Boston University

OpenBU **http://open.bu.edu**

Computer Science CAS: Computer Science: Technical Reports

1999-09-15

Adaptive Reliable Multicast

Yoon, Jaehee

Boston University Computer Science Department

Yoon, Jaehee; Bestavros, Azer; Matta, Ibrahim. "Adaptive Reliable Multicast", Technical Report BUCS-1999-012, Computer Science Department, Boston University, September 15, 1999. [Available from: http://hdl.handle.net/2144/1789]

https://hdl.handle.net/2144/1789

Boston University

Adaptive Reliable Multicast*

Jaehee Yoon
jaeheey@cs.bu.edu

Azer Bestavros
bestavros@cs.bu.edu

Ibrahim Matta
matta@cs.bu.edu

Computer Science Department
Boston University
Boston, MA 02215

September 1999

Abstract

An increasing number of applications, such as distributed interactive simulation, live auctions, distributed games and collaborative systems, require the network to provide a *reliable multicast* service. This service enables one sender to reliably transmit data to multiple receivers. Reliability is traditionally achieved by having receivers send negative acknowledgments (NACKs) to request from the sender the retransmission of lost (or missing) data packets. However, this Automatic Repeat reQuest (ARQ) approach results in the well-known *NACK implosion problem* at the sender. Many reliable multicast protocols have been recently proposed to reduce NACK implosion. But, the message overhead due to NACK requests remains significant. Another approach, based on Forward Error Correction (FEC), requires the sender to encode additional redundant information so that a receiver can independently recover from losses. However, due to the lack of feedback from receivers, it is impossible for the sender to determine how much redundancy is needed.

In this paper, we propose a new reliable multicast protocol, called *ARM* for Adaptive Reliable Multicast. Our protocol integrates ARQ and FEC techniques. The objectives of ARM are (1) reduce the message overhead due to NACK requests, (2) reduce the amount of data transmission, and (3) reduce the time it takes for all receivers to receive the data intact (without loss). During data transmission, the sender periodically informs the receivers of the number of packets that are yet to be transmitted. Based on this information, each receiver predicts whether this amount is enough to recover its losses. Only if it is not enough, that the receiver requests the sender to encode additional redundant packets. Using ns simulations, we show the superiority of our hybrid ARQ-FEC protocol over the well-known Scalable Reliable Multicast (SRM) protocol.

Keywords: Reliable Multicast, FEC using Reed-Solomon and Tornado Codes, Simulation.

*This work was supported in part by NSF research grants ESS CCR-9706685, CAREER ANIR-9701988, and MRI EIA-9871022.

1 Introduction

An increasing number of distributed applications involve one sender transmitting data to many recipients concurrently. Examples include distributed games, tele-conferencing, live auctions, concurrent engineering, and interactive distance learning. Such applications require the underlying network to provide a one-to-many (*multicast*) communication.

With multicast support, the source sends only one copy of a message to an address that represents the group of recipients. The message then traverses a tree of links rooted at the sender and whose branches lead to the individual receivers. The message is replicated only where paths to different receivers diverge, thus conserving network resources.

Many networks provide multicast support. For example, broadcast networks, such as Ethernet or satellite, easily support multicast as multiple hosts can read a specially addressed packet off the broadcast medium. Recently, the Internet has implemented multicast over the overlay MBONE network [7]. As for point-to-point (unicast) traffic, the Internet Protocol (IP) multicast service is unreliable best-effort. To support the needs of applications, *reliable multicast* has been an active area of research and many reliable multicast transport protocols have been recently proposed.

Reliable multicast transport protocols can be categorized into two groups: (1) ARQ (Automatic Repeat reQuest) based protocols, which retransmit lost data upon request, and (2) FEC (Forward Error Correction) based protocols, which transmit redundant data, called parity data, along with the original data. The ARQ technique is appropriate for unicast communication, such as in the Transmission Control Protocol (TCP), but problems arise when a straightforward ARQ based protocol is used in a multicast setting. A major problem is the so-called *NACK implosion problem*, which takes place when every receiver sends a negative acknowledgment (NACK) message for the same lost packet back to the sender. SRM (Scalable Reliable Multicast) [1] is one of the most popular ARQ based protocols that have been proposed to reduce this NACK implosion. Reliable Multicast Transport Protocol (RMTP) [2] and Tree-based Multicast Protocol (TMTP) [3] are other examples of ARQ based transport protocols.

The basic principle of FEC is that the original data is encoded to obtain some parity data, which is sent by the sender along with the original data. This parity data is used by a receiver to *independently* recover lost data. The advantage of FEC is that different receivers can recover their different lost packets using the *same* parity packet(s). With FEC, retransmission can, in principle, be completely avoided, thus significantly reducing latency to receive all data intact (without loss) at all receivers. However, FEC by itself cannot provide full reliability, because the sender does not receive any feedback from the receivers about their losses, thus there is no way for the sender to know how much redundancy is needed to fully recover lost data.

Many reliable multicast protocols based on merging FEC and ARQ techniques have also been proposed. The main idea behind these hybrid ARQ-FEC approaches is that the sender encodes data and transmits the original data along with some redundant data. If a receiver detects losses which cannot be recovered from the data received from the sender, then the receiver requests the *number* of (lost) packets that it needs to fully recover the original data. The benefit of this approach is that the sender and receivers need to be only aware of the number of lost packets and *not* their sequence numbers. Thus, the *same* (repair) packets sent by the sender, in response to NACK requests from receivers who may have lost *different* packets, can be used by all receivers for loss recovery.

Hybrid ARQ-FEC approaches clearly reduce the number of repair packets while reducing NACK implosion. However, the problem of setting the proper redundancy so as to avoid retransmission in such hybrid protocols still remains. If the sender uses a fixed redundancy that is independent of the loss rates experienced by receivers, then if the loss rates are much higher than what FEC is able to mask, the sender

may suffer from NACK implosion. Also, the need for retransmissions will increase the overall transmission latency. On the other hand, if the loss rates are much lower than predicted, bandwidth is wasted due to the unnecessary redundant data sent to receivers. Finally, with a static redundancy, additional repair packets may not be available when NACK requests arrive from receivers. This suggests that the sender should adjust the amount of redundancy *dynamically* based on feedback from the receivers about their loss state. This, however, raises challenging issues regarding the times of these adjustments and the loss conditions under which a receiver sends a feedback (NACK) message.

Our Contribution: We propose an *Adaptive Reliable Multicast* protocol, called ARM, that is based on a hybrid ARQ-FEC approach. In our protocol, the sender dynamically adjusts the amount of redundancy needed for full recovery from losses. This is achieved as the data is being transmitted. The sender in ARM keeps track of the required redundancy by periodically sending probes which are piggy-backed on the data packets. Based on their estimated loss rates, receivers predict the number of packets they will successfully receive, and *only if* this number is not sufficient to fully recover the original data, a receiver responds to the probe with a NACK. Upon receiving this NACK information, the sender readjusts the required redundancy, and encodes more repair packets if needed.

ARM has several salient features: (1) the proper number of redundant packets is encoded based on the loss state of receivers as it changes over the lifetime of the data transmission, (2) the transmission time needed for all receivers to receive the original data intact is significantly reduced, and (3) the message overhead due to data and NACK transmission is significantly reduced.

The reduction in transmission times is due to the elimination of most NACK requests as a result of the proper dynamic adjustment of redundancy employed in ARM. Furthermore, encoding additional redundancy to combat expected (future) losses is overlapped with data transmission.

The reduction in message overhead is due to the fact that the sender encodes *new* repair packets as needed, hence receivers do not receive duplicate (useless) packets. Also, receivers do not send NACK requests if the redundancy that is currently estimated is enough for full recovery, hence dramatically reducing NACK implosion. Finally, once a receiver receives the number of packets needed to fully recover the original data, it can leave the multicast group, thus the multicast routing tree shrinks (and hence less resources are consumed) over time.

The rest of paper is organized as follows. Section 2 discusses related work. Section 3 describes our proposed ARM protocol in detail. Section 4 presents our simulation results. Section 5 concludes the paper with a summary of findings and future work.

2 Related Work

The first category of reliable multicast transport protocols is based on ARQ (Automatic Repeat reQuest). Here, the sender retransmits lost data upon request from the receiver. A straightforward application of ARQ in a multicast setting results in the so-called *NACK implosion* problem. This problem occurs when every receiver sends a NACK message to request retransmission of the same packet, causing an implosion at the sender. To prevent this implosion of control packets, Xpress Transport Protocol (XTP) [17] proposed that a receiver multicasts control packets to the entire group. A receiver waits for a random time before sending a NACK packet, and refrains from sending a NACK if it sees a NACK from another receiver for the same packet. SRM (Scalable Reliable Multicast) [1] uses similar mechanisms to control the sending of request (NACK) and repair (retransmitted data) packets. In SRM, the random delay before sending a request (repair) packet is a function of the receiver's distance from the node that triggered the repair (request). Each node estimates its distances from other nodes by multicasting session messages. The

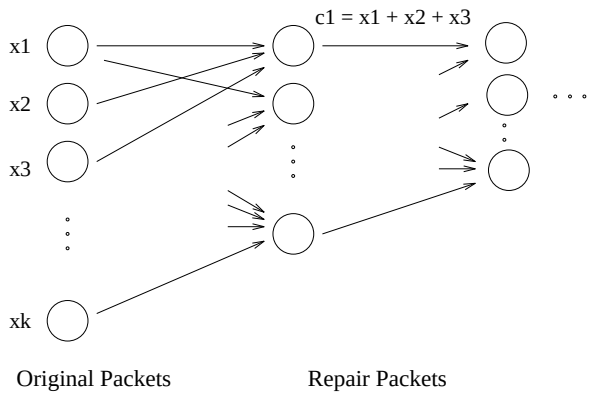

Figure 1: Dynamic encoding using Tornado (see [6])

random timer is set to be inversely proportional to the distance. Thus, although a number of receivers may all miss the same packet, a receiver close to the point of failure is likely to timeout first and multicast the request. Other receivers that are also missing the data hear the request and suppress their own request.

Another category of reliable multicast protocols is based on FEC (Forward Error Correction). Here, the original data is encoded to obtain additional repair packets that are used to recover data packet loss. A popular coding technique is the Reed-Solomon Code (RSC) [13], which is used in many FEC-based reliable multicast protocols [10, 9].[1] However, the encoding and decoding times of RSC are relatively high, and increase as the size of the original data block increases. Thus, the data block size has to be kept as small as possible.

Another FEC coding technique is Tornado [4], which promises fast encoding and decoding, and overcomes the limitation of small block size of Reed-Solomon codes. It produces linear-time encodable and decodable codes based on a series of random bipartite graphs, as depicted in Figure 1. The leftmost layer stores the original data, and the redundant packets are generated by taking the exclusive-or of packets in the previous stage. Assume c_1 in Figure 1 is generated from x_1, x_2, and x_3. If x_3 is lost during transmission, the receiver can recover x_3 upon receiving x_1, x_2, and c_1. The technique is described in detail in [4, 5]. In [6], Byers, Luby and Mitzenmacher propose the use of "Digital Fountains"—a reliable multicast protocol that uses Tornado codes.

In [6], the sender (server) initially chooses a stretch factor, N/K. It generates an encoding of the original data, where K denotes the number of original packets and N denotes the total number of packets (including the redundant packets). The sender sends the N packets repeatedly until every receiver receives $K' = (1 + \epsilon)K$ packets, where ϵ is the reception overhead. In [6], ϵ was found to be very small. Before sending, packets are permuted to reduce the probability of receiving duplicate packets. The advantage of Tornado codes over RSC is that one can trade off a negligible increase in reception overhead for a substantial decrease in encoding and decoding times.

The work described in [15, 8, 16, 18, 9, 10] suggest merging FEC and ARQ techniques for reliable multicast. Here, the sender transmits the original data along with some redundant data packets. Receivers request the number of lost packets that are needed for full recovery. Upon receipt of this feedback NACK information, the sender transmits the requested number of packets that satisfy the worst receiver's loss. This technique reduces the number of repair packets while reducing NACK implosion. A challenging issue here is to determine when should receivers send their feedback (i.e. how many packets should be transmitted by the sender before receivers send their feedback to the sender?) and how should the sender

[1] FEC techniques using RSC codes have also proved useful for tackling the fragmentation of IP over ATM [21] and for real-time communication protocols [20].

make use of this feedback information to adjust the amount of redundant (repair) data.

In [9] Rubenstein et al suggested using a proactive FEC technique with ARQ. In particular, when the sender starts transmission, it sends data packets along with some repair packets proactively. This scheme assumes that data is grouped in small blocks, and thus the proactive factor is statistically calculated based on feedback about the reception of previous blocks. If the block size is large, or if the data is not grouped in blocks, it is unclear how the sender would calculate this proactive factor.

Our ARM protocol allows the sender to receive feedback about the packet loss rates experienced by receivers, *without* assuming such block boundaries. For this, the sender takes a more active role in estimating the appropriate level of redundancy to be injected proactively. This is done using *probes* sent to receivers periodically. As data is being transmitted, the sender informs the receivers of how many packets are yet to be transmitted. Upon receiving this information, receivers decide whether this forthcoming number of packets (plus the number of already received packets) will be enough to recover the original data (through proper decoding). This decision takes into consideration an estimate of the loss rate at the receiver. If this condition is satisfied, the receiver does not need to respond to the probe. Otherwise, the receiver sends a NACK message, which includes loss information at the receiver. Upon receiving the NACK, the sender adjusts the level of redundancy it is using. Since the redundancy is set to satisfy the maximum loss across all receivers, the NACKs of receivers experiencing lower loss rates are effectively suppressed. Therefore, our protocol dramatically reduces the number of feedback (request) messages. We quantify this in our simulations detailed in Section 4.

3 The Adaptive Reliable Multicast Protocol

In this section we detail our Adaptive Reliable Multicast (ARM) Protocol. We start with an overview. Next, we present a detailed description of the algorithm. Finally, we highlight two of the distinct aspects of ARM—namely ARM's probing mechanism and RTT estimation.

3.1 ARM Protocol Overview

As with other FEC-based protocols, we assume that all receivers know the least number of packets, K', that they must receive to be able to reconstruct the original data. Therefore, a receiver is not concerned about receiving a *particular* set of packets. Rather, it is concerned with receiving a minimum *number* of distinct packets. To reduce latency, the sender starts by multicasting original packets. In the meantime, the sender encodes the original data to obtain additional repair packets. Thus, the encoding time is *overlapped* with the data transmission time. To reduce the number of packets transmitted by the sender, the sender transmits probes (piggy-backed on data packets), and (a small subset of the) receivers respond with NACKs that allow the sender to estimate the number of repair packets that are needed to mask the effects of *current* loss conditions.

We assume that the encoding and decoding technique used by the sender and receivers is Tornado [4].[2] Our choice is motivated by the following reasons: (1) the encoding and decoding times are small, (2) it is easy to encode more repair packets with Tornado (cf. Section 2) as ARM adjusts the stretch factor dynamically, and (3) the block size can be relatively larger than other encoding techniques. As mentioned earlier, with Tornado, a receiver must receive at least $K' = (1 + \epsilon)K$, where ϵ is the reception overhead. In [6], ϵ was found to be very small.

[2]The use of Tornado codes—while preferred—is not necessary. In particular, ARM could be used with traditional Reed-Solomon Coding (e.g. Rabin's Information Dispersal Algorithm [19].)

To determine the amount of redundancy needed, the sender periodically sends a probe with information about the number of packets that are yet to be sent. We denote this quantity by *PTS*.

Upon receiving a probe, a receiver uses the *PTS* information to predict whether or not the yet-to-be transmitted packets are enough to reconstruct the original data—based on the current loss rate it is experiencing. If the forthcoming packets are *insufficient* to recover the original data, the receiver sends a (unicast) NACK to the sender, which includes the maximum sequence number that should be delivered. If the forthcoming packets are *sufficient* to recover the original data, the receiver simply does not need to respond to the sender's probe.

Upon receiving a NACK from a receiver, the sender increases the maximum sequence number to accommodate the needs of that receiver. Future probes multicast by the sender will reflect this increased maximum sequence number. This feedback mechanism enabled through probing is minimal in the sense that *only* those receivers with loss rates that are "worse" than the loss rate predicted by the sender are required to send NACKs. If no such receivers exist, then *no* feedback is generated in response to a probe.

3.2 ARM Protocol Description

Table 1 introduces the notation we adopt throughout this paper to describe our ARM protocol.

Symbol	Meaning
K	The number of original data packets
K'	The minimum number of packets that a receiver must receive
N	The total number of packets transmitted, which includes original and redundant packets
$maxseqno$	The maximum sequence number of data to be transmitted
$seqno$	The sequence number when the probe is sent
PTS	The number of packets yet-to-be-transmitted at the time a probe is sent
RPC	The Received Packet Counter denotes the number of packets received thus far
sf	The Stretch Factor $sf = N/K$
d	The value used to decrement $maxseqno$ when no NACKs are received in response to a probe
RTT	The maximum Round-Trip Time between the sender and a receiver
$relax$	The excess packets multicast by a sender beyond $maxseqno$

Table 1: Notation used in our ARM protocol description

We describe the details of ARM by presenting the steps undertaken by the Sender and Receiver(s) at various stages of the protocol. Figure 2 shows the pseudo-code for ARM sender and receiver agents.

Sender: Start

SS.1 Sender sets $maxseqno$ to K' before transmitting the first packet.

SS.2 Sender starts to transfer the original data packets.

SS.3 Concurrently with step SS.2,[3] the sender applies Tornado coding to the original K data packets to obtain N packets (where $N = sf \times K$). These packets constitute the original K packets and the $N - K$ additional *repair* packets ($K < N$).

Sender: Probing

SP.1 Periodically, the sender transmits a probe piggy-backed on a data packet. The probe consists of a time-stamp that identifies the time at which the probe is sent and PTS. Namely, $PTS = maxseqno - seqno$, where $seqno$ is the sequence number of the packet transmitted with the probe.

[3] We should emphasize that our protocol overlaps data transmission with encoding, hence dramatically reducing latency for all receivers to receive the number of packets needed for recovering the original data.

Receiver: Packet Processing

RP.1 Whenever a receiver receives a packet, it increases the Received Packet Counter (RPC) by one to keep track of the number of packets received.

RP.2 If RPC is greater than or equal to K', the original data can be reconstructed from the packets received so far. The receiver then decodes the received data and leaves the multicast group.[4]

RP.3 If RPC is less than K', then the receiver proceeds as follows:

RP.3.1 Compute the loss rate r experienced so far (up to probing time), where r is estimated based on the proportion of packets received.

RP.3.2 Compute m, the number of packets expected to be received given the current $maxseqno$ at the sender and the expected loss rate, r, where $m = PTS \times (1 - r) + RPC$

RP.3.3 If the value of m (computed in RP.3.2) is greater than or equal to K', the receiver does not respond to the probe (i.e. it does not send a NACK).

RP.3.4 If the value of m (computed in RP.3.2) is less than K', then the forthcoming packets are not enough to recover the original data. The receiver proceeds as follows.

RP.3.4.1 A new $maxseqno$ is computed so that the value of m (computed in RP.3.2) is greater than or equal to K'. This leads to the following inequality: $maxseqno \geq seqno + (K' - RPC)/(1-r)$, where $(K' - RPC)$ is the number of additional packets that a receiver needs to receive. Thus, at least $(K' - RPC)/(1 - r)$ additional packets should be sent in order to endure packet losses at the currently estimated loss rate of r.

RP.3.4.2 The receiver sends a NACK that includes the new lower bound on $maxseqno$ calculated in step RP.3.4.1

Sender: NACK Processing

SN.1 If a NACK does not arrive after a delay of $2 \times RTT$ following the transmission of a probe, the sender decrements $maxseqno$ by d since it is safe to assume that the current level of redundancy is higher than warranted by the loss conditions experienced by receivers.

SN.2 Upon receipt of a NACK, the sender updates $maxseqno$ as requested by the receiver.

SN.3 If the new $maxseqno$ requested by a receiver is greater than N, the sender needs to encode more repair packets, and the new value of N becomes: $N = maxseqno + relax$. This makes it possible to adjust the *stretch factor* in the middle of a multicast transmission.[5]

Sender: End

SE.1 The sender transmits packets up to $maxseqno + relax$, instead of $maxseqno$ specified by the feedback from receivers. This is to make the protocol more resilient to unexpected loss rates experienced by the last transmitted packets. By sending these additional $relax$ packets preceded with a probe, a receiver experiencing more losses than expected has the chance to inform the sender of a new larger $maxseqno$ to make up for these unexpected packet losses.

SE.2 The transmission ends once the sender transmits all $maxseqno + relax$ packets and all receivers leave the multicast group after each receiving at least K' packets.

[4] By allowing receivers to leave the multicast group once they receive the K' packets needed, we significantly reduce the bandwidth consumed over the network.

[5] Note that the *fixed* stretch factor of other FEC-based protocols results in the receiver requiring more packets to recover from losses. This is so because under a fixed stretch factor, once repair packets are exhausted, the sender loops back over the old packets that it had earlier transmitted. Thus, the receiver may receive duplicate (useless) packets, causing a waste in network bandwidth and an increase in transmission delays.

```
initialize maxseqno ← K';
while seqno ≤ (maxseqno + relax)
    transmit packet;
    encode data into N packets;

    if seqno = next probe then
        if NACK has not arrived during last 2 × RTT then
            maxseqno ← maxseqno − 1;
        PTS ← maxseqno − seqno;
        send probe with data;
    else
        send data;

    if NACK with new maxseqno arrives then
        update maxseqno;
    if maxseqno > N then
        encode (maxseqno − N) more packets;
        update N ← maxseqno + relax
```

(a) Sender algorithm

```
initialize RPC ← 0;
if data packet is received then
    RPC ← RPC + 1;

    if RPC < K' then
        if probe arrived then
            compute loss rate r;
            if PTS × (1 − r) + RPC < K' then
                maxseqno ← seqno + (K' − RPC)/(1 − r);
                send NACK with maxseqno;
            else
                do nothing;

    else /* RPC ≥ K' */
        leave multicast group;
        decode the received packets
```

(b) Receiver algorithm

Figure 2: Pseudo Code for the ARM protocol

3.3 ARM Probing and RTT Estimation

During data transmission, the sender transmits probes periodically. A probe packet includes the number of packets to be sent (PTS) and time-stamp for when the packet is sent. The purpose of using probes is two-fold: First, a probe is used to trigger NACKs from receivers. Upon receiving the probe containing PTS, a receiver makes a local decision whether this is enough to sustain its current loss rate as we described earlier. Second, a probe is used to estimate the round-trip time (RTT). When a receiver responds with a NACK, it sends the time-stamp for when the NACK is sent. The time-stamps are used to calculate RTT in the same manner as in [14]: the sender sends a probe at time t_1, and a receiver receives the probe at time t_2. If the receiver sends a NACK at time t_3, it includes (t_1, Δ), where $\Delta = t_3 - t_2$. Once the sender receives the NACK at time t_4, it computes RTT as $RTT = t_4 - t_1 - \Delta$.

In other protocols such as SRM [1] and SHARQFEC [10], the estimation of RTT is very critical for NACK suppression and repair. However, in ARM, the RTT value is not critical. The sender only needs an estimate of the maximum RTT from receivers to set the period of probing. As more data is transmitted, the feedback from receivers about their losses in response to probes becomes more critical so that the sender can determine if more repair packets should be encoded. Therefore, it is desirable to set the period of probing to be large at first, and then gradually decrease it. However, the period of probing should be at least equal to RTT to avoid sending duplicate probes.

4 Performance Evaluation

In this section we present the results of our prototype implementation and performance evaluation of ARM.

4.1 Simulated Protocols

We evaluated the performance of our ARM protocol by comparing it to the well-known SRM protocol of Floyd et al [1].

ARM ns Prototype Implementation: We prototyped an implementation of our ARM protocol using the UCB/LBNL/VINT network simulator, ns-2.1b4 [12]. A new agent, called ARM, is created as a subclass of AgentClass and defined in `arm.cc` and `arm.h`. This agent implements ARM for reliable multicast. The sender starts transmitting data at time 3.0. The simulation run is stopped once *all* receivers receive the needed packets to recover the original data.

As mentioned in Section 3, ARM makes use of Tornado coding to dynamically encode more repair packets based on estimated loss rates experienced by receivers. With Tornado, a receiver must receive at least $K' = (1+\epsilon)K$, where K is the number of original data packets and ϵ is the reception overhead. In [6], ϵ was found to be very small. In our simulations, we take $\epsilon = 0.03$. The stretch factor in our ns prototype implementation of ARM is set to 2 at first. It is increased (without waiting) if need be as described in step SN.3 of Section 3. To estimate the loss rate r experienced by a receiver, we assume the receiver calculates r as $1-(RPC/seqno)$, where RPC is the number of packets received so far, and $seqno$ is the sequence number of the packet carrying the probe, thus $RPC/seqno$ is the proportion of packets successfully received. Regarding the period of probing, as mentioned in Section 3.3, more frequent probing is needed toward the end of the data transmission so as to trigger NACKs and encode more repair packets if needed. In our simulations, we send the first probe after sending the first $K/5$ packets, then we increase the probing frequency by sending one probe every RTT. In order to account for scenarios where the last transmitted packets experience unexpected losses, ARM transmits a number $relax$ of additional packets as described in step SE.1 of Section 3. In this paper, we did not consider such simulation scenarios, and so the value of $relax$ is zero. For the additive decrease of $maxseqno$ in step SN.1 in Section 3, we take d to be 1.

SRM ns Prototype Implementation: We used the SRM implementation of ns version 2.1b4. The code is modified to stop the simulation once all receivers receive the K packets, instead of stopping at a predefined simulation time. SRM senders start sending session messages at time 1.0 and start sending data at time 3.0. Session messages are sent periodically, so receivers can estimate RTT.

In our experiments, we didn't account for the Tornado encoding and decoding times and we did not account for the delays resulting from SRM's need to send session messages periodically to estimate RTT. As discussed in Section 3, in ARM, encoding is overlapped with packet transmission. Also, decoding time at the client is relatively fast [6].[6] In particular, ARM decoding time is more than offset by SRM's need to send session messages to estimate RTTs—an overhead that we did not take into consideration either.

4.2 Simulation Model and Metrics

To evaluate the performance of ARM and SRM we set up a simulated multicast network using the 15-node tree topology depicted in Figure 3. In this topology, a CBR (Constant Bit Rate) data source is attached to node 14 and all other nodes (i.e. nodes 0 to 13) act as receivers. In our simulations, the packet interarrival time for the CBR source is set to 0.01 seconds. In Figure 3, link labels represent the average loss rate induced in our simulations. Losses were induced uniformly and independently on a per link basis. The bandwidth of the links in our simulated topology are set to 1.5Mbps. The link delays are set to be 500ms

[6] In [6], Byers, Luby, and Mitzenmacher showed that the decoding time is small compared to transmission time—it was 1.75 seconds for 16MB data.

for link (14,12), 100ms for link (12,13), 400ms for links (12,9), (12,10), and (12,11), and 100ms for the rest of links. The packet size is 1KB. So, for example, if K is 1000, the data size is 1MB. We assume that the receivers join the multicast group before starting data transmission.

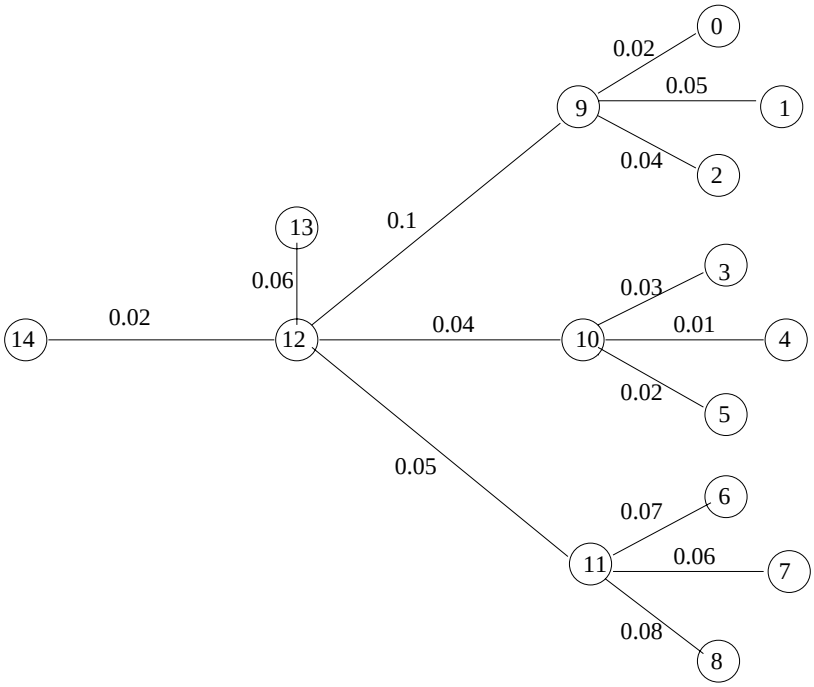

Figure 3: The network topology used in our simulations

In our simulations, we measured three performance metrics. The first is the *transmission time*, which is defined as the time it takes from the start of the multicast transmission until *all* receivers are able to reconstruct the original data transmitted. The second is the *total number of packets injected by the sender into the network*. This metric allows us to evaluate the *goodput*, which is the ratio of the packets needed to the packets actually sent. The third is the *request traffic*, i.e., the number of NACKs emitted from the receivers to the sender.

4.3 Simulation Results

Transmission Time: In Figure 4, we compare the transmission time of ARM against that of SRM. As expected, ARM complete its transmission faster than SRM. The advantage of ARM is especially evident when the number of packets to be communicated is relatively small. For example, the transmission delay for $K = 600$ packets is cut by almost 57% (from 20 to 8.5 seconds) by using ARM as opposed to SRM. In ARM, receivers do not recover their lost packets by waiting for retransmissions as in SRM. Rather, receivers can recover their losses as they receive "fresh" repair packets from the sender. The relative reduction in transmission time is more pronounced when the number of packets to be communicated (i.e. K) is small.

Bandwidth Utilization: Figure 5 shows the total number of packets transmitted by the sender (on the Y-axis) to complete a K-packet reliable multicast transmission (on the X-axis). Under SRM, the total number of transmitted packets is the number of original and retransmitted data packets from the sender, as well as the repair packets sent by receivers. Under ARM, it is the total number of packets sent from the sender (including original and repair packets). The figure shows that ARM consistently reduces the total number of packets and thus has a better *goodput* than that of SRM.

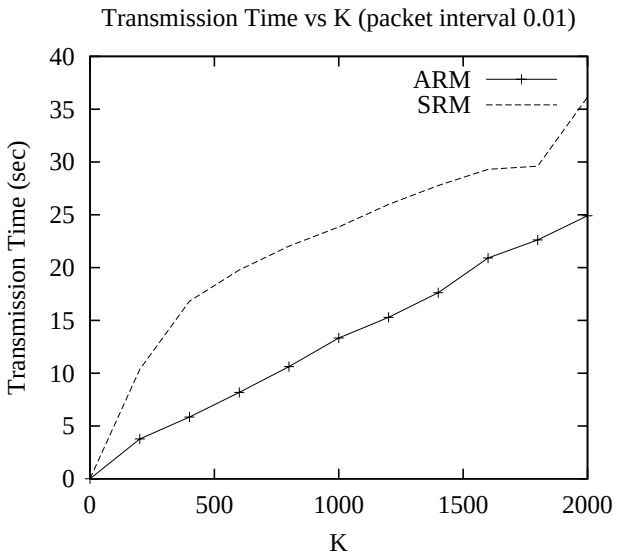

Figure 4: Transmission time vs number of original packets

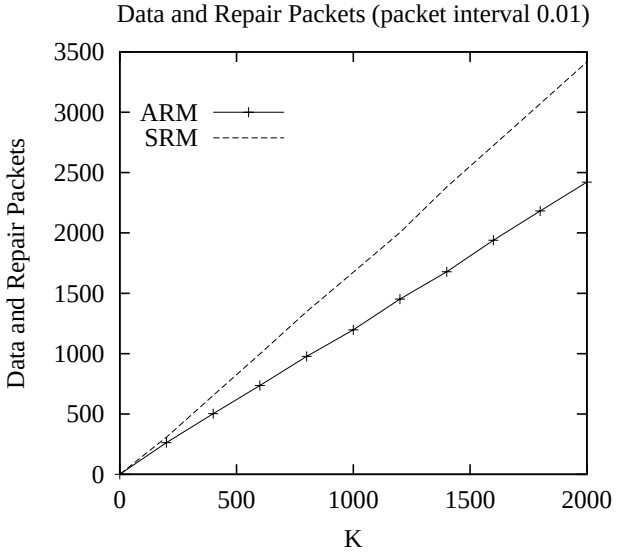

Figure 5: Total number of transmitted data packets vs number of original packets

ARM transmits enough packets to compensate for the worst-case loss among all receivers—i.e. it transmits $K/(1-R)$ packets, where R is the worst-case loss among all receivers. The expected steady-state total loss between the source and a receiver is calculated as follows:

$$\text{Total Loss } = 1 - \prod Pr(\text{No loss for all links from the sender to the receiver}) \quad (1)$$

Node 1 in Figure 3 has the maximum loss probability. Using Equation (1), the expected loss rate for node 1 is 0.1621. This means that the sender should transmit at least $K/(1-0.1621)$ to meet the receiver's requirement. Indeed, our simulations show that ARM transmits approximately this number of packets, resulting in a *goodput* of approximately 84% (versus 58% for SRM).[7]

Request (NACK) Traffic: Figure 6 shows the number of NACK packets handled by the sender as a function of the number of original packets transmitted (i.e. K). Compared to SRM, ARM results in significantly fewer NACKs. Moreover, under ARM, NACK traffic levels off (as opposed to linearly increasing) as the value of K increases.

Under ARM, NACK traffic is larger when K is small. This can be explained by noting that ARM starts probing after an initial preset number of packets (namely, $K/5$ in our simulations)—independent of the maximum RTT between the sender and receivers. When K is small, this initial probing interval may be less than the actual maximum RTT, resulting in an "eager" probing behavior, which induces an (unnecessary) increase in NACK traffic. Figure 6 shows that once K is large enough (and hence $K/5$ becomes of the same magnitude as RTT), this "eager" probing behavior disappears, resulting in a *constant* NACK traffic—independent of K.[8]

Adaptation to Network Conditions: In Figure 7, we studied how the value of *maxseqno* changes over time. As explained in steps SN.1-3 of the ARM protocol described in Section 3, the value of *maxseqno*

[7] To appreciate the advantage of the adaptive nature of FEC used in ARM, it suffices to note that a pure FEC protocol (using a stretch factor of 2, i.e. $N = 2K$) would result in the transmission of all N packets—a goodput of 50%.

[8] The estimation of RTT in ARM is not crucial. However, excess NACK traffic resulting from "eager" probing could be eliminated if the maximum RTT is estimated *a priori*.

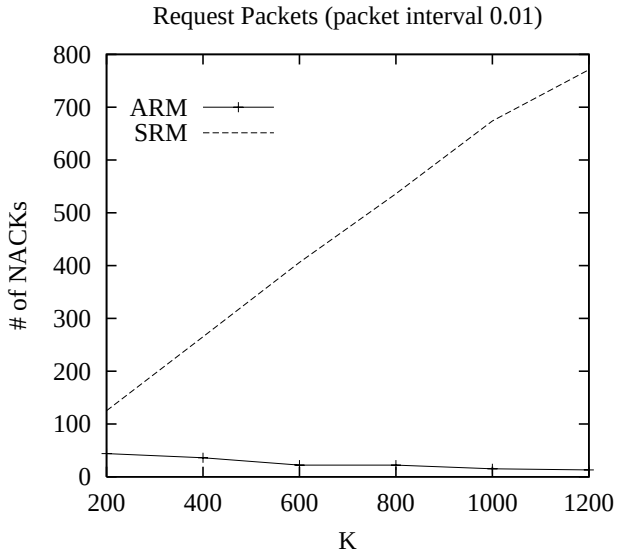

Figure 6: Number of request packets (NACKs) vs number of original packets.

is increased based on the NACK feedback from receivers. It is decreased (additively) for every probe for which no NACKs were received within two RTTs. This process ensures that *maxseqno* adjusts dynamically to the actual loss rates experienced by receivers. For the static loss rates induced in our simulations (as shown in Figure 3), Figure 7 (left) shows that *maxseqno* quickly approaches its quiescent value; Figure 7 (right) shows the "sawtooth" like behavior of *maxseqno* when we zoom in on a small segment of the multicast transmission.

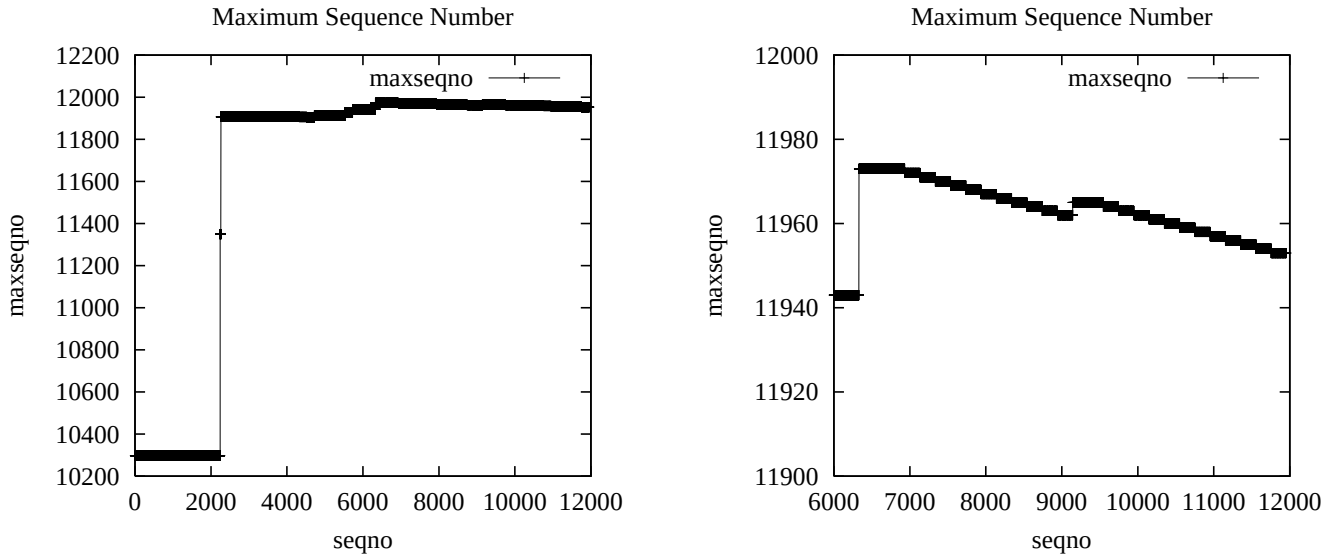

Figure 7: Adaptation of maxseqno over entire multicast session (left) and over a segment thereof (right).

The constant NACK traffic under ARM depicted in Figure 6 and the rather smooth value of *maxseqno* depicted in Figure 7 (left) are consistent with the loss model adopted in our simulations. In particular, since the loss rate on the various links of our simulated topology is *static*, it follows that a constant number of NACKs will be enough to adjust the level of redundancy (and hence *maxseqno*) used by the sender—

independent of the length of the multicast transmission. Under a more dynamic loss model, this will not be the case. In particular, NACK traffic will be proportional to the length of the multicast transmission—or more accurately to the variability of network loss characteristics throughout the multicast session. Similarly, under a more dynamic loss model, *maxseqno* is likely to be more variable. Such increase in NACK traffic and the burstiness of *maxseqno* are tightly related to the variability in network loss characteristics. By tuning the various parameters of ARM, such variability could be effectively controlled using more dynamic loss rate estimation techniques (e.g. based on a sliding-window approach) and using a more dynamic technique for adjustment of redundancy.

5 Conclusion and Future Work

In this paper we have proposed and evaluated a hybrid ARQ-FEC Adaptive Reliable Multicast (ARM) protocol, which uses *minimal* feedback from receivers to dynamically adjust the amount of redundant data that the sender must transmit to ensure a reliable delivery of multicast data to all receivers. In particular, an ARM sender employs a probing mechanism to solicit feedback from *only* those receivers experiencing a loss rate that cannot be accomodated given the current level of redundancy adopted by the sender. Such feedback (or lack thereof) is used by an ARM sender to select (or readjust) "on the fly" the level of redundancy to be used to mask packet losses. Our preliminary evaluation of ARM suggests that it promises shorter transmission times, decreased NACK traffic, and improved bandwidth utilization (or goodput) when compared to the well-known SRM reliable multicast protocol.

We are currently investigating a number of issues that were not addressed in this paper. In particular, in the simulations of Section 4, we adopted (1) a simple "static" model of packet losses, whereby packet losses on a given link are independent and constant throughout a multicast transmission, and (2) a simple "static" model for RTTs, whereby the RTT is constant for a given receiver throughout a multicast transmission. Both of these assumptions are not realistic. We are currently investigating the robustness of ARM when faced with highly-variable network conditions.

In this paper we assumed that the data to be multicast is available at the sender *a priori* and that the delivery of such data is not subject to *real-time* constraints. While these assumptions apply to many applications of reliable multicast (e.g. on-line auctions), they are not adequate for others (e.g. live stock-market feeds). In the latter class of applications, an important question is the determination of an appropriate level of buffering to use for FEC on live/real-time feeds.

Acknowledgments: We would like to thank John Byers for the many discussions on Tornado codes and Digital Fountains. This work was supported in part by NSF research grants ESS CCR-9706685, CAREER ANIR-9701988, and MRI EIA-9871022.

References

[1] S. Floyd, V. Jacobson, L. Ching-Gung, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", *Proceedings of ACM SIGCOMM '95*, pp. 342-356, Aug. 1995.

[2] John C. Lin and S. Paul, "RMTP: A Reliable Multicast Transport Protocol", *IEEE INFOCOM '96*, March 1996, pp. 1414-1424.

[3] R. Yavatkar, J. Griffioen, and M. Sudan, "A Reliable Dissemination Protocol for Interactive Collaborative Applications," *Proceedings of the ACM Multimedia '95 Conference*, November 1995.

[4] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, "Practical Loss Resilient Codes", *Proceedings of the* 29^{th} *ACM Symposium on Theory of Computing*, 1997.

[5] M. Luby, M. Mitzenmacher, A. Shokrollahi, "Analysis of Random Processes via And-Or Tree Evaluation", *Proceedings of the* 9^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, January, 1998.

[6] J. Byers, Luby, and Mitzenmacher, "A Digital Fountain Approach to Reliable Distribution of Bulk Data (Tornado)", *Proceedings of ACM SIGCOMM '98*, Vancouver, September 1998.

[7] S. Deering, "Multicast routing in a datagram internetwork," Tech. Rep. No. STAN-CS-92-1415, Stanford University, California, Dec. 1991.

[8] J. Nonnenmacher, E. Biersack, D. Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission", *Computer Communications Review ACM SIGCOMM*, vol. 27, No. 4, 1997.

[9] D. Rubenstein, J. Kurose, D. Towsley, "Real-Time Reliable Multicast Using Proactive Forward Error Correction", *NOSSDAV '98*, Cambridge, UK, July, 1998.

[10] R. Kermode, "Scoped Hybrid Automatic Repeat Request with Forward Error Correction (SHARQFEC)", *ACM SIGCOMM 98*, September 1998, Vancouver, Canada.

[11] Request For Comments, RFC 2117, "Protocol Independent Multicast - Sparse Mode (PIM-SM) : Protocol Specification", June 1997.

[12] UCB/LBNL/VINT Network Simulator, ns, URL: *http://www-mash.cs.brekeley.edu/ns*.

[13] A. McAuley, "Reliable Broadband Communication Using a Burst Erasure Correcting Code," *ACM SIGCOMM '90*, Sep. 1990, Philadelphia, pp. 297-306.

[14] D. Mills, "Network Time Protocol (version 3)" Request For Comments, RFC 1305, March 1992.

[15] C. Huitema, "The case for packet level FEC", *Proceedings of IFIP 5th International Workshop on Protocols for High Speed Networks (PfHSN'96)*, France, October, 1995.

[16] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols", *ACM Computer Communications Review*, vol. 27, n.2, Apr. 1997, pp. 24-36.

[17] W. Strayer, B. Dempsey, and A. Weaver, "XTP: The Xpress Transfer Protocol", Addison-Wesley, URL: *http://hegschool.aw.com/cseng/authors/dempsey/xtp/xtp.nclk*.

[18] L. Rizzo and L. Vicisano, "A Reliable Multicast data Distribution Protocol based on software FEC techniques", *Proceedings of the Fourth IEEE, HPCS'97 Workshop*, Chalkidiki, Greece, June 1997.

[19] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[20] Azer Bestavros. An Adaptive Information Dispersal Algorithm for Time-critical Reliable Communication. In Ivan Frisch, Manu Malek, and Shivendra Panwar, editors, *Network Management and Control, Volume II*, chapter 6, pages 423–438. Plenum Publishing Corporation, New York, New York, 1994.

[21] Azer Bestavros and Gitae Kim. TCP Boston: A Fragmentation-tolerant TCP Protocol for ATM Networks. In *Proceedings of Infocom'97: The IEEE International Conference on Computer Communication*, Kobe, Japan, April 1997.