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

Abstract

We present 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.

1 Introduction

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

The Internet implements a multicast service that is unreliable best-effort [2]. 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) [3] is one of the most popular ARQ based protocols that have been proposed to reduce this NACK implosion. The basic idea is to have a receiver multicast NACK 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.

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. 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 (e.g. [8, 4]). 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 in-

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

dependent 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 the paper is organized as follows. Section 2 describes our proposed ARM protocol. Section 3 presents our simulation results. Section 4 concludes the paper. Due to space limitations, we refer the reader to [9] for detailed discussion of related work and ARM description.

2 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 protocol.

2.1 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 [5].[1] With Tornado coding, a receiver must receive at least $K' = (1+\epsilon)K$, where K is the number of original data packets and ϵ is the reception (decoding) overhead.[2] In [1], ϵ was found to be very small.

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 as follows: first, it is used 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. Second, by adding PTS to the current sequence number of the probe packet, a receiver calculates the current *maxseqno* of the sender. If it is the same as the receiver's *maxseqno* previously calculated and sent back to the sender in a NACK, the receiver considers itself the bottleneck as it is the one that had set the *maxseqno* of the sender. Then, the receiver responds to the probe with a NACK containing the updated *maxseqno*, which can be less or greater than the current value of the sender.

Upon receiving a NACK from a receiver, the sender adjusts its maximum sequence number to accommodate

[1] 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 [7]).

[2] In our simulations, we take $\epsilon = 0.03$.

the needs of that receiver. Future probes multicast by the sender will reflect this adjusted 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 except only by the bottleneck to decrease the $maxseqno$.

2.2 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.

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, the sender applies Tornado coding to the original K data packets to obtain N packets.[3] 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 piggybacked 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. The purpose of the timestamp is to estimate the maximum round-trip time (RTT).[4]

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.[5]

RP.3 If RPC is less than K' and a probe is received from the sender, then the receiver proceeds as follows:

RP.3.1 Compute m, the number of packets expected to be received, as follows: $m = PTS \times (1-r) + RPC$, where r is the expected loss rate computed as in RE.2.

RP.3.2 If the value of m (computed in RP.3.1) is greater than or equal to K', the receiver does not respond to the probe (i.e. it does not send a NACK).

RP.3.3 If the value of m (computed in RP.3.1) is less than K', then the forthcoming packets are not enough to recover the original data. The receiver proceeds as in RP.3.5 and RP.3.6.

RP.3.4 The receiver decides whether it is the bottleneck, i.e. the one that had set the current value of $maxseqno$ of the sender. This is so if $maxseqno$ that it had sent to the sender in a previous NACK, equals the current sequence number plus PTS. In case the receiver is the bottleneck, it proceeds as in RP.3.5 and RP.3.6.

RP.3.5 A new $maxseqno$ is computed as follows:

$$maxseqno = seqno + (K' - RPC)/(1-r)$$

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.3.6 The receiver sends a NACK that includes the new $maxseqno$ calculated in step RP.3.5.

Receiver: Loss Rate Estimation

RE.1 Periodically, a receiver updates its current estimate of loss rate l as follows:

$$l = 1 - (\Delta RPC / \Delta seqno)$$

where $\Delta seqno$ is the difference in sequence numbers of packets received at the beginning and end of the update time interval. ΔRPC is the number of packets received during the update interval. Thus, the ratio l gives the current proportion of packets lost.

RE.2 Based on l, a receiver maintains exponential moving average and deviation of the loss rate. Specifically,

$$\begin{aligned} AvgL &= \alpha \times AvgL + (1-\alpha) \times l \\ DevL &= (1-\delta) \times \mid l - AvgL \mid + \delta \times DevL \\ r &= AvgL + \gamma \times DevL \end{aligned}$$

where r is the estimated loss rate. $AvgL$ and $DevL$ are the moving average and deviation, respectively.[6]

Sender: NACK Processing

SN.1 Upon receipt of a NACK, the sender updates $maxseqno$ as the maximum value among $maxseqno$ returned by receivers in response to the same probe.

SN.2 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: $N = maxseqno$. This makes it possible to adjust the level of redundancy in the middle of a multicast transmission.

[3] In our simulations, we take $N = 2K$ at first. It is increased (without waiting) if need be as described in step SN.2.

[4] 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 uses the timeout mechanism described in step SE.1 until every receiver receives K' packets.

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

[6] In our experiments, we take $\alpha = \delta = 0.5$, and we set γ to 3. γ could be set to higher values to account for high variability in loss conditions. We take the estimation update interval to be 0.2 seconds.

SENDER: END

SE.1 After transmitting all *maxseqno* packets, the sender sets a timer with *RTT*. If a NACK is received before the timer expires, the sender resets the timer. If the timer expires before every receiver receives K' packets, the sender increases *maxseqno* by the number of packets per *RTT*, which is calculated as $RTT \times packet\ sending\ rate$.

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

3 Performance Evaluation

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

Simulated Protocols: We evaluated the performance of our ARM protocol by comparing it to the well-known SRM protocol of Floyd *et al*. [3].

We prototyped an implementation of our ARM protocol using the UCB/LBNL/VINT network simulator, ns-2.1b4 [6]. 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 25.0. The simulation run is stopped once *all* receivers receive the needed packets to recover the original data.

We used the SRM implementation of ns version 2.1b4. The code is modified to stop the simulation once every receiver receives K packets, instead of stopping at a predefined simulation time. SRM senders start sending session messages at time 20.0 and start sending data at time 25.0. Session messages are sent periodically, so receivers can estimate RTT [3].

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.

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 1. 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. Each link in the network is subjected to a maximum of 32 on-off cross connections generated by a UDP-based agent. This UDP-based agent generates connections with an inter-arrival time uniformly distributed between 0 and 0.1 second. Each connection is an on-off source with Pareto distributed "on" and "off" periods with average durations of 0.1 second and 0.9 second, respectively. The Pareto distribution has a skew parameter of 1.35. During the "on" periods, packets are generated at a rate of 1000Kbps. This cross-traffic resulted in up to 30% loss rates observed at receivers. The bandwidth of the links in our simulated topology are set to 1.5Mbps. All links have a propagation delay of 15ms. 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.

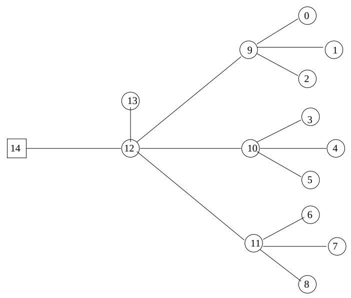


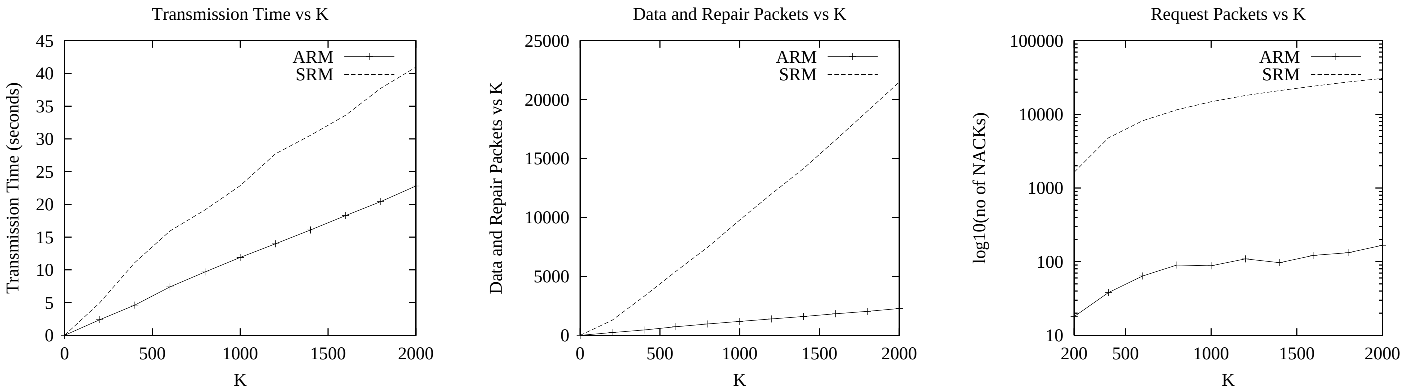
Figure 1. Simulated Network Topology

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.

Simulation Results: In Figure 2(a), we compare the transmission time of ARM against that of SRM. As expected, ARM completes its transmission faster than SRM. The transmission delay is cut by almost 50% 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.

Figure 2(b) 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. 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.

Figure 2(c) shows the number of NACK packets handled by the sender as a function of the number of original packets transmitted (i.e. K). The NACK traffic depends on the loss model. 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. Compared to SRM, ARM results in significantly fewer NACKs under the dynamic cross-traffic loss model used in our experiments.



(a) (b) (c)

Figure 2. Performance vs. Number of Original Packets: (a) Transmission Time, (b) Total Number of Transmitted Data Packets, (c) Number of Request Packets (NACKs)

In Figure 3, we studied how the value of *maxseqno* of the sender changes over time. As explained in step SN.1 of the ARM protocol described in Section 2.2, the value of *maxseqno* is increased based on the NACK feedback from receivers. It might also be decreased based on the NACK feedback from bottleneck nodes as described in step RP.3.4. This process ensures that *maxseqno* adjusts dynamically to the actual loss rates experienced by receivers, which are estimated as in RE.1 and RE.2.

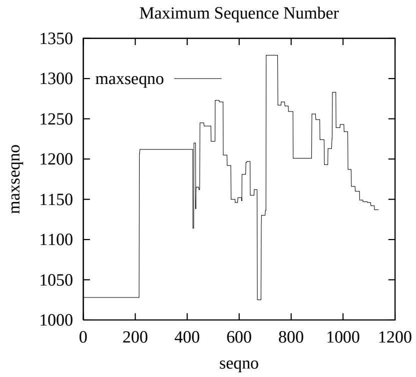


Figure 3. Adaptation of maxseqno.

4 Conclusion

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 accommodated 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 have recently proposed SomeCast, a paradigm that extends ARM to support reliable multicast delivery subject to *real-time* constraints [10].

Acknowledgment: We would like to thank John Byers for the many discussions on Tornado codes and Digital Fountains.

References

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

[2] S. Deering. Multicast routing in a datagram internetwork. Tech. Rep. No. STAN-CS-92-1415, Stanford Univ., CA, Dec. 1991.

[3] S. Floyd, V. Jacobson, L. Ching-Gung, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *Proc. ACM SIGCOMM '95*, Aug. 1995.

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

[5] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman. Practical Loss Resilient Codes. *Proc. 29th ACM Symposium on Theory of Computing*, 1997.

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

[7] 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, Apr. 1989.

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

[9] J. Yoon, A. Bestavros, and I. Matta. Adaptive Reliable Multicast. Tech. Rep. No. BU-CS-1999-012, Sep. 1999.

[10] J. Yoon, A. Bestavros, and I. Matta. SomeCast: A Paradigm for Real-Time Adaptive Reliable Multicast. *To appear in IEEE Real-Time Technology and Applications Symposium (RTAS) 2000*, Washington D.C., Jun. 2000.