A Reliability Window for Flexible and Scalable Multicast Services

Dah Ming Chiu and Jaiwant Mulik

A Reliability Window for Flexible and Scalable Multicast Services

Dah Ming Chiu and Jaiwant Mulik

SMLI TR-2000-91 September 2000

Abstract:

This paper proposes a simple mechanism to allow trade-off between reliability with performance and scalability in a reliable multicast transport protocol. A reliability window is used in the data packet header to indicate which packets are useful to recover if lost. In addition, we describe a simple API that allows applications to set the reliability window in a way meaningful to the application. The usefulness of the trade-off is demonstrated using experimental results from a test network. We expect this mechanism to be useful for large-scale content distribution where the content ages over time.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
dahming.chiu@east.sun.com
jmulik@unix.temple.edu

© 2000 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>. The entire technical report collection is available online at http://www.sun.com/research.

A Reliability Window for Flexible and Scalable Multicast Services

Dah Ming Chiu, Jaiwant Mulik
Sun Microsystems Laboratories, Burlington, Massachusetts

1. Introduction

IP multicast is designed to be efficient and scalable, but only "best-effort" in delivering reliability. As we add reliability (at higher layers), we give up some level of efficiency, performance and scalability.

For example, one way to add reliability is to use forward error correction (FEC), which adds redundancy to data. The more redundancy added, the more reliability is provided. This creates a direct trade-off between reliability and efficiency. With increased redundancy, performance is also reduced proportionally. The scalability of IP multicast, however, is not affected, at least for those receivers which can receive enough packets to make sense of the data.

Another way to achieve reliability is to use group-based (or tree-based, as groups are hierarchically linked into a tree) repair service. In each subgroup, there is a repair service. One designated receiver (known as a repair head) monitors losses by the rest of the receivers and repairs them. The multicast session slows down to accommodate the most needy receiver. In this case, the reliability level is fixed (each receiver is ensured to receive all packets); there is marginal cost to efficiency (due to the amount of traffic to monitor the receivers' reliability needs); but the performance tends to decrease with scale (in terms of receiver population and reach). In order to avoid completely giving up performance, it is possible to set a minimum data rate. This minimum data rate is used as a criterion to remove those receivers that cannot reliably receive data for some period of time even under this minimum rate. This creates a binary form of reliability service - full reliability for those who can sustain the minimum data rate; and no reliability[1] for those who cannot sustain the minimum data rate.

To better accommodate different receive capabilities, another multicast solution is for the sender to divide up the content into different layers. The base layer contains the essence of the content, and each additional layer adds to the fidelity of the content. All receivers subscribe to the base layer; the subscription to additional layers is based on each receiver's receive capability. This solution, however, assumes that the content can be meaningfully divided into layers.

In this paper, we describe a very simple extension to the conventional tree-based repair approach described above. This simple extension allows the application to tell the transport not to repair packets that have "aged" sufficiently (as defined by the application). In times of congestion and severe packet losses, this mechanism helps the transport make the tradeoff between reliability and performance/scalability. When receivers have different receive capabilities, this service becomes similar to the layered multicast service. By not repairing some lost packets for the receivers with lower receive capabilities, the transport is essentially delivering different layers of information to different receivers depending upon their capabilities. In this case, the "layering" is stochastic, achieved adaptively, and different for each receiver, rather than predefined by the sender.

In the rest of the paper, we first describe our proposal in terms of a simple API, and how applications might use this API. Then we describe what changes we make to the tree-based reliable multicast protocols. Following that, we review some other proposals and protocols for supporting resilient data and compare them to our approach. Finally, we describe some

[1] An alternative model, to offer best-effort repair service for those receivers not in a repair group, is also possible.

experimental results to show the benefits of our approach.

2. A Simple Semi-reliable Multicast API and Its Applications

We begin our discussion by introducing a slightly modified socket API, and suggest how applications might use this API.

2.1 The ForgetBefore API

A traditional application programming interface for sending data on a multicast transport looks something like this (in pseudo code):

```
socket = open(address, port);
socket.send(data1);
socket.send(data2);
...
socket.send(dataN);
socket.close();
```

We propose a simple extention to this API, as follows:

```
socket = open(address, port);
fb = 1;
prev = socket.send(data1, fb);
fb = whattoforget(prev, fb);
prev = socket.send(data2, fb);
...
fb = whattoforget(prev, fb);
prev = socket.send(dataN, fb);
socket.close();
```

Each send() returns a number that is allocated by the transport, as a reference. The returned number is typically the sequence number used by the transport, starting from 1. The send() also includes a reference number with the following restriction:

1. The reference number in each send() must not be greater than the highest number returned from the transport so far. For the first send(), it is valid to use 1 as the reference number.
2. The reference numbers in successive send()'s must not be decreasing.

The reference number included in send() is used to tell the transport protocol to not bother with recovery of data units with numbers smaller than the given reference number. We call this parameter the *forgetbefore* number and this API the *forgetbefore API*.

In the above pseudo code, whattoforget() is an application-specific function that determines what is suitable to use as the next forgetbefore number, based on the forgetbefore number used thus far, and the previously sent packet number.

When the transport labels each send()'s data unit using consecutively increasing integers starting from 1, we call this *normal labeling*. With normal labeling, we use a shorthand to denote each send() as a tuple

(n, m)

where n is the labeling of the data sent, and m is the forgetbefore that comes with this data unit.

Semantically, the following two extreme cases represent the boundaries of semi-reliable service provided by the forgetbefore API:

1. (n,1), n=1,2,3,…
2. (n,n-1), n=1,2,3,…

The former represents the fully reliable service; and the latter represents the unreliable service.

At the receiver side, the socket API is extended to include the transport reference (sequence) number for each packet, as shown in the following pseudo code:

```
socket = open(address, port);
next = 1;
…
socket.receive(packet);
n = packet.getNum();
if (n – next > 0) {
  // something is missing
  …
}
data(n) = packet1.getData();
next = n+1;
...
socket.close();
```

2.2 How Applications Use the API

The forgetbefore API is particularly useful for applications that distribute content which ages quickly, especially when the new content overwrites the old content. The following are good examples of such content: weather and road traffic, stock quotes, and news.

Let us consider some examples of how an application uses the forgetbefore API.

2.2.1 Single Stream of Updating Data

Suppose an application sends a stream of normally labeled data packets, and each sequence of n packets are updated (obsolesced) by the next sequence of n packets. In other words, the packets

m*n+1, m*n+2, …, m*n+n

are overwritten by

(m+1)*n+1, (m+1)*n+2, …, (m+1)*n+n

for all m.

Using the forgetbefore API, these packets can be sent as:

(1,1), (2,1), ..., (n,1),
(n+1,1), (n+2, 1) ..., (2n-1,1), (2n,n+1),
(2n+1,n+1) ..., (3n-1,n+1), (3n,2n+1),
(3n+1,2n+1),...

In other words, we may tell the transport to forget about packets 1,2,…n at the time we complete sending (n+1),(n+2),…,2n, and so on. This definitely reduces the reliability service but, hopefully, it lets the transport focus its efforts in recovering and delivering more recent data than outdated data.

2.2.2 Multiplexing Streams

The application may send data that originate from multiple sources, with different update behaviors. In this case, the application needs to implement a simple multiplexing layer on top of the forgetbefore API.

Let two source packet streams X and Y be multiplexed and merged into a single packet stream Z.

A simplistic implementation of the multiplexer uses the same reference number space for X, Y and Z. In this case, it means, for example, Z = {1,2,3,4,5} and X = {1,3} and Y= {2,4,5}. For each source, the multiplexer keeps track of its last (which is also the highest) forgetbefore number. For our example, let X(k) and Y(k) denote respectively the highest forgetbefore number used thus far for stream X and Y at the time of transmitting packet k in stream Z. If packet (k+1) is from stream X, then X(k+1) would be the new forgetbefore number requested by source X, whereas Y(k+1) would remain the same as Y(k). Then the forgetbefore number to be used for packet (k+1) is simply

min(X(k+1), Y(k+1))

2.2.3 Time bound at the Sender

A third way to use the API is for the sender to suggest a reliability window based on a source time bound. Suppose the sender is configured with a content timeout value T. This means that after T seconds has passed, the sender can suggest in a subsequent packet to forget about recovering those packets sent more than T seconds earlier.

Let T(k) denote the time when packet k is sent, and

D(k, k-h) = T(k)-T(k-h)-T

Then the packet stream that implements a time bound T configured at the sender is the packet sequence

(k, k-h)

where each h minimizes D(k, k-h) over all non-negative D(k,k-h).

3. Implementation

In this section, we briefly describe how we implemented this feature in a tree-based reliable multicast protocol TRAM [TRAM]. The changes to the protocol are relatively minor:

- Change the socket send API to use the forgetbefore parameter, as described in the previous section.
- Add a field to the data message header to carry the forgetbefore number. Together with the data sequence number, this gives the reliability window.
- At the receiver side, add the current forgetbefore number as a state variable. When sending acknowledgements, treat any packet with sequence number older than the current forgetbefore number as received.
- For ordered delivery, incoming packets at the receiver are queued before being delivered to the application. Mark any missing packet older than the forgetbefore number as already delivered to the application.[2] When delivering packets to the application, include the transport sequence number.

There are two interesting design issues:

1. If a data packet whose sequence number is older than the current forgetbefore number (at the

[2] If such missing packets arrive before their turn for delivery to the application, they may still be inserted into the queue. This is discussed later.

receiver) arrives at the receiver, should it be delivered to the application or discarded?

2. Should a repair head discard packets older than the current forgetbefore number (at the repair head) from its retransmission buffer?

We decided in the default case to (1) deliver the late arrivals to the application, and (2) remove packets from the retransmission buffer only after all (child) receivers have indicated receipt of those packets. This decision reflects our philosophy of using the reliability window only to avoid unproductive repairs, rather than to enforce a truly time bounded delivery. While the latter might be a useful service sometimes, the former is usually more useful. It is always possible to also support the latter semantics via a global session configuration parameter. This decision means no other changes are necessary to the repair head and packet delivery modules.

4. Experiments

This section describes several experiments to help quantify the effect of using the reliability window to trade off reliability with performance, efficiency and scalability.

The experiments were designed to be simple, repeatable, and relatively independent of a particular reliable transport protocol. For simplicity and ease of configuration, we do not rely on a complicated WAN setup. Instead, all receivers are on the same LAN and are configured to emulate a specific *receive rate*. Let the multicast group have n receivers with receive rates $\{R_1, R_2, \ldots, R_n\}$ and slowest receive rate

$$R_{slowest} = \min_i (R_i)$$

The normal behavior of the transport is to multicast data at $R_{slowest}$. The sender can be configured with a maximum and minimum transmission rate, R_{max} and R_{min}, respectively. Those receivers who cannot sustain a data rate of R_{min} are removed (pruned) from the multicast group [PRUNING].

In all the following experiments, the delivery to the application is unordered. Since we are only monitoring the average throughput and number of receivers pruned, there is not much difference in ordered and unordered delivery, especially with the moderate maximum data rate we are using.

Our experiments first study how the use of a reliability window affects the throughput and pruning of a group of receivers with fixed receive rates. For simplicity, we use a *sliding reliability window*, which means the *forgetbefore* number is max((k-w),1) when the current data sequence number is k. The value w is a constant, and is referred to as the reliability window size (RWS). We expect the use of a smaller reliability window size to reduce the level of reliability, but also to reduce the amount of pruning and to increase the level of throughput.

We next study how the use of the reliability window can deliver different performance (throughput) and reliability to different receivers, providing an adaptive layered service.

We also experimented with a group of receivers with receive rates that vary over time. This effectively tries to characterize the effect on scalability, since as groups become large and sessions become long, the likelihood of receive rate variation over time goes up.

4.1 Receive Rate Emulation

To make a receiver receive at a configured rate, we added a *receive-rate emulator* at each receiver. The reliable multicast transport is implemented on top of UDP. The rate emulator (RE) is a module placed in between the transport and UDP as shown in Figure 1.

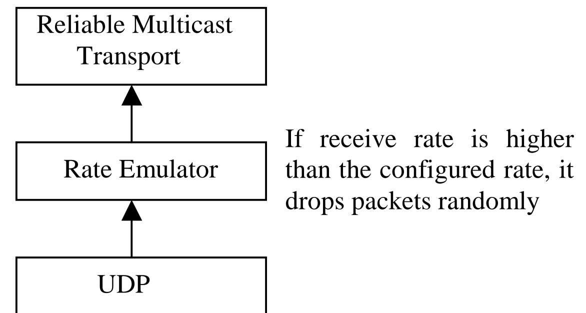

Figure 1. Rate Emulator

The rate emulator is configured with a *maximum receive rate*. As packets come in, it computes a current receive rate as a moving average. When the current rate is higher than the configured rate, it drops packets randomly (with probability *0.33*). This emulates a slow link on the path from the sender to the receiver, and packets get randomly dropped as the bottleneck bandwidth is reached.

The effectiveness of the rate emulator is first quantified by a calibration exercise. A single-receiver multicast group is created with a specific configured rate at the rate emulator. The following table

summarizes the measured average throughput[3] against the configured rate.

Configured Rate (B/sec)	Measured Rate		
	Average	Minimum	Maximum
100000	93683	93333	94382
150000	130116	125373	137704
200000	169143	168000	171428
250000	227137	221052	233333

Table 1. Calibration of receive rate emulation

Each measurement is repeated several times, with the average, minimum and maximum values noted. The measured throughput is consistently lower than the configured rate. This is expected due to the way we do the emulation described earlier. It is possible to tune the emulation algorithm so that the result is closer to the configured (desired) rate, but it is never going to be very exact. Figure 2 shows the dynamic behavior of the emulation of a constant receive rate. Microscopically, there is significant amount of oscillation, representing the cycle of catching up to the desired average rate and then adjusting back by introducing losses artificially.

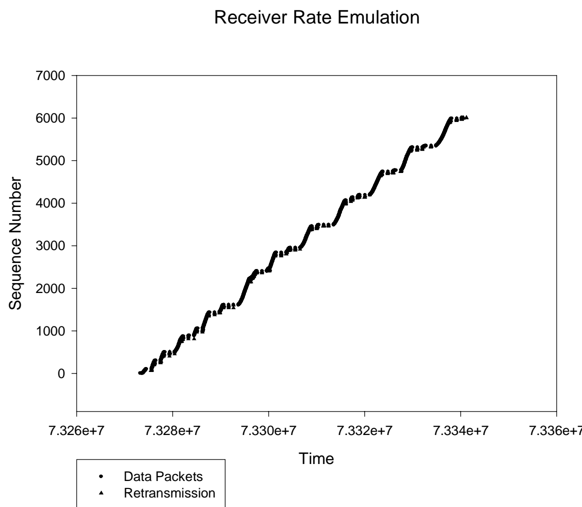

Figure 2. Dynamics of receive rate emulation

Each slow-down in the cycle is quite significant because the losses trigger repair activities as well as rate reductions or congestion window closings.

[3] The throughput can be measured either at the sender or at the receiver. For completely reliable service, both should result in similar values. In our case, we measured throughput here at the sender, for simplicity.

Since the emulator's performance is consistent for different configured rates, we decided to leave it in its simplest form (without further tuning).

4.2 Measuring the Effect of the Reliability Window

In the following experiments, the multicast group has four receivers with emulated receive rates (by configuration) of 100K, 150K, 200K and 250KB/sec. The sender's maximum data rate is 500KB/sec, with different minimum data rates for the different experiments. In each experiment, the sender sends 6000 packets of 1400 bytes. The sender is the repair head for all four receivers.

The tables below show the throughput and number of receivers pruned respectively, as we vary the sender's minimum data rate (which determines when to prune) and the reliability window size. Each column corresponds to a different minimum data rate used by the sender; and each row corresponds to a different reliability window size:

	50K	110K	160K	210K	260K
∞	88.6K	123.7K	171.7K	182.9K	195.5K
40	91.5K	133.6K	200.3K	233.7K	218.7K
25	107.9K	150.3K	195.7K	221.4K	226.1K
20	113.7K	127.5K	158.8K	205.2K	233.8K
15	142.6K	145.1K	187.0K	210.4K	231.7K
10	215.7K	221.4K	205.2K	215.7K	233.7K
5	227.4K	240.4K	233.7K	233.7K	233.7K

Table 2. Throughput for different pruning rates and reliability window sizes

The number of receivers pruned in each of the above sessions is given in the following table, with the same corresponding rows and columns:

	50K	110K	160K	210K	260K
∞	0	1	2	3	4
40	0	1	2	3	4
25	0	1	2	3	4
20	0	0	1	3	4
15	0	0	1	2	4
10	0	0	0	1	2
5	0	0	0	0	0

Table 3. Number of receivers pruned for different pruning rates and reliability window sizes

First, examine Table 3. For the first row, the ∞ reliability window represents the fully reliable case. Generally speaking, as we increase the pruning rate,

more receivers get pruned; also as we decrease the reliability window size (meaning accepting less reliability) fewer receivers get pruned. These results are very much as expected.

In Table 2, the throughput values generally increase as we decrease the reliability window size (meaning accepting less reliability), or increase the minimum data rate, which is as expected. This is not completely uniform. For example, in column 2 the throughput increases until the RWS is 25; then it drops at RWS 20, before increasing again. As can be seen in Table 3, for RWS 25 or above, one receiver is pruned whereas for RWS below 25, no receiver is pruned. In the latter case, the slowest receiver is definitely expected to drag down the throughput for the whole group until it is pruned.

Another seeming anomaly is that the throughput may not be always increasing with decreasing RWS, even for the same number of receivers pruned. This phenomenon is seen in column 3 and 4. This is due to the fact that throughput is measured for the whole session of transmitting 6000 packets, so the average throughput for a session when a particular number of receivers are pruned also depends on when the receivers are pruned. For example, the throughput for minimum data rate 210KB/sec and RWS 25 is 221.4KB/sec, yet the throughput for minimum data rate 210KB/sec and RWS 20 is 205.2KB/sec. In both of those experiments, three receivers were pruned. In the former case, the third receiver was pruned when the sender was transmitting sequence number 2000+, whereas in the latter case the pruning of the third receiver occurred at sequence number 5000+. So the third receiver influenced the session rate for a much longer time during the latter session.

These results are very interesting and encouraging, as they show the use of reliability window can have an effect on performance and pruning for the multicast group. To put things in perspective, however, the RWS at which we see an impact is relatively small. When the sending rate is 100KB/sec, and with packet size of 1400 bytes, we are transmitting 70+ packets per second. A reliability window of 25 translates into roughly 1/3 of a second of delay. On the other hand, the test network in these experiments was very small (LAN with four receivers). With a larger multicast group in a wide area setting, it is very likely to find more packets in the "pipe" in the steady state. Therefore, we expect larger RWS will also help in a multicast group's performance, which translates into longer time bound at the application level.

4.3 Service Differentiation According to Receive Rate Differences

As we discussed in the introduction, part of the motivation of this work is to use the reliability window as a mechanism to deliver different amounts of data to receivers with different receive rates. In a sense, this achieves a similar effect as layered content delivery [LAYERED].

From the same experiment setup described earlier, the following table shows the percentage of total data delivered to the application for the receivers with receive rates 100K, 150K, 200K and 250K B/sec.

	Rcv 1	Rcv 2	Rcv 3	Rcv 4
prune at 50K RWS=25	88.72	98.48	99.02	99.73
prune at 200K RWS=10	62.65	66.80	69.98	77.90

Table 4. Percentage of data delivered to receiver (applications) with different receive rates

In the first case (row), the sender's minimum data rate is 50KB/sec, which all receivers can happily sustain. No receiver is pruned, but there are occasional losses due to the way we emulate receive rates. Using a reliability window of size (RWS) 25, receivers 2, 3 and 4 all received most of the packets. Receiver 1 missed slightly more than 10% of the packets. Although the configured receive rate for Receiver 1 is 100K, its actual emulated receive rate is somewhat less (93K); being the slowest receiver in the group, receiver 1 also loses many more packets than the rest of the receivers.

In the second case (row), the sender's minimum data rate is 200KB/sec. At full reliability (RWS=∞), receivers 1, 2 and 3 would be pruned as they cannot sustain the minimum data rate. At RWS=10, however, they actually manage to receive a lower fraction of packets and avoid being pruned. Receivers 1, 2 and 3 received between 60%-70% of the packets, while receiver 4 (with receive rate above the minimum data rate of 200KB/sec) received close to 80% of the packets.

The key observation from these tests is that the reliability window allows different receivers with different receive rates to receive different amounts of data during the same session. The result is similar to what a layered multicast service might achieve. The way the "layers" are created is quite different. A layered multicast service creates the layers by conscious encoding of the original data stream into multiple streams. Here, the "layering" is achieved

dynamically, adapting to two forces: each receiver's ability to receive, and the application's suggestion of whether (certain) lost packets are worth recovering at a given time.

4.4 Experiments with Varying Receive Rates

As the number of receivers increases, and/or the duration of the multicast session becomes more extended, one inevitable consequence is that there will be more variability in receivers (in terms of their ability to receive). This can be either due to changes directly attributable to the receivers (e.g., resources shared with other applications on the receiver machine) or due to changes in the path to the receivers (e.g., varying traffic conditions). In the last experiment, we simulate this receiver population heterogeneity by introducing changes to the configured receive rate during a multicast session.

The sender is configured to send with a maximum data rate of 500KB/sec as before, and a minimum data rate of 150KB/sec. Each receiver is configured to start with a receive rate of 200KB/sec. At random times during the session, a receiver's receive rate drops to 125KB/sec for some time before reverting back to 200KB/sec to simulate receiver heterogeneity. The progress of the session measured at the sender's side is plotted in Figure 3, for constant reliability window sizes of ∞ and 10.

When RWS is ∞ (fully reliable), all four receivers eventually were pruned (the second and third prunes happened at about the same time; hence, it appears as one in the figure). This happened before the sender could finish transmitting all 6000 packets. When the RWS is 10, the sender was able to complete transmitting to all the receivers without pruning anyone. This latter case, of course, came with a cost of reliability. The four receivers lost 20.1%, 19.7%, 18.3%, and 14.9% of the packets, respectively. Also note that the curve for RWS=∞ has a lot of oscillations whereas the curve for RWS=10 is quite smooth. This is because in the RWS=10 case, the small reliability window caused most of the packet losses to go unnoticed by the sender and repair heads. Therefore, the sender sent at a steady rate without interference from repair traffic.

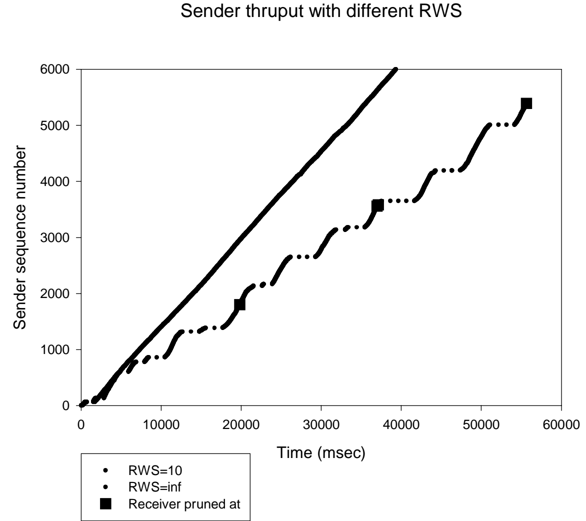

Figure 3 Measuring sender's progress when receivers have varying receive rates, for a couple of different reliability window sizes

The actual times the different receivers changed their receive rates were determined randomly during each of the above experiments, while using the same randomization algorithm. Therefore, the above comparison is strictly not apples-to-apples, but it is justified from a statistical sense. The following tables summarize the actual times when each receiver used the lower receiver rate in each experiment, expressed in terms of ranges of packet sequence numbers. In other words, the times are the times when those packets in the ranges are received.

	Periods (range of packets), RWS=10
rcv 1	[1136, 2267], [2925, 4112]
rcv 2	[1053, 1981], [2875, 4235]
rcv 3	[936, 2176], [3038, 3822]
rcv 4	[960, 1812], [3370, 354]

	Periods (range of packets), RWS=∞
rcv 1	[1238, till pruned]
rcv 2	[1046, 1584], [2661, 4400], [5095, till pruned]
rcv 3	[661, 1690], [3376, till pruned]
rcv 4	[1446, 1704], [2526, till pruned]

Table 5. Periods when receivers used a lower receive rate (125KB/sec), for the two experiments (RWS=10 and RWS=∞)

5. Review and Discussion of Related Work

In the broad context of reliable multicast services, we discussed in the introduction several different approaches to provide reliability that yield different trade-offs with performance, efficiency and scalability. These approaches include the use of forward error correction codes, dividing content into different layers, or using a repair tree to provide full reliability.

In this section, we discuss in more detail other proposals and techniques for relaxing the reliability service that we are aware of and contrast them to the reliability window and forgetbefore API approach we are proposing.

5.1 Time-bounded service

In RMTP-II [RMTP-II], a *time-bounded* service was proposed. Specifically, the sender attaches a timestamp to each data packet it sends, indicating the time the packet was generated. For a given session, there is a configured time-bound. When the packet's life has exceeded the time-bound, it can be dropped. The intention of the (time-bound) timestamp is similar to ours: to avoid wasting efforts in repairing packets that are no longer useful. In fact, the intention goes one step further in also trying to avoid *delivering* packets that are no longer useful.

We found two problems with the time-bounded approach in RMTP-II. First, every element that provides the time-bounded service (minimally all the receivers, but potentially also all the repair heads and other intermediate nodes in the delivery path) needs to calibrate its clock skew with the sender. Although this needs to be done only once per session (or once per long interval if the session continues indefinitely), it is still rather undesirable and a hindrance for this service.

Secondly, we find the use of the time-bound to decide on delivery too harsh, as we discussed in the implementation of the forgetbefore service. This is especially true if the receivers are dropping the packets because they are too old. There might be some applications for which this semantic is important, but we suspect these applications are rare.

Other than these differences, we consider our proposal of the forgetbefore API and the reliability window a way to implement the time-bounded reliability service.

5.2 Application Level Framing

The forgetbefore API allows applications to work with the transport in determining the level of reliability service. A much more general framework for accomplishing this application-transport cooperation was proposed in [ALF].

Application Level Framing (ALF) is an API that lets applications use meta-data to describe relationships between the application layer data units. This helps the transport to effectively decide which lost packets are worth recovering and which are not. Additionally, the receiver application can also participate in determining the relevance of data based on user interaction. For example, the application might be a whiteboard or cooperative editing application. What a user at the receiver is viewing or working on can also help determine whether certain lost data is worth recovering.

Although ALF is very powerful, we feel it is somewhat complicated for applications to use. In comparison, the forgetbefore API introduces minimum complexity to the applications.

5.3 Real-time Transports

Then there are also real-time or semi-real-time transports that are designed for audio/video applications. One example is RTP [RTP]. In trying to support real-time applications, these protocols must on the one hand be smart in recovering lost packets only if needed and useful; but on the other hand, deliver a smooth data stream to the application without jitter. The reliability service is usually purely based on receiver capabilities.

In comparison, the reliability windowing mechanism only tries to deal with the efficiency and robustness/scalability of the reliability service, with the sender's help. The smooth playback problem can probably be added in a different layer.

5.4 TRACK Architecture

The Internet Engineering Task Force (IETF) is in the process of standardizing several reliable multicast transport protocols. One of them is a tree-based ack protocol (TRACK). The TRACK architecture document [TRACK] contains a description of a sender reliability window. That is the same as the reliability window here, and is based on the proposal from this work.

6. Conclusions and Future Work

In this paper, we propose a reliability window mechanism and an accompanying forgetbefore API which are very simple to implement. Although we only implemented and experimented with a tree-based reliable multicast protocol, we believe the same technique can be applied to NACK-based semi-reliable protocols such as SRM [SRM] and PGM [PGM], and other real-time protocols, and possibly in the unicast context.

We justify the API by describing a number of ways applications might be able to make use of the API to relax the reliability service based on application data semantics.

We carried out a number of experiments to show that this mechanism indeed can help make the transport more robust, scalable and deliver higher performance from the multicast group point of view. The higher performance is effectively achieved by letting the slower receivers live with less reliability service rather than being pruned (discarded) from the group.

Moving forward, much further research would be useful. Some of the future research ideas are listed below:

- More testing would be helpful, perhaps with a more accurate receive rate emulation, and in a WAN environment.
- More experimentation with applying this mechanism to other reliable or semi-reliable multicast protocols will help prove the usefulness of this idea in other contexts.
- A study of how real applications might use the proposed API would solidify its claimed usefulness.
- By not trying to recover some lost packets, this mechanism certainly can affect the flow and congestion control algorithms of a transport protocol, especially if the congestion control relies on measured packet losses to determine data rate. In general, the forgetbefore mechanism tends to make the transport more aggressive. How this impacts the fairness of bandwidth sharing with other transports deserves more studies.
- The forgetbefore information may also be used by routers/switches to intelligently manage their queues. This, however, may require per-flow state in the routers and the tradeoff analysis can be quite complicated.

Acknowledgements

The authors received valuable help from many people. Specifically, Joe Wesley helped us with the initial design; Joe Provino gave us the receive rate emulator and worked with us on some of the experiments; Miriam Kadansky, Haifeng Zhu, Radia Perlman and Steve Hanna all participated in discussing the idea; and Miriam gave us very helpful review comments. The authors also thank Brian Whetten for fruitful discussions.

References

[ALF] S. Raman and S. McCanne, "Scalable Data naming for Application Level Framing in Reliable Multicast", Proceedings of ACM Multimedia98, 1998.

[FEC] J. Nonnenmacher and E. W. Biersack, "Reliable Multicast: Where to Use Forward Error Correction", Proceedings 5th Workshop on Protocols for High Speed Networks, pp. 134-148, October 1996.

[LAYERED] S. McCanne et al, "Receiver-driven Layered Multicast", Proceedings of Sigcomm'96, 1996.

[PGM] T. Speakman et al, "PGM Reliable Transport Protocol", IETF Internet Draft draft-speakman-pgm-spec-03.txt, work in progress.

[PRUNING] D. Chiu et al, "Pruning Algorithms for Multicast Flow Control", Sun Labs Technical Report, TR-2000-85. To appear in NGC2000, Stanford, November, 2000.

[RMTP-II] B. Whetten et al, "The RMTP II Protocol", Work in progress, draft-whetten-rmtp-ii-00.txt, April, 1998.

[RTP] H. Schulzrinne et al, "RTP: A Transport Protocol for Real-Time Applications", IETF RFC 1889, January, 1996.

[SRM] S. Floyd et al, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", Proceedings of ACM Sigcomm'95, pp 342-356, 1995.

[TRACK] B. Whetten et al, "TRACK Architecture: A Scalable Real-time Reliable Multicast Protocol", Work in progress, draft-ietf-rmt-track-arch-00.txt, July, 2000.

[TRAM] D. Chiu et al, "TRAM: A Tree-based Reliable Multicast Protocol", Sun Labs Technical Report TR-98-66, 1998.

About the Authors

Dah Ming Chiu is a researcher at Sun Microsystems Laboratories in Burlington, Massachusetts. His recent research is on reliable multicast protocols and multicast flow/congestion control. Prior to Sun, he worked at Digital Equipment Corporation, and AT&T Bell Labs. His research interests include performance modeling and analysis of network protocols, distributed systems, and WWW-based applications. He received a Ph.D. degree from Harvard University and a B.Sc. degree from Imperial College, University of London.

Jaiwant Mulik is a Ph.D. student in computer science at Temple University, Philadelphia. His current area of research is in reliable multicast, and in particular network support for reliable multicast. He received an M.S. degree in Computer Science from Temple University, and a Bachelor degree in Computer Engineering from Mumbai (formerly Bombay) University, India in 1998.