACKs contained in the responses sent by receivers after poll requests. Recovery is achieved by means of retransmissions, which may be either by (multiple) unicast operations or a single multicast, depending on the number of copies to be retransmitted. Poll requests and responses may be lost too, and their loss is detected by the sender through timeouts. If so, the sender re-sends a polling request and sets a new, enlarged timer to wait for a response; the process is repeated until either a response is received or the sender gives up on the receiver and removes it from the session. The receive window slides forward according to the receipt of data packets, and their (in FIFO order) consumption by an *upper layer* (application or group communication system). The send window slides forward according to responses from receivers, allowing the transmission of new data. This mechanism dictates that a new data packet can only be transmitted if the sender can guarantee that the packet can be safely received and stored in receiver's buffers. Below, we briefly describe each of the control mechanisms.

3.1 Implosion Control

The Implosion Control mechanism of PRMP is based on a poll-*planning* scheme: the sender plans when responses should arrive so that they are *uniformly* distributed in time, and do not exceed a given rate. To control the arrival time of responses, the sender uses the RTT up to receivers and adjusts the time in which requests are sent (delay transmissions).

Time ahead is divided in epochs, periods of equal length. The purpose of the poll planning mechanism is to schedule a maximum number responses per epoch. To keep track of scheduled responses, it employs a vector, whose entries count the number of responses expected per epoch. Whenever an epoch is full, the mechanism finds the next "available" epoch in the future able to accept a new response. When so, the transmission of the request is delayed accordingly.

To save bandwidth, the mechanism is designed so that, if possible, a poll request is sent piggybacked onto a data packet. Further, poll requests are sent *on demand:* when there is no data to be ACKed, or Session Control to be performed, the sender does not send poll requests unnecessarily.

3.2 Session Control

Session Control has typically three phases: *connection establishment*, *connection management* and *connection tear-down*. In PRMP, the *connection establishment* is responsible for establishing contact between sender and receivers. Like TCP, PRMP uses the *three-way handshake scheme* ([18]). PRMP has two connection establishment models: *invitation* and *announcement*. In the former, the sender takes a list of receivers (IP addresses and ports to contact) and spawns a given number of threads, which continuously employ the three-way handshake until all receivers on the list have been contacted or given up. The number of concurrent threads represent a tradeoff between efficiency and scalability. In the *announcement model*, for a given time length, the sender keeps transmitting periodical announcement messages to a well-known IP multicast group. It waits and collects join requests from receivers, and for each request received, it spawns a thread that handles the three-way handshake protocol entirely.

Having the connection been successfully established, *connection maintenance* (the phase of actual data transmission) begins. Like with TCP Session Control, suspicions of host crashes or network disconnections are based on continuous exchange of polling requests and responses. After sending a packet, the sender waits on a timer which must be sufficiently large to allow a receiver to receive a packet, send an ACK, and such an ACK arrive at the sender. When there is a timeout, the sender assumes that the packet or its corresponding ACK has been lost, and tries again. After a given number of consecutive retries, the sender *suspects* the receiver, the host or the network that connects to it has failed, and considers the connection to the receiver to be *broken* (the receiver is then removed from the group, and this condition is reported to the upper layer).

Since nothing is assumed of the data generated by the application, there can be long periods of inactivity. To keep the connection alive, the sender periodically sends a request to elicit a response from a receiver. A suspicion cannot always be correct, as network loads and hence

RTTs cannot be always accurately predicted. So, false suspicions are possible and are even acted upon as the only way to ensure liveness in applications. However, at the end of a multicast session, the sender can guarantee that the set of receivers that have remained in the session have received all data transmitted.

A receiver may also leave a session spontaneously. The sending upper layer will be informed through an exception, but the session goes on if there exists at least one receiver. In contrast, new receivers are *not* admitted in an ongoing session. To implement a join membership change during the multicast session, the group communication system should open a new session and close the ongoing one. This will force a synchronization.

A *connection tear-down* can only be initiated by the sender. Receivers are polled once more, to confirm the receipt of all pending data. This is done scalably by the usual planning of poll request and responses.

3.3 Error Control

As already mentioned, packet loss detection and recovery is based on an efficient multicast sliding window scheme which allows multiple data packets to be outstanding in the network (see [3] for details). Sender and receivers negotiate the window size at connection establishment. Each entry of the window corresponds to a fixed-size data unit. Each data unit is uniquely identified by a sequence number and transported in data packets. These may drop or reordered by the network. The receiver suspects it missed a packet when it finds a gap in the packet sequence. If polled, a receiver sends a response which will reproduce the gap to the sender (this corresponds to a NACK).

Reliable multicast cannot scale if all losses result in multicast retransmissions. So, a protocol must balance when to retransmit with multicast and when with unicast. In PRMP, losses of data packets reported are treated together. NACKs are collected before a decision is made regarding the way the retransmission is done. The sender decides to retransmit whenever it has collected sufficient NACKs to justify a multicast retransmission, or else it has collected all NACKs and ACKs regarding the packet and there is no justification for multicast (it sends via multiple unicasts).

After sending a poll request, then sender waits on a timeout for receiver responses. This timeout period, called RTO (retransmission timeout) must be long enough to allow all receivers responses to be received and treated. The RTO is fixed based on the sender's estimate of RTTs so that premature retransmissions are avoided.

3.4 Flow Control

PRMP prevents unnecessary losses due to receiver overrun. Packets are delivered in FIFO order to the upper layer. When packets arrive at a receiver, they are placed in a buffer as long as there is space. As in TCP, even though data may be ready for consumption, the upper layer may not be. Therefore, unlike protocols for bulk data transmission, a TCP-like multicast service *must cope* with the situation where the upper layer blocks for arbitrarily long, and packets "clog up" in the receive buffer. In receiver-initiated protocols, the sender may transmit packets without feedback from receivers, and a large number of packets might be discarded.

PRMP is conservative, since the sender only sends new packets when it can guarantee that all receivers have enough space to safely store the packet. Not a single packet is discarded due to overrun; if the upper layer blocks or is slow, the buffer will eventually fill, and the space reported (through responses) to the sender will prevent it from transmitting new packets.

3.5 Congestion Control

PRMP performs congestion control almost the same way as TCP. For this reason, we describe briefly Congestion Control in TCP. TCP uses a congestion window which restricts the transmission of new data packets by reducing the (value of) "available window" to no more than *cw* packets. The value of *cw* is additively incremented when packets are positively ACKed and multiplicatively decremented when losses are detected. In TCP, this corresponds to a retransmission timeout event. At the start of the session, the value of *cw* is set to 1 and increased exponentially every RTT until a loss is detected; this is called *slow start* (see [18] for details).

To implement additive increase, the sender must enlarge the window in 1 packet for each full window successfully sent. This way the sender slowly probes for additional capacity, until the maximum window size is encountered, or a loss is detected (there is a retransmission timeout). If there is a loss, *cw* is reset to 1 and the protocol enters slow start, during which the value of *cw* is incremented in 1 at each ACK received, making the size of *cw* double at each RTT. Slow start stops when *cw* reaches half its value when the loss occurred. Alternatively, if a technique called *fast recovery* is used, there is no slow start phase after losses, and for each timeout, *cw* is reduced in half.

In PRMP, the congestion window is one for all send windows. The value of *cw* is increased when a data packet gets *fully ACKed*. When a packet gets NACKed for the first time, it is equivalent to a TCP timeout: the value of *cw* is set to 1, and slow start begins.

3.6 Requirements

Having described PRMP controls, now we show how PRMP satisfies all TCP-like multicast service requirements:

(r1) it effectively reduces feedback and avoids implosion, which can be seen on the simulations presented in [2] and the practical results shown later in Section 5;

(r2) it detects data packet losses through responses, and losses of poll requests and responses through timeouts, being both are recovered through retransmissions;

(r3) it has a clever Session Control mechanism which allows the sender to (scalably) keep track of the membership similarly to TCP;

(r4) it fully prevents overrun losses since the sender has enough information to carefully decide when to send new data; and

(r5) it achieves TCP-friendly multicast congestion control as it mimics the congestion window of TCP by monitoring full ACKs and first NACKs.

4 Protocol Architecture and Implementation

4.1 Why Java

The Java platform has proven to be to be a suitable environment for the development of PRMP. At first, the main factor leading against its choice was the amount of system resources required by the JVM (Java Virtual Machine), in terms of processing and memory consumption, although this tends to be less of an issue as hardware evolves ([11]). On the other hand, the Java language, API (Application Programming Interface) and virtual machine provides a consistent environment for the implementation of a protocol like PRMP. Its object-oriented approach suited the modeling of the protocol very well. Further, the binary-level portability of Java code is a great advantage when it comes to heterogeneous network environments, such as the Internet.

In Java, the sockets API is available through a set of classes[3] that are distributed with the Language implementation. Actually, Java eases the task of network programming by extending basis socket classes, providing useful abstractions. Two such examples are TCP input/output streams and object *serialization* (also known as marshaling). In particular, PRMP employs `DatagramSocket` and `DatagramPacket` classes, which allow the creation/transmission/reception of datagram packets in a well structured and simple manner.

PRMP packets can carry relatively complex objects, like the sliding window contained in response packets. As datagram packets can only transport raw bytes, the Java-provided `Serializable`[4] interface is used to marshal and unmarshall objects. An object of a class that implements this interface can be converted to a byte array that will contain all information about the internal state of the object. The array can be used to reassemble (unmarshall) the object, recovering the original data. The main disadvantage of this scheme is the added overheads: the processing overhead, due to the time spent serializing and de-serializing objects, and the bandwidth overhead, due to the extra space in packets that is required to store the structure and identification of the class of the object being transmitted.

Due to its sophisticated Controls, the PRMP sender is somewhat complex. The complexity is reduced through a multi-threading architecture: concurrent threads interact through queues and tables, making use of monitors to avoid race-conditions. Mutual exclusion is implemented through the `synchronized` modifier. Java has support for concurrent programming through a set of classes to allow the creation/monitoring/elimination of threads, as well as monitors.

4.2 Architecture

The internal protocol structure is completely based on object orientation and concurrent programming. The complexity is broken by using one different synchronous thread to perform

[3]http://java.sun.com/j2se/1.3/docs/api/java/net/package-frame.html
[4]http://java.sun.com/j2se/1.3/docs/api/java/io/Serializable.html

each of the main jobs. We identify four main jobs in the sender, namely: (i) transmission of data and poll packets (including the planning of polls); (ii) reception and handling of feedback packets (and the associated loss detection and recovery); (iii) handling of asynchronous events (setting timers, canceling timers and treating timeouts); and (iv) interfacing with the upper-layer, including separation of data into blocks ready for transmission. Hence, there are four threads.

Figure 1 illustrates the overall architecture. The Generator Module (GM) interfaces with the upper layer. The upper layer writes data to GM, which "generates" and queues up data packets for transmission in the Transmission Queue (TxQ). The Transmitter Module (TxM) reads from TxQ, performs the (re)transmissions of data and poll packets, programming timeouts whenever poll requests are sent. To program a retransmission timeout, the TxM enqueues a new event in the Timeout Queue (ToQ). The Event Module (EM) reads from ToQ and is responsible for timeout handling; when so, EM enqueues re-polling requests in TxQ for TxM to handle. The Response Handler Module (RHM) receives response packets sent by receivers, canceling pending timeout events in ToQ when all responses expected from a poll request have been received. Also, when a response is received from a given receiver, RHM updates the estimate of RTT for that receiver and its corresponding send window. When updating the send window, RHM may start error recovery: it detects a loss, collects ACKs and NACKs, and when it eventually decides to retransmit a packet, it does so enqueing the packet in TxQ. When all receivers have positively ACKed and delivered the packet, RHM may slide forward the window, enabling GM to take more data from the upper-layer (end-to-end Flow Control).

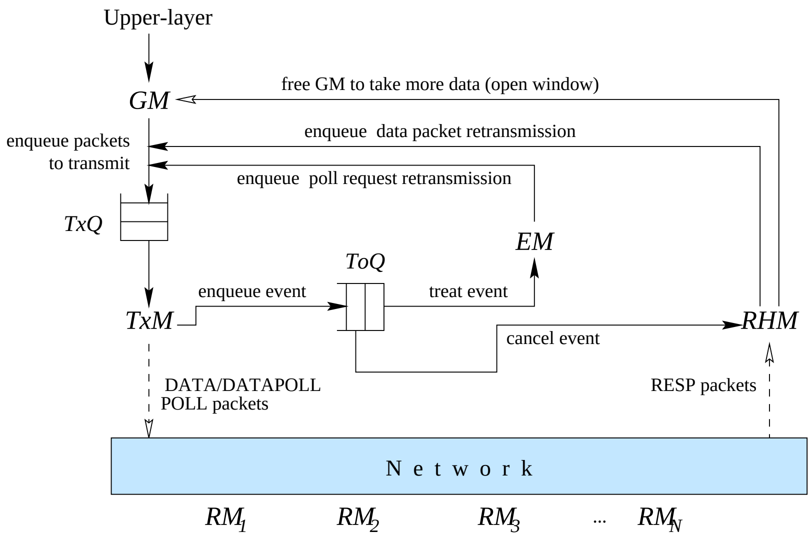
Upper-layer
GM
free GM to take more data (open window)
enqueue packets to transmit
enqueue data packet retransmission
enqueue poll request retransmission
TxQ
EM
ToQ
enqueue event
treat event
TxM
RHM
cancel event
DATA/DATAPOLL
POLL packets
RESP packets
N e t w o r k
RM1
RM2
RM3
...
RMN

Figure 1: Protocol internal architecture.

5 Experimental Results

This section presents the results obtained from a set of practical experiments we have conducted with PRMP and two other protocols: *multiple TCP streams* and the *Full Feedback protocol*. As the name indicates, the multiple TCP streams scheme emulates multicast through $N-1$ TCP streams. Because TCP has built-in Error, Flow, Congestion and Session Controls ([18]), the sender may simply send (write to a stream) and forget. A "multicast packet" is sent by writing the required number of bytes $N-1$ times, one for each stream. The second alternative is the Full Feedback protocol (see [2]), or simply "FF". All data packets are sent via IP multicast. FF is a reliable multicast protocol with sliding window Error and Flow Controls, in which receivers send ACKs for each packet received (this protocol is a similar, but slightly wiser, version of the A protocol found in [20]).

5.1 Experiment Settings

The frequency and size of messages generated by the group communication system will depend on the particular application. Therefore, one should not make assumptions about transmission bursts, average message size and so on. In our tests, we chose to emulate a group communication system for a *symmetric*, *closed* group model of N nodes that behave synchronously (see below). Each "application node" transmits and receives messages to the other $N-1$ nodes and is implemented through a multi-threaded Java program comprised of 1 sender and $N-1$ local receivers.

The group computation advances in "rounds", similarly to the synchronous model described by Lynch ([14]): a round consists of sending a message to neighbors (in this case, everyone), receiving one message from each neighbor (everyone), and then processing messages to make a state transition. These messages are of the type DATA and have size S bytes. A node sends a DATA message to the group, waits until $N-1$ DATA messages are received from its peers, and then spends a random time (uniform distribution between 0 and Tp ms) while "processing" the messages. Each node performs M of the above rounds, and then gracefully closes its session.

An external "coordinator" controls the experiment and collects output information. The experiments starts with each node establishing a "multicast session" with the other $N-1$ members, resulting in a full mesh[5]. After establishing a session, a node sends a READY message to the coordinator; when the coordinator has collected all READY messages, it sends a START message to all nodes, which then start the first round.

In the tests, we fixed the amount of data to be transmitted by each node in $1,000,000$ bytes. We performed two sets of experiments, one using a message size S of 1000 bytes (thus M was 1000 rounds), and another using a message size S of 4000 bytes (thus M was 250 rounds). Tp was set to 100 ms (mean 50 ms). This means that each group node will send to the group either 1000 or 250 messages of 1000 or 4000 bytes each, respectively, and that at end of the session a node will have received $1,000,000 \times (N-1)$ bytes of data. Messages of size S will be transported by PRMP and FF in packets of 1000 bytes; therefore,

[5]with multiple TCP stream protocol, connections are bi-directional and so $N(N-1)/2$ streams are required.

a message may trigger the sending of 1 or 4 data packets, plus at least one feedback packet. (In multi-TCP, the packet size will vary.) Each protocol will therefore generate a flow of feedback packets. In PRMP such flow was restricted by setting the parameter Response Rate (RR) to 20 responses/sec. That is, each sender (at each node) will control feedback such that no more than 20 response packets are received per second. However, a node will also receive messages (i.e., data packets) from $N-1$ nodes. FF and multi-TCP do not control the amount of feedback.

The experiments were run in a controlled environment. Although the configuration was a Local Area Network comprised of around 70 PCs, in the tests we employed a subset of 10 machines equally-equipped: PC Pentium III-450MHz with 64 MB of RAM, and Fast Ethernet boards used however in ordinary Ethernet mode (10 Mbps). The operating system was GNU/Linux v.2.2.14, and the development/runtime environment was Sun JDK v.1.3.

In order to evaluate PRMP and compare it with FF and multi-TCP, we chose two metrics: T, defined as the "total time", and C, the "total network cost" in bytes. Below, these metrics are explained along with the experiment and result description.

5.2 Total Time

The total time Ti at each node i is measured as the time between the delivery (to the upper layer) of the START message and the last of all $N-1$ messages of the round M. The overall, total time T is the largest Ti recorded among all nodes, that is, maximum Ti. Figure 2 presents the total time T (in seconds) in function of N, for $S = 1000$ (left) and $S = 4000$ (right). In both graphs, we show a curve that corresponds to the minimum time T, a lower bound on T calculated as the sum of all processing times Tp. Since average Tp was set to 50 ms, T could never be less than 50 seconds for $M = 1000$ rounds and 12.5 seconds for $M = 250$ rounds. In other words, the best result for T would only be achievable with *instantaneous* copy of messages among remote nodes.

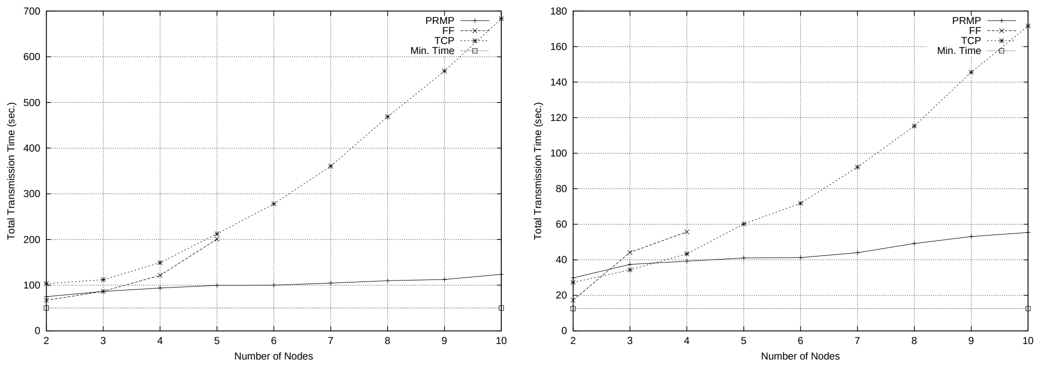

Figure 2: Total time T in seconds in function of N, for $N = 2$ to 10, with $S = 1000$ bytes (left) and $S = 4000$ bytes (right).

First compare the scale on the two graphs of Figure 2. Note that although the results are consistent between the two, the performance on the right ($S = 4000$) is much higher than on the left, which occurs because group synchronization on the right takes place 4 times less frequently. In other words, if $S = 4000$, a node can send 4 packets of 1000 bytes without having to wait for the next round[6].

Note also that, for any protocol, T is expected to grow with N. There are reasons for this: the volume of data being transferred grows exponentially with N, and so does the combined probability of a receiver experiencing a packet loss in any single multicast transmission. Additionally, since in each round a node will receive $N-1$ messages, more and more messages need to be delivered to the upper layer.

As it can be seen in Figure 2, the multi-TCP scheme presented the worst performance of all: its time T grows exponentially with N. This is mainly due to the fact that the multi-TCP approach requires each message to be transmitted $N-1$ times over the network, with two implications. First, the sending of packets takes time, in particular larger data packets. Second, the amount of traffic grows heavily and will cause losses, which on their turn will require loss detection and recovery. For TCP, Congestion Control will also affect performance for $S = 4000$, as the congestion window will be reduced to 1 segment after each loss; this prevents a network collapse. Like TCP, the time required for FF also grows exponentially. However, unlike TCP, FF will try to recover from losses by re-multicasting packets, which is equivalent to trying to "put off a fire with gasoline". That is why the FF curve in the graphs are incomplete: all experiments for $N > 5$ with $S = 1000$ or $N > 4$ with $S = 4000$ led to network collapse, being aborted after several minutes of no progress being recorded at all. In contrast to multi-TCP and FF, the time T of PRMP grows slowly with N, from 80 to 120 sec as N increases 5 times. These results demonstrate that PRMP remains efficient as N grows, in particular if one considers the increasing host load and combined probability of loss that a protocol must cope with.

5.3 Network Cost (bandwidth)

The cost Ci is the total number of bytes passed by the protocol to UDP for transmission (via multicast or unicast), either data or feedback packets, and considers all the overhead included by Java object serialization. Note that Ci intentionally benefits multicast: the number of bytes in a multicast transmission is the same of unicast. The overall C, presented in the graphs, is the sum of all Ci's.

Figure 3 shows in log scale the network cost N measured in Mbytes, for all three protocols. It also shows, for the sake of comparison, an optimal case ("optimum"). The optimum curve is a lower bound on the bandwidth consumption: it considers *only* the data transmitted by the upper layer, without *any* protocol overhead such as headers and feedback packets. The Ci for a given node i cannot be less than the amount to be transmitted by each node: $1,000,000$ bytes (nearly 1 Mbyte). The overall C for the optimum curve is therefore $N \times 1,000,000$ bytes, or 9.5 Mbytes for $N = 10$.

The network cost shown for multi-TCP in Figure 3 is also an optimistic value. Since the cost C for the multi-TCP protocol could not be measured without using kernel-level tools, C for multi-TCP represents a *lower bound*, adding only an overhead of 20 bytes (TCP header) for each message transmitted to a node (this number would be greater for $S = 4000$ bytes, where more than 1 packet would be necessary to transport each of the 250 messages).

[6]subject to the sliding window of the protocol underneath.

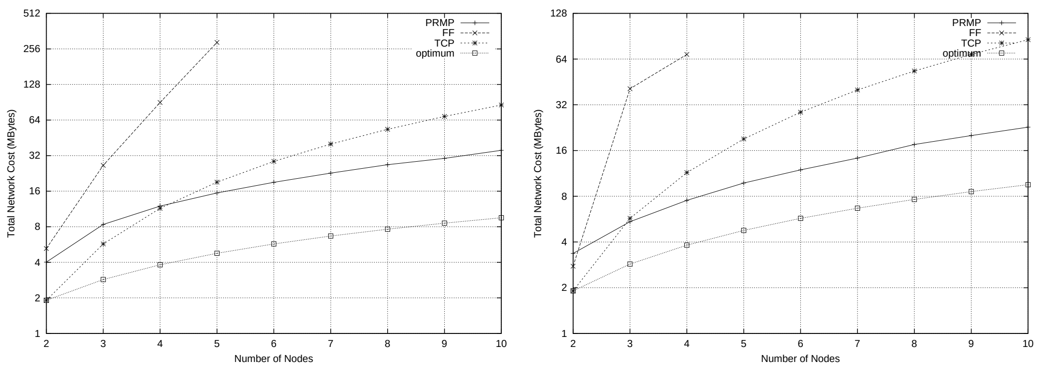

Figure 3: Network cost C in Mbytes in function of N, for $N = 2$ to 10, for $S = 1000$ bytes (left) and $S = 4000$ bytes (right).

First note that, as for time T, the cost C presented on the graph on the left ($S = 1000$) are higher than those on the right ($S = 4000$). This is because 4 times more feedback packets will be required when $S = 1000$ (except for TCP, which may piggyback ACKs). Note also that the Y axis is in log scale.

Second, note that the unscalable nature of FF, without Implosion Control, appears clearly in the results; in Figure 3 (left), to transfer approximately N Mbytes, FF cost grows aggressively from 5 to 300 Mbytes when group size grows from 2 to 5. The multi-TCP approach is more economical than PRMP for $N = 2$, and similar for $N = 3$ or 4. This is because the multi-TCP approach was designed to take advantage of the fact that TCP streams are bi-directional: with 2 nodes only, there is a single connection and no overhead (in fact, it is TCP traditional point-to-point communication!) For larger groups, multi-TCP demands on network traffic grow acutely, which is explained by the fact that each node must send the same data $N - 1$ times. PRMP, instead, presents a network cost that remains consistently higher than the optimum, but proves to be substantially better than the other alternatives.

6 Concluding Remarks

The main contributions of this paper are (i) to specify which are the requirements for a TCP-multicast service, an assumption of some group communication systems; (ii) to show how our protocol, PRMP (presented in [2]) satisfies all these requirements; (iii) to provide an architecture and implementation for PRMP in Java; and (iv) to show practical results taken from an experimental evaluation with the implementation.

Unlike other "scalable reliable multicast protocols", PRMP does not trade off reliability, performance or network cost for unlimited scalability. The TCP-multicast service is the layer over which group communication systems can be built. Multiple TCP connections are inefficient and restricted to a few receivers; broadcast is wasteful and does not scale.

Other scalable reliable multicast protocols are not suitable for group communication support because they emphasize scalability to the extreme, negatively affecting other aspects, and typically assume a bulk data dissemination. PRMP is the first protocol to attack all TCP-like multicast requirements together. Unlike other scalable reliable multicast protocols, in PRMP the sender maintains the membership information and uses this information for the benefit of other mechanisms. A multicast protocol cannot provide "end-to-end reliability" unless a sender knows and controls the membership set it is transmitting to. Indeed, PRMP benefits from keeping receiver's state in several ways: (a) it can avoid implosion, by planning arrival of feedback packets; (b) it can efficiently detect and recover from packet losses; (c) it uses up-to-date RTT estimates to make fewer mistakes while suspecting receiver failures; (d) it does not make assumptions regarding the traffic generated by the application; (e) it does not require infinite buffers to deliver all data to all receivers in the session; (f) it prevents unnecessary losses by overrun; (g) it uses a congestion window to perform TCP-friendly Congestion Control.

To illustrate our point, we have described a PRMP implementation in Java and discussed its performance. The results indicate that PRMP presents much lower network cost and better throughput than the most commonly used alternatives. In absolute terms, we expect the performance results for PRMP presented here may be increased by a factor of 10 if PRMP is re-implemented in C or C++ ([11]).

In this paper, we were able to execute experiments in a single LAN. Performing practical experiments with multicast applications using several networks has been very difficult for many reasons. First, it requires many other people/institutions to collaborate in the test. Second, a multicast transmission might easily swamp the network with excessive traffic. Third, multicast must be enabled in all networks involved, and specially in the links interconnecting them (like with IP multicast tunnels). Finally, experiments should be conducted "out of hours" in order to reduce the likelihood of extraneous traffic.

Nonetheless, as future work, we plan to extend our experiments to include several networks. Even though it has been already shown that PRMP works well for such configurations (see [3]), and that it can deal properly with heterogeneous RTTs, we would like to present an experimental evaluation to validate the use of PRMP in supporting group communication systems over sparsely distributed, large groups.

References

[1] O. Babaoglu, A. Bartoli and G. Dini, "Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems", IEEE Transactions on Computers, v.46, n.6, June 1997, pp. 642-658.

[2] M. Barcellos and P. D. Ezhilchelvan, "An End-to-End Reliable Multicast Protocol Using Polling for Scalability", In IEEE INFOCOM'98, San Francisco, April 98, pp.1180-1187.

[3] M. Barcellos, "PRMP: Poll-based Scaleable Reliable Multicast Protocol", Ph.D. Thesis, University of Newcastle, Newcastle upon Tyne, Oct. 1998, 200p.

[4] K. Birman, "Building Secure and Reliable Network Applications", Manning: Prentice Hall, 1996. 500p.

[5] R. Budhia, "Performance Engineering of Group Communication Protocols", Ph.D. Dissertation, University of California at Santa Barbara, Eletrical and Comp. Eng., Aug. 1997, 169p.

[6] S. Deering and D. Cheriton, “Multicast Routing in Datagram Internetworks and Extended LANs”, ACM Transactions on Computer Systems, pp.85-111, May 1990.

[7] D. Dolev and D. Malki, “The Transis Approach to High Availability Cluster Communication”, Communications of the ACM, v.39, n.4, April 96, pp. 64-70.

[8] P. Ezhilchelvan, R. Macedo, and S. Shrivastava, “Newtop: A Fault-Tolerant Group Communication Protocol”. In IEEE 15th Intl. Conf. Distributed Computing Systems, pp.296-306, May 1995.

[9] P. Felber, R. Guerraoui and A. Schiper, “The Implementation of CORBA Object Service”, Theory and Practice of Object Systems, v.4, n.2, 1998, pp. 93-105.

[10] S. Floyd, V. Jacobson, S. McCanne, C. Liu and L. Zhang, “A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing”, IEEE/ACM Transactions on Networking, v.5, n.6, Dec. 1997, pp. 784-803.

[11] B. Krupczak, K. Calvert, M. Ammar, "Implementing Protocols in Java: The Price of Portability", In IEEE INFOCOM'98, San Francisco, April 98.

[12] B. Levine, J.J. Garcia-Luna-Aceves, “A comparison of known classes of reliable multicast protocols”, In IEEE International Conference on Network Protocols, 1996, pp112-121.

[13] C. Liu, P. Ezhilchelvan, and M. Barcellos, “A Multicast Transport Protocol for Reliable Group Applications”, Lecture Notes in Computer Science, 1736. (First International Workshop on Networked Group Communication - NGC'99). Springer-Verlag, 1999, pp.170-187.

[14] N. Lynch, “Distributed Algorithms”, Morgan Kaufmann, San Francisco, 1996, 872p.

[15] G. Morgan, S. K. Shrivastava, P. D. Ezhilchelvan and M. C. Little, “Design and Implementation of a CORBA Fault-Tolerant Group Service”, In 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Services, Helsinki, June 99.

[16] S. Paul, “Multicasting on the Internet and Its Applications”, Kluwer Academic Publishers, 421p., 1998.

[17] R. V. Renesse, K. P. Birman and S Maffeis, “HORUS: A flexible Group Communication System”, Comm. of the ACM, v.39, n.4, April 96, pp. 76-83.

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

[19] R. Talpade and M. H. Ammar, "Single Connection Emulation (SCE): An Architecture for Providing a Reliable Multicast Service", In 15th IEEE International Conference on Distributed Computing Systems (ICDCS95), Vancouver, Canada, June 95, pp. 144-152.

[20] D. Towsley, J. Kurose, and S. Pingali, “A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols”, IEEE Journal of Selected Areas in Communications, v.15, n.3, pp.398-406, 1997.

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