An API for Scalable Reliable Multicast

Jim Gemmell
Microsoft Research
301 Howard St, #830
San Francisco, CA 94105
jgemmell@microsoft.com

Jörg Liebeherr
Polytechnic University
Department of Electrical Engineering
6 MetroTech Center
Brooklyn, NY 11201
jorg@catt.poly.edu

Dave Bassett
Computer Science Department
University of Virginia
Charlottesville, VA 22903
dgb2n@cs.virginia.edu

Abstract

Most approaches to scalable reliable multicast utilize receiver-oriented retransmissions. Defining an API for receiver-oriented reliable multicast is difficult because it is not clear how to manage the sender's cache and to schedule repairs. We outline an approach to defining an API based on logical cache persistence that addresses these problems.

1 Introduction

There are many scenarios in which the same data must be delivered over a packet switched network to a large set of receivers. The Internet enables efficient multi-point multicast transmissions through IP multicast by allowing data transmission to all receivers with a single send. IP multicast provides an unreliable, connectionless, datagram service, without guarantees for correct delivery, in-order delivery, or delivery without duplication. While an unreliable service is adequate for many applications, for example, audio and video, many other applications require, or can benefit from, retransmission of lost packets. Examples of such applications include slide shows, file-transfer, multi-party shared document editing and electronic whiteboards. Common to these applications is that any transmitted application-level data is relevant ("persistent") for long time periods relative to the time required for recovering lost data.

Since the inception of IP multicast, various proposals have been made for reliability mechanisms that provide delivery guarantees on top of IP multicast. In this paper we explore the issues involved in defining an API for reliable multicast protocol for the Internet that can scale to millions of receivers.

Some would argue that scalable reliable multicast must be implemented at the application level, i.e., that it is not possible to offer multicast reliability as a separate service that is not integrated into the application [FLO95]. To a certain extent this argument has merit. The widely varying requirements of different applications for a scalable reliable multicast service are broad enough to prohibit a general-purpose solution. However, we will show that it is possible to provide a service that is useful to a sizeable category of applications. In this paper, we define an API for reliable, scalable multicast, while only making minimal assumptions about the protocol. Our goal is to have an API that is useful for experimenting with different scalable protocols. The API that we propose could equally use another protocol to achieve reliable multicast. Defining an "on the wire" protocol for scalable reliable multicast is beyond the scope of this paper.

2 Scalable Reliable Multicast

Since the early 1980's, many protocols have been introduced for reliable multicast communications [BIR91, CHA84, FLO95, HOL95, PAU97, TAL95, WHE95, YAV95]. There are many ways to implement a reliable multicast protocol. Protocols are referred to as ***sender-oriented*** if the sender of a packet is responsible for ensuring that all receivers have obtained a copy of the packet. Protocols are called ***receiver-oriented*** if the responsibilities for detecting missing packets lie with the receivers. A protocol with ***unordered*** delivery does not make guarantees on the order in which packets are delivered to the receiver. The delivery is called ***source-ordered*** if the protocol maintains the order of transmission for each sender, i.e., a receiver obtains the packets from a specific sender in the same order in which the packets were transmitted by that sender. A ***totally-ordered*** protocol ensures that all packets are received by all receivers in the same order. Examples of protocols that employ source and/or total ordering are RMP [WHE95] and SCE [TAL95].

Typically, with a sender-initiated approach, the receivers acknowledge (ACK) each packet they receive. It is up to the sender to keep track which packets have been ACKed, and to resent if they are not ACKed within a certain time interval. This means

that the sender must keep state information and timers for each receiver. Receiver-oriented schemes shift the burden to the receiver. It is up the receiver to detect a lost packet and request retransmission; the sender need not keep state for each receiver. Typically, to achieve this the sender puts a sequence number in each packet transmitted. When it has no packets to transmit, it sends a heartbeat packet containing the last sequence number. By looking at the sequence number, the receiver can detect if it has missed any packets and request for them to be retransmitted by sending a negative acknowledgment (NACK) back to the sender.

Sender-initiated schemes have difficulty with scalability. Keeping state for each receiver becomes infeasible as the number of receivers stretches up into the millions. Furthermore, the volume of ACK messages may overwhelm the sender, referred to as the *Implosion Problem* [CRO88, DAN89, JON91]. Figure 1 shows a sender keeping state for each receiver and suffering from NACK implosion.

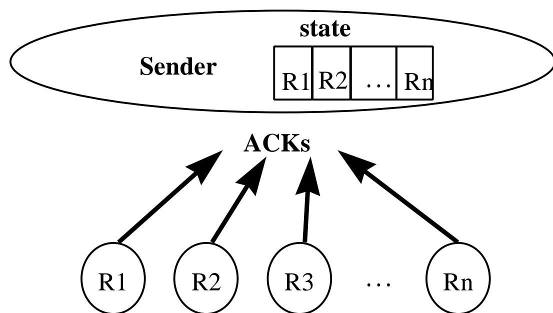

Figure 1 - State implosion with sender-initiatedreliability.

Receiver-oriented schemes solve the problem of keeping state information at the sender, but still may suffer from NACK implosions. One approach to avoiding NACK implosions is to organize the participants into a tree, thereby bounding NACKs according to the order of the tree (e.g. TMTP [YAV95], LBRM [HOL95]). Another approach is to employ ***NACK Suppression***, originally proposed in [RAM87]. NACK suppression works as follows: whenever a receiver detects a lost packet, it waits for a random amount of time and then multicasts its NACK to the whole group and sets a timer. If it receives another NACK for the same packet while it is waiting, it will back off and not send the NACK, but will set its timer as if it had sent the NACK. Essentially, the NACK is delayed in the hope that someone else will generate it. If the timer expires without receiving the corresponding packet, then it repeats the process of randomly backing off and sending a NACK. The SRM framework [FLO95] is an example of a scheme utilizing NACK suppression. NACK suppression can also be used in conjunction with hierarchy, as in TMTP [YAV95].

Basic receiver-based reliable multicast has the source generating sequence numbers in each packet. When it is not sending data packets, it sends ***heartbeat packets*** (containing the last sequence number) at a regular rate. The receiver can detect a loss when it does not receive a data or heartbeat packet within a given interval.

The discussion in this chapter has exemplified that sender-oriented protocols introduce severe scalability problems. Also, the overhead introduced by a requirement for totally-ordered delivery is prohibitive. For the remainder of the paper, we assume that reliable multicast is achieved via receiver-oriented protocol. Using mechanisms such as NACK suppression, heartbeat packets, etc., one can contain the volume of control traffic and support very large receiver sets.

3 Defining a Receiver-Oriented API - The Problems

When designing a sender-oriented reliable multicast transport protocol, there are a number of options, such as ordering, as discussed above. However, the basic semantics of the protocol are well defined and easily understood: for each receiver, the sender must ensure that it gets a copy of each packet sent. In contrast, the semantics of a receiver-oriented reliable multicast are not so obvious. In particular, sender cache persistence and scheduling of re-sends pose difficulties.Since a sender in a multicast group cannot store a packet forever, a protocol must specify rules that define how long a sender must hold on to a transmitted packe; this is referred to as cache persistence. The problem with cache persistence is exposed by asking the following questions:

- When can a packet be deleted from the sender's cache?
- When should NACKs be withheld since the corresponding data is no longer cached?

Clearly, the questions are related. There is no point NACKing a packet that is no longer cached, and there is no point in caching something that will never be sent again. The issue at stake is determining when the protocol is finished with a given packet.

With receiver-oriented reliability, the answers to the above questions are application dependent. For some applications it is desirable to store all packets persistently, i.e., cache packets forever, and never withhold a NACK for a missing packet. However,

other applications may allow withholding the transmission of NACKs. As an example of the latter group, consider a multicast transmission of a set of slides; this slide-show application could decide to conserve bandwidth by only transmitting data from the current slide. NACKs for previously transmitted slides should be withheld, since they are not currently being viewed. Or, consider an application that transmits on stock quotes with updates at least daily. In this application, only packets that are less than 24 hours old need to be cached; NACKs for packets that are older than 24 hours are superfluous.

The second problem concerns scheduling of retransmissions and is embodied in the following question:

- Following a NACK reception, when should the retransmission of a packet (A.K.A. repair) be scheduled?

Again, the answer is application dependent. For example, repairs may be considered highest priority and sent immediately, prior to any new data. Or repairs may be considered lowest priority and need to wait for an empty transmission queue. Disregarding new data, there is also the question of scheduling repairs when several repairs are outstanding. The most obvious approach is to schedule them first-come-first-serve, but applications may desire a different order.

We should note that TCP answers repair scheduling questions on behalf of applications, and transmits repairs ahead of new data. However, TCP only deals with a single receiver. Applied to a large receiver set, a repairs-first policy allows one poor receiver to hold up the transmission for millions of other receivers in a reliable multicast.

4 Defining a Receiver-Oriented API - The Solution

From our discussions in the previous sections it should become clear that a one-size-fits-all service to provide scalable reliable multicast is not possible. However, it is possible to design a "one-size-fits-many" service, with the ability to accommodate some important caching strategies and to schedule repairs, which can be exploited by a large variety of applications. Such a service can be limited to relieving the applications from the burden of caching data and performing repairs. With such a service all senders can perform transmission of packets without concerning themselves with reliable delivery; reliability is provided by the multicast protocol. Similarly, after initializing the service, every receiver can accept packets without concern regarding transmission of NACKs, NACK suppression, etc.

As discussed above, one of the key questions is how to manage caching of packets at the receiver. Two obvious caching strategies are:

- *Caching with Spatial Persistence:* Cache the most recent N packets, where N is a dynamically tunable parameter.
- *Caching with Temporal Persistence:* Cache packets for time T, where T is dynamically tunable parameter.

If N and T are static values over the course of a session, then these schemes are unlikely to match the needs of a real application. By making them dynamic, they become more useful, but are still awkward because applications must track the number of packets sent or time sent. A more useful scheme than spatial or temporal persistence is given as follows:

- Give each packet a non-decreasing "epoch" value, which is transmitted in its header. Cache for E epochs, where E is a parameter that may be updated with each packet sent.

In [BAS96], the above scheme was referred to as *logical persistence*. Clearly, using an epoch-based cache deletion strategy is not general because it only allows variations on a least-recently-used (LRU) cache deletion strategy. It could not satisfy the application that, say, wanted to cache even packets for a minute and odd packets for a day. To extend the usefulness of this scheme, the service should also support packets being sent which remain in the cache for the duration of the session. In our example API, this is accomplished by sending with an epoch of EPOCH_PERSISTENT.

While not being completely general, combining epoch-based deletion with session-persistent packets enables us to define a service that is useful for a large class of applications. In particular, network shows (e.g. presentations, dramas, etc.) tend to have scenes, slides, or sections that naturally break up in a way that lend themselves to epoch-based cache deletion. The Multicast PowerPoint application discussed below illustrates this. File distribution applications would seem at first glance to not be suited to epoch-based caching. By sending file A, and then start sending file B, file A does not necessarily become stale immediately. However, in some cases, using epoch-based caching for file transfer does make sense. For example, the policy may be to only multicast repairs for the most recent file

transmissions, and to require all older repairs to be accomplished by (unicast) retrieval from a server.

The epoch-based scheme makes the assumption that the underlying protocol somehow communicates information about caching (i.e. the current epoch number and the number of epochs to cache, *E*). Note that the way in which this is communicated need not be static in the protocol; the session announcement could indicate how this is to be done. In particular, the number of bits used to represent the epoch and Ecould be different for different sessions.

For scheduling repairs, our API supports three simple strategies: repairs first, new data first, and first-come-first-serve.

5 The API

Figure 2 shows a prototypes for the proposed API. We show only the calls related to reliable multicast. Naturally, there must be a way to join and drop a multicast group, set the TTL, etc. Furthermore, rate control should be performed at a lower level so that NACK and repair traffic does not violate the rate policies.

The `Send()` call sends a packet and returns an error status. If the send is successful, then the sequence number pointed to by `pSequence` is updated to the sequence number assigned to the packet. It is the responsibility of the application to save this value if it wants to retransmit the packet. `SetEpoch()` sets the epoch number to be used for subsequent sends, and the value E– the number of epochs to be cached. `SetScheduleOption()` is used to indicate how to schedules repairs. `NumOutstandingNACKs()` returns the number of NACKs which have been received for which no repair has been sent yet. This gives senders an indication as to whether any receivers are still missing data before terminating a session (when it returns zero there still may be unsatisfied receivers whose NACKs are being lost).

`SetHeartBeat()` is used to control the heartbeats of the underlying protocol. It sets a payload to be used for the heartbeat packets so that the heartbeats can contain application specific data. It also sets the minimum and maximum heartbeat rate (some protocols, like LBRM [HOL95] have variable heartbeat rates).

```
//client can use EPOCH_FIRST or higher.
Lower epoch values are reserved
#define EPOCH_PERSISTENT  4 //use for
items that never become stale
#define EPOCH_FIRST 5 //1st valid user
epoch
```

```
//Set scheduling strategy, return error
status
// Opt must be one of: FCFS,
REPAIRS_1ST, or DATA_1ST
int    SetScheduleOption (tSchOpt Opt);
//IN option

// sends a packet unreliably – return
error status
int    SendUnRel(    char* pPacket, //IN
packet to send
                            long  lLen
                  ); //IN length of
                  packet

// Sets the payload for heartbeat
packets and the
// rate to send heartbeats when the
session is idle
int    SetHeartBeat(char *pPayLoad,
//IN payload for heartbeat
                    long lLen, //IN
length of payload
                    long lMinRate, //IN
min rate per hour
                    long lMaxRate);
//IN max rate per hour

//set the current epoch and number of
epochs to cache
int    SetEpoch(
                    tEpoch Epoch, //IN
                  current epoch number
                    tEpoch E); //IN
                  epochs to cache

//send a packet, return error status
int    Send(
                    char   *pPacket,
             //IN packet to send
                    long   lLen, //IN
             length of the packet
                    tSeq*
             pSequence);//OUT sequence
             number assigned

// Returns the number of NACKs received
for which a repair has not yet been
sent
int    NumOutstandingNACKs(void);

// Reliable multicast receive
int    Recv(  char* pBuffer, //OUT
buffer to put packet in
                  int pnBufferLen, //IN
         size of buffer
                  tEpoch* pEpoch, //OUT
         epoch of packet received
```

```
            tSeq* pSeq); //OUT
        sequence number of packet
        received
```

Figure 2 - Prototype Definition of the API.

6 Experience and Conclusion

The Multicast PowerPoint project at Microsoft Research made use of a reliable multicast software library based on the calls described above. Multicast PowerPoint is a modification to PowerPoint conferencing that allows PowerPoint slides to be multicast. It uses a modified form of SRM to achieve reliability. Multicast PowerPoint was demonstrated in the Microsoft booth at the ACM 97 Conference in San Jose, California, and was also used to transmit some ACM 97 presentations (those that used PowerPoint) to desktops on the Microsoft corporate network. The API design discussed in this paper was also used at the University of Virginia for implementing the Tunable Multicast Protocol (TMP) [BAS96]. TMP exploited the notion of logical persistence based on user-defined epochs. For example, in a multicast file transfer application that was implemented with TMP, each transmitted file constitutes a separate epoch.

It is infeasible to create a scalable reliable multicast protocol that will suit all applications. The biggest roadblock lies in the choice of caching strategy. Since there is no single caching strategy that suits all applications, there is a widespread belief that the only way to arrive at a general purpose protocol is to leave caching up to the application. In this paper we have proposed an epoch-based caching strategy that is useful to a wide range of applications. We have outlined an API based on epoch-based caching which allows application developers to easily harness the power of scalable reliable multicast.

7 References

[ARM92] Armstrong, S., Freier, A. and Marzullo, K., "Multicast Transport Protocol", *RFC 1301, Internet Engineering Task Force*, February 1992.

[BAS96] Bassett, D. G., "Reliable Multicast Services For Tele-Collaboration", *Masters Thesis, University of Virginia*, 1996.

[BIR91] Birman, K., Schiper, A., and Stephenson, P., "Lightweight Causal and Atomic Group Multicast", *ACM Transaction on Computer Systems*. 9(3): 272-314. August 1991.

[CHA84] Chang, J. M., and Maxemchuck, N. F., "Reliable Broadcast Protocols", ACM Transactions on Computing Systems, 2(3):251-273. August 1984.

[CLA90] Clark, D. D., and Tennenhouse, D. L., "Architectural Considerations for a New Generation of Protocols", *Proc. of ACM SIGCOMM '90*, Pages 201-208, September 1990.

[CRO88] Crowcroft, J., and Paliwoda, K., "A Multicast Transport Protocol", *Proc. of ACM SIGCOMM '88*, pp. 247-256, 1988.

[DAN89] Danzig, P., "Finite Buffers and Fast Multicast", *ACM Sigmetrics `89, Performance Evaluation Review,* Vol. 17, pp. 108-117, May 1989.

[FLO95] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L., "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", *Proc. of ACM SIGCOMM '95*, Cambridge, MA, August 1995.

[HOL95] Holbrook, H.W., Singhal, S.K., and Cheriton, D.R., "Log-based Receiver-Reliable Multicast for Distributed Interactive Simulation", *Proc. of SIGCOMM '95*, Cambridge, MA, August 1995.

[JON91] Jones, M.G.W., Sorensen, S. A., and Wilbur, S., "Protocol Design for Large Group Multicasting: The Message Distribution Protocol", *Computer Communications,* 14(5):287-297, 1991.

[PAU97] Paul, S., Sabnani, K. K., Lin, J. C.-H., and Bhattacharyya, S., "Reliable Multicast Transport Protocol (RMTP), , IEEE *Journal on Selected Areas in Communications*, Vol. 15, No. 3, pp. 407 – 421, April 1997.

[RAM87] Ramakrishnan, S. and Jain, B. N., "A Negative Acknowledgement With Periodic Polling Protocol for Multicast over LANs", *Proc. of IEEE Infocom '87*, pp. 502-511, March/April 1987.

[TAL95] Talpade, R., Ammar, M. H., "Single Connection Emulation: An Architecture for Providing a Reliable Multicast Transport Service", *Proc. of 15th IEEE Intl. Conf. on Distributed Computing Systems*, Vancouver, June 1995.

[WHE95] Whetten, B., Montgomery, T., and Kaplan, S., "A High Performance Totally Ordered Multicast Protocol", *International Workshop on Theory and Practice in Distributed Systems,* 5-9 Sept. 1994, Springer-Verlag, pp.33-57.

[YAV95] Yavatkar, R, Griffioen, J, Sudan, M, "A Reliable Dissemination Protocol for Interactive Collaborative Applications", *Proc. of ACM Multimedia 95*, Pages 333-343, November 1995.