The Case For Reliable Concurrent Multicasting Using Shared Ack Trees

Brian Neil Levine David B. Lavo J.J. Garcia-Luna-Aceves
{brian, lavo, jj}@cse.ucsc.edu
Department of Computer Engineering
University of California
Santa Cruz, CA 95064

ABSTRACT

Such interactive, distributed multimedia applications as shared whiteboards, group editors, and simulations require reliable concurrent multicast services, i.e., the reliable dissemination of information from multiple sources to all the members of a group. Furthermore, it makes sense to offer that service on top of the increasingly available IP multicast service, which offers unreliable multicasting. This paper establishes that concurrent reliable multicasting over the Internet should be based on reliable multicast protocols based on a shared acknowledgment tree. First, we show that organizing the receivers of a reliable multicast group into an acknowledgment tree and using NAK*-avoidance with periodic polling in local groups inside such a tree provides the highest maximum throughput among all classes of reliable multicast protocols proposed to date. Second, we introduce* Lorax*, which demonstrates the viability of implementing a reliable multicasting approach in the Internet based on acknowledgment trees in a scalable manner. Lorax is the first known protocol that constructs and maintains a single acknowledgment tree for reliable concurrent multicasting, eliminates the need to maintain an acknowledgment tree for each source of a reliable multicast group, and can be used in combination with any of several tree-based reliable multicast protocols proposed to date.*

Keywords: Reliable Concurrent Multicast, Performance Evaluation, Transport Protocols, Collaboration, Internet

1 INTRODUCTION

Interactive, distributed multimedia applications like shared whiteboards, group editors, and simulations require a *reliable concurrent multicasting* service. Such a service consists of disseminating information from multiple sources to all members of a multicast group, such that (a) every packet from each source is delivered to each receiver within a finite time, free of errors, with no duplicates, and in the order sent by the source; and (b) nodes responsible for retransmitting packets can delete packets from memory within a finite time.

The development and implementation of end-to-end protocols for reliable concurrent multicasting over the Internet is being enabled by the increasing availability of multicast routing in Internet routers. IP-Multicast routers permit sources to transmit data unreliably to multiple receivers [1]. The most critical challenge for the successful development and implementation of end-to-end reliable protocols built on top of IP multicast consists of avoiding the *acknowledgment (*ACK*) implosion* problem in large multicast groups: in a very large reliable multicast session, the sources may be overwhelmed by the amount of work required to process the acknowledgments sent by the large receiver set.

A considerable amount of work has been reported in the recent past on how to cope with or eliminate the ACK-implosion problem [2]—[16]. However, the design of reliable multicast protocols is complex and there is no consensus yet on which is the best approach for the implementation of protocols for scalable, reliable concurrent multicasting over the Internet. This paper makes the case that end-to-end reliable concurrent multicasting over the Internet should be based on protocols based on a shared acknowledgment tree. We establish our case in three parts.

First, in Section 2, we summarize the known classes of protocols that have been proposed for end-to-end reliable multicasting. In Section 3, we use this taxonomy and an approximate model to analyze the maximum throughput of these protocol classes. Our analysis shows that the *tree-NAPP protocol* class is the most scalable approach with respect to the number of receivers and provides the highest maximum throughput among all reliable multicast protocol classes proposed to date.[1] In a tree-NAPP protocol, the receivers of a reliable multicast group are organized into an acknowledgment tree (ACK tree) built on top of the multicast routing tree(s) provided by such multicast routing protocols as DVMRP [17], PIM [18], CBT [19], or OCBT [20]. A source multicasts packets to all the receivers through the multicast routing tree, and responsibility for retransmissions is delegated to the receivers. Retransmissions take place only in local groups of the ACK tree, and the number of ACK traffic within each local group is reduced by means of NAK-avoidance with periodic polling.

Second, Section 4 presents a simple extension of any ACK tree-based reliable multicast protocol. This extension allows the source to safely deallocate packets from memory when the ACK tree needs to be modified.

[1]These results are consistent with the experimental results reported in [6].

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **1996**	2. REPORT TYPE	3. DATES COVERED **00-00-1996 to 00-00-1996**
4. TITLE AND SUBTITLE **The Case for Reliable Concurrent Multicasting Using Shared Ack Trees**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**12**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Finally, we note that it is not reasonable to set up an ACK tree for every source in a concurrent multicast session, and that the ACK tree should adapt to changes in the constituency of either the receiver set or the multicast routing tree(s). Section 5 completes our case by describing Lorax ([21]), which is the first known protocol that constructs and maintains a single shared ACK tree for reliable concurrent multicasting. Lorax eliminates the need to maintain an ACK tree for each source of a reliable multicast group, and can be used in combination with any of several tree-based reliable multicast protocols proposed to date (e.g., [5, 6]).

Section 6 compares our approach with related work and discusses why Lorax and tree-NAPP protocols are the best approach to date for the provision of scalable, reliable concurrent multicasting services in the Internet. Conclusions and directions for future work are offered in Section 8.

2 BACKGROUND

To provide a summary of known classes of reliable multicast protocols, we use a taxonomy that decouples the definition of the mechanisms needed for the pacing of data transmission from the mechanisms needed for the allocation of memory at the source [10]. Each protocol class can be viewed as using two windows: a congestion window (*cw*) that advances based on feedback from receivers regarding the pacing of transmissions and detection of errors, and a memory allocation window (*mw*) that advances based on feedback from receivers as to whether the sender can erase data from memory. In practice, protocols may use a single window for pacing and memory allocation (e.g., TCP [22]) or separate windows (e.g., NETBLT [23]). In all classes, packets are multicast unreliably from the source directly to all receivers. The protocol classes differ on how acknowledgments flow from the receivers back to the source. A more detailed description of all generic protocols can be found in [10].

2.1 Sender-Initiated Protocols

A sender-initiated reliable multicast protocol is one that requires the source to receive ACKs from all members of a known receiver set before it is allowed to release memory for the data associated with the ACKs. The receivers are not organized into any structure, and may contact the source directly. An example of this type of protocols is presented in [13]. It is well known this scheme suffers from the ACK-implosion problem. Whether the source or the receivers are in charge of pacing the source and scheduling retransmissions is unimportant for our taxonomy. In other words, regardless of whether a sender-based or receiver-based retransmission strategy is used, the source is still in charge of deallocating memory after receiving all the ACKs for a given packet or set of packets. The use of NAKs encourages a shortened retransmission latency, but is not necessary for protocol correctness. The main limitation of sender-initiated protocols is not that ACKs are used, but rather the need for the source to process all of the ACKs and to know the receiver set.

2.2 Receiver-Initiated Protocols

The critical aspect of receiver-initiated protocols for our taxonomy is that no ACKs are used. The receivers send NAKs back to the source when a retransmission is needed, detected by either an error, a skip in the sequence numbers used, or a timeout. Because the source receives feedback from receivers only when packets are lost and not when they are delivered, the source is unable to ascertain when it can safely release data from memory. There is no explicit mechanism in a receiver-initiated protocol for the source to release data from memory (i.e., advance the *mw*), even though its pacing and retransmission mechanisms are scalable and efficient (i.e., advancing the *cw*).

Because the source may experience NAK-implosion if many receivers detect transmission errors, previous work on receiver-initiated protocols ([8, 11]) adopts the NAK-avoidance scheme first proposed in [2]: upon detection of a lost packet, receivers schedule a NAK for a random time in the near future. During that time the receiver listens for a NAK by another multicast group member for the same packet. If another NAK is heard, the transmission is scheduled for a subsequent time. It is hoped that only one NAK is sent by the whole group to the parent for a lost packet. We refer to this protocol subclass as RINA (for Receiver-Initiated with NAK-Avoidance). The scalable reliable multicasting (SRM) protocol [11] and the "log-based receiver-reliable multicast" (LBRM) protocol [12] are examples of RINA protocols.

2.3 Ring-Based Protocols

Our generic description of ring-based protocols is based on the Reliable Multicast Protocol (RMP) [7], which is based on the Token Ring Protocol (TRP) [3].

Ring-based protocols work by organizing the receivers into a ring, with a rotating token site designated as the only node to ACK back to the source for the current packet. The source deletes packets only when an ACK/token is received. The ACK also serves to pass the token and to timestamp packets, so that all receiver nodes have a global ordering of the packets for delivery to the application layer. Receivers send NAKs to the token site for selective repeat of lost packets that were originally multicast from the source. Both TRP and RMP specify that retransmissions are sent unicast from the token site. The token is not passed to the next member of the ring of receivers until the new site has correctly received all packets that the former site has received. Once the token is passed, a site may clear packets from memory. We can characterize ring-based protocols as placing the token site in control of the *mw* (conditional on passing the token), and placing control of the *cw* with either the token site or the source.

2.4 Tree-Based Protocols

Tree-based protocols designate three types of nodes over a pre-constructed ACK tree during reliable transmission: source, hop, and leaf nodes. *Source nodes* multicast to the entire receiver set and are responsible to at most B children for retransmissions of data. *Leaf nodes* are strictly receivers and have no children in the ACK tree. *Hop nodes* are intermediate nodes between the source and leaves, responsible for requesting lost data from a parent hop node and for retransmitting data requested by at most B children. A hop node and its children constitute a *local group*, with the hop node as the *group leader*. Note that leaf nodes are essentially hop nodes with no children, and source nodes are hop nodes with no parent.

Because ACKs are sent to the parent node and not the source, we refer to them as hierarchical-acknowledgments (HACKs). In our generic tree-based protocol, a node sends a HACK to its parent as soon it receives a packet correctly (in order to move the *cw*), not when all its own children (if any) have sent their HACKs. If the source had to wait for ACKs to be aggregated all the way from the leaf nodes, it would have to be paced based on the slowest tree path. In the tree-based protocols proposed to date ([4, 5, 6]) the *cw* is advanced by HACKs, but in there is no provision for deleting packets and advancing the *mw* safely. Section 4 addresses this important point in more detail, providing an extension to the tree-based class that allows the source to safely delete packets and advance the *mw*.

We assume that the source and group leaders control the retransmission timeouts; however, such timeouts can be controlled by the children of the source and group leaders. Accordingly, when the source sends a packet, it sets a timer, and each hop node sets a timer as it becomes aware of a new packet. If there is a timeout before all HACKs have been received, the packet is assumed to be lost and is retransmitted by the source or group leader to its children. We assume a selective repeat strategy is used, so that once a packet is received correctly, it is never rebroadcast to the local group again. Several tree-based protocol possibilities are discussed in [4], and have been fully developed as the Reliable Multicast Transport Protocol (RMTP) [5].

2.5 Tree-NAPP Protocols

Tree-NAPP protocols are a subclass of tree-based protocols. The utilization of NAK-avoidance and periodic polling described in [2] by the local groups in a tree-based protocol defines this subclass. NAKS alone are not sufficient to guarantee reliability with finite memory, so receivers send a periodic positive (hierarchical) acknowledgment to their parents so that the *cw* may be advanced. Note that the setting of timers needed for NAK avoidance is done entirely on the local group scale, so it is scalable.

An implementation of tree-NAPPing can be found in the Tree-based Multicast Transport Protocol (TMTP) [6]. One approach to implement NAK-avoidance within a local group of an ACK tree consists of using a multicast address for each local group of the ACK tree.

3 MAXIMUM THROUGHPUT ANALYSIS

To analyze the relative maximum throughput of reliable multicast protocols, we continue to use the same model used in [10] and first introduced in [8], which focuses on the processing requirements of generic reliable multicast protocols, rather than the communication bandwidth requirements. Accordingly, the maximum throughput of a generic protocol is a function of the per-packet processing rate at the sender and receivers, and the analysis focuses on obtaining the processing times per packet at a given node.

We assume a single sender, X, multicasting a constant stream of packets to R identical receivers. For clarity, we assume a single ACK tree rooted at the source. All loss events at any node in the multicast are mutually independent, the probability of packet loss is p for any node, and no ACK is ever lost.

Following the notation in [8] and [10], we place a superscript $H2$ on any variables relating to the generic tree-NAPP protocol. Additional notation and variables are introduced as needed in the analysis; Figure 1 is a complete list of all variables used in this paper for quick reference. The following paragraphs derive the maximum throughput for tree-based protocols with local NAPP; the maximum throughputs for the rest of the classes are derived in [8, 10].

Assuming a finite amount of memory at every node, it is easy to show [10] that the generic sender-initiated, ring-based, and tree-based protocols are free of deadlocks and deliver packets reliably, while RINA protocols incur deadlocks. Table 2 summarizes the results on maximum throughput and correctness reported in [10], together with the tree-NAPP throughput result derived next.

3.1 Throughput of Tree-NAPP Protocol

To bound the overall system throughput in the generic tree-NAPP protocol, we first derive and bound the expected cost at the source, hop, and leaf nodes. To make use of symmetry, we assume, without loss of generality that there are enough receivers to form a full tree at each level.

3.1.1 Source node We consider first X^{H2}, the processing costs required by the source to successfully multicast an arbitrarily chosen packet to all receivers using the $H2$ protocol. The processing requirement for an arbitrary packet can be expressed as a sum of costs:

$$\begin{aligned} X^{H2} &= (\text{initial transmission}) + (\text{retransmissions}) \\ &\quad + (\text{receiving NAKs}) + (\text{receiving periodic HACKs}) \\ X^{H2} &= X_f + \sum_{i=1}^{M} X_p(i) + \sum_{m=2}^{M} X_n(m) + BX_\phi \qquad (1) \end{aligned}$$

where X_f is the time to get a packet from a higher layer, $X_p(i)$ is the time for (re)transmission attempt i, $X_n(m)$ is the time for receiving NAK m from the receiver set, X_ϕ is the amortized time to process the periodic HACK associated with the current congestion window, and M is the number of transmissions attempts the source will have to make for this packet. Taking expectations, we have

$$\begin{aligned} \mathrm{E}[X^{H2}] &= \mathrm{E}[X_f] + \mathrm{E}[M]\,\mathrm{E}[X_p] \\ &\quad + (\mathrm{E}[M]-1)\,\mathrm{E}[X_n] + B\,\mathrm{E}[X_\phi] \qquad (2) \end{aligned}$$

Following our previous analysis for tree-based protocols [10], we derive the value of M, given that the source has a local receiver subset of size B from which to collect NAKs and retransmit packets to. The expected number of transmissions per packet is [2, 8]

$$\mathrm{E}[M] = \sum_{i=1}^{B} \binom{B}{i} (-1)^{i+1} \frac{1}{(1-p^i)} \qquad (3)$$

It is shown in [9] that $\frac{\mathrm{H}_B}{-\ln p} \le \mathrm{E}[M] \le 1 + \frac{\mathrm{H}_B}{-\ln p}$, where $\mathrm{H}_B = \sum_{i=1}^{B} 1/i$, the harmonic numbers. From the known inequality $\ln(1+p) \ge \frac{p}{1+p}$, it follows that $-\ln p < \frac{p-1}{p}$. Using this result, assuming all operations (e.g. X_f and X_p) are of constant cost, and taking into account that $\mathrm{H}_B \in O(\ln B)$, it is shown in [8] that

B	-	Branching factor of a tree, the group size.
R	-	Size of the receiver set.
X_f	-	Time to feed in new packet from the higher protocol layer.
X_p	-	Time to process the transmission of a packet.
X_a, X_n, X_h	-	Times to process transmission of a ACK, NAK, or HACK.
X_t, Y_t	-	Time to process a timeout at a sender or receiver node respectively.
Y_p	-	Time to process a newly received packet.
Y_f	-	Time to deliver a correctly received packet to a higher layer.
Y_n, Y_h	-	Times to transmit a NAK , or HACK respectively.
$X_\phi Y_\phi$	-	Times to process the reception and transmission, respectively, of a periodic HACK .
p	-	Probability of loss at a receiver; losses at different receivers are assumed to be independent events.
M_r	-	Number of transmissions necessary for receiver r to successfully receive a packet.
M	-	Number of transmissions for all receivers to receive the packet correctly; $M = \max_r\{M_r\}$
X^{H2},Y^{H2}	-	the processing time per packet at the sender and receiver respectively in protocol $H2$
H^{H2}	-	Processing time per packet at a hop node in tree-based protocols.
Λ_x^w	-	Throughput for protocol $w \in \{A, N1, N2, R, H1, H2\}$ where x is one of the source s, receiver (leaf) r, or hop-node h. No subscript denotes overall system throughput.

Figure 1: Notation

protocol	*processor requirements*	*p as a constant*	$p \to 0$	*correctness*
Sender-initiated [8]	$O\left(R(1+\frac{p\ln R}{1-p})\right)$	$O(R\ln R)$	$\longrightarrow O(R)$	safe and live
Receiver-initiated NAK-avoidance [8]	$O\left(1+\frac{p\ln R}{1-p}\right)$	$O(\ln R)$	$\longrightarrow O(1)$	not correct
Ring-based (*unicast retrans.*) [10]	$O\left(1+\frac{(R-1)p}{1-p}\right)$	$O(R)$	$\longrightarrow O(1)$	safe and live
Tree-based [10]	$O(B(1-p)+pB\ln B)$	$O(1)$	$\longrightarrow O(1)$	safe and live
Tree-based with local NAPP	$O\left(1+(\frac{1-p+p\ln B+p^2(1-4p)}{1-p})\right)$	$O(1)$	$\longrightarrow O(1)$	safe and live

Figure 2: Analytical bounds and results on correctness.

$$\mathrm{E}[M] \in O\left(1+\frac{p}{1-p}\ln B\right) \tag{4}$$

Using Eq. 4, we can bound Eq. 2 as follows

$$\mathrm{E}[X^{H2}] \in O(1+1+\frac{p}{1-p}\ln B) \in O(1+\frac{p}{1-p}\ln B) \tag{5}$$

It then follows that when p is a constant $\mathrm{E}[X^{H2}] \in O(1)$.

3.1.2 Leaf nodes

Let Y^{H2} denote the requirement on nodes that do not have to forward packets (leaves). Let $Y_p(i)$ be the time it takes to process the (re)transmission i, $Y_n(i)$ be the time it takes to send NAK i, $X_n(i)$ be the time it takes to receive NAK i (from another receiver), Y_t be the time to set the i^{th} timer, Y_f be the time to deliver a packet to a higher layer, and Y_ϕ be the amortized cost of sending a periodic HACK for a group of packets of which this packet is a member.

$$\begin{aligned} Y^{H2} &= \text{(receiving transmissions)} + \text{(sending periodic HACKs)} \\ &\quad + \text{(sending NAKs)} + \text{(receiving NAKs)} \\ Y^{H2} &= \sum_{i=1}^{M}(1-p)Y_p(i) + Y_f + Y_\phi \\ &\quad + \sum_{i=2}^{M}(\frac{Y_n}{B} + (B-1)\frac{X_n(i)}{B}) \\ &\quad + Prob\{M_r > 2\}\sum_{i=2}^{M_r-1} Y_t(i) \end{aligned} \tag{6}$$

Taking expectations of Eq. 6,

$$\begin{aligned} \mathrm{E}[Y^{H2}] &= \mathrm{E}[M](1-p)\,\mathrm{E}[Y_p] + \mathrm{E}[Y_f] + \mathrm{E}[Y_\phi] \\ &\quad + (\mathrm{E}[M]-1)(\frac{\mathrm{E}[Y_n]}{B} + (B-1)\frac{\mathrm{E}[X_n]}{B}) \\ &\quad + \mathrm{Prob}\{M_r > 2\}(\mathrm{E}[M_r|M_r > 2] - 2)\,\mathrm{E}[Y_t] \end{aligned} \tag{7}$$

It follows from the distribution of M_r that [8]

$$\mathrm{E}[M_r|M_r > 1]\,\mathrm{E}[Y_t] = (2-p)/(1-p) \tag{8}$$

$$\mathrm{E}[M_r|M_r > 2]\,\mathrm{E}[Y_t] = (3-2p)/(1-p) \tag{9}$$

Therefore, noting that $Prob\{M_r > 2\} = p^2$, we derive the expected cost as

$$\begin{aligned} \mathrm{E}[Y^{H2}] &= \mathrm{E}[M](1-p)\,\mathrm{E}[Y_p] + \mathrm{E}[Y_f] + \mathrm{E}[Y_\phi] \\ &\quad + (\mathrm{E}[M]-1)\left(\frac{\mathrm{E}[Y_n]}{B} + (B-1)\frac{\mathrm{E}[X_n]}{B}\right) \\ &\quad + p^2\left(\frac{3-2p}{1-p} - 2\right)E[Y_t] \end{aligned} \tag{10}$$

Again, using the bound of $\mathrm{E}[M]$ given in Eq. 4, we can bound Eq. 10 by

$$\mathrm{E}[Y^{H2}] \in O\left(1+(\frac{1-p+p\ln B+p^2(1-4p)}{1-p})\right) \tag{11}$$

When p is treated as a constant $\mathrm{E}[Y^{H2}] \in O(1)$.

3.1.3 Hop nodes

To evaluate the processing requirement at a hop node, h, we note that a node caught between the source and a node with no children has a two jobs: to receive and to retransmit packets. Because it is convenient, and because a hop node is both a sender and receiver, we will express the costs in terms of X and Y. Our sum of costs is

$$\begin{aligned}H^{H2} &= \text{(receiving transmissions)} + \text{(sending periodic HACKs)} \\ &\quad + \text{(receiving periodic HACKs)} + \text{(receiving NAKs)} \\ &\quad + \text{(sending NAKs)} + \text{(retransmissions to children)} \\ H^{H2} &= (1-p)\sum_{i=1}^{M} Y_p(i) + Y_\phi + BX_\phi + Y_f \\ &\quad + \sum_{i=2}^{M}\left(\frac{Y_n(i)}{B} + (B-1)\frac{X_n(i)}{B}\right) \\ &\quad + \text{Prob}\{M_r > 2\}\sum_{i=2}^{M_r=1} Y_t(i) \\ &\quad + \sum_{i=2}^{M}(X_n(i) + X_p(i)) \qquad (12)\end{aligned}$$

Computing the expected value of H^{H2},

$$\begin{aligned}\mathrm{E}[H^{H2}] &= (1-p)\,\mathrm{E}[M]\,\mathrm{E}[Y_p] + \mathrm{E}[Y_\phi] + B\,\mathrm{E}[X_\phi] + \mathrm{E}[Y_f] \\ &\quad + (\mathrm{E}[M]-1)\left(\frac{\mathrm{E}[Y_n]}{B} + (B-1)\frac{\mathrm{E}[X_n]}{B}\right) \\ &\quad + p^2\left(\frac{3-2p}{1-p} - 2\right)\mathrm{E}[Y_t] \\ &\quad + (\mathrm{E}[M]-1)(\mathrm{E}[X_n] + \mathrm{E}[X_p]) \qquad (13)\end{aligned}$$

In other words, the average cost on a hop node is the same as a source and a leaf, without the cost of receiving the data from higher layers and one less transmission (the original one)

$$\mathrm{E}[H^{H2}] = \mathrm{E}[Y^{H2}] + \mathrm{E}[X^{H2}] - \mathrm{E}[X_f] - \mathrm{E}[X_p] \quad (14)$$

Therefore, Eq. 13 can be bounded by

$$\begin{aligned}\mathrm{E}[H^{H2}] &\in O(E[Y^{H2}]) \cup O(\mathrm{E}[X^{H2}]) \\ &\in O\left(1 + \left(\frac{1-p+p\ln B + p^2(1-4p)}{1-p}\right)\right) \quad (15)\end{aligned}$$

When p is a constant $\mathrm{E}[H^{H2}] \in O(1)$. Therefore, all nodes in the tree-NAPP protocol have a constant amount of work to do with regard to the number of receivers.

3.1.4 Overall system Let $\Lambda_s^{H2} = 1/\,\mathrm{E}[X^{H2}]$, $\Lambda_h^{H2} = 1/\,\mathrm{E}[H^{H2}]$, $\Lambda_r^{H2} = 1/\,\mathrm{E}[Y^{H2}]$ equal the throughput at the sender, hops, and leaves, respectively, then

$$\Lambda^{H2} = min\{\Lambda_s^{H2}, \Lambda_h^{H2}, \Lambda_r^{H2}\} \quad (16)$$

From Equations 5, 11, 15, and 16, it follows that

$$1/\Lambda^{H2} \in O\left(1 + \left(\frac{1-p+p\ln B + p^2(1-4p)}{1-p}\right)\right) \quad (17)$$

Accordingly, if either p is a constant or $p \to 0$, we obtain from Eq. 17 that $1/\Lambda^{H2} \in O(1)$. Therefore, the maximum throughput of the tree-NAPP protocol, as well as the throughput with non-negligible packet loss, is independent of the number of receivers. Tree-based protocols is the only class of reliable multicast protocols that exhibits such a degree of scalability with respect to the number of receivers.

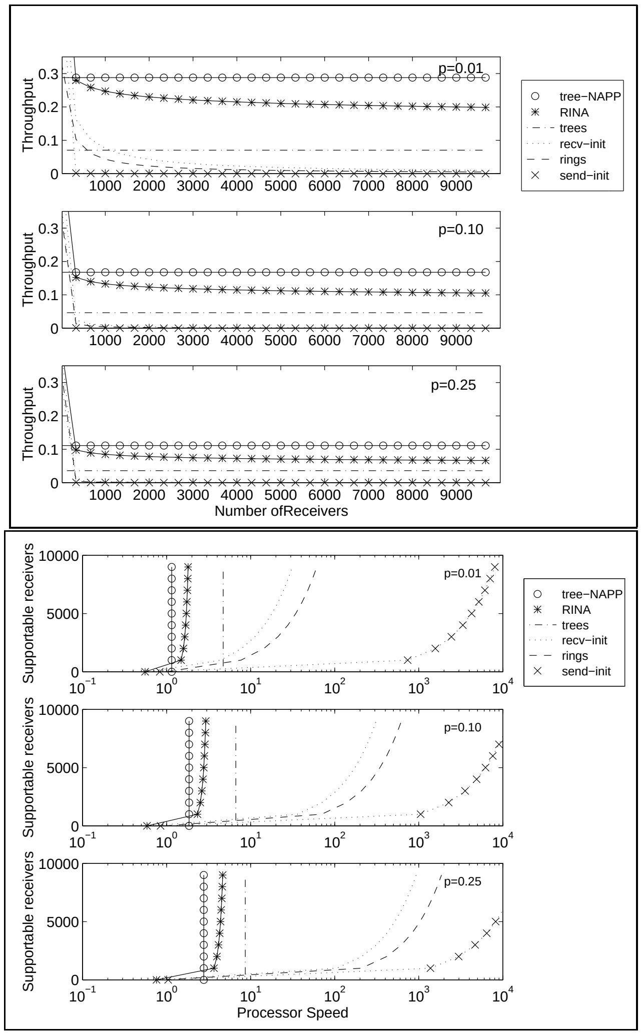

Figure 3: Above: The maximum throughput for each protocol. Bottom: Number of supportable receivers for each protocol. The branching factor for trees is set at 10.

3.2 Numerical Results

To examine the relative performance of the various classes of protocols, all mean processing times are set equal to 1, except for the periodic costs X_ϕ and Y_ϕ which are set to 0.1. Figure 3(a) compares the relative throughputs of the protocols described in Section 2. The graph represents the inverse of Eq. 13 as the exact expected throughput for tree-NAPP protocols as well as the throughput equations derived in [8, 10] for all other classes. The top, middle and bottom graphs correspond to increasing probabilities of packet loss, 1%, 10%, and 25%, respectively.

The throughput of tree-based and tree-NAPP protocols are independent of the size of the receiver set, and therefore any increase in processor speed would directly increase throughput. A smaller branching factor would also increase throughput at the cost of a longer path that retransmissions must traverse to an expecting receiver.

Figure 3(b) shows the number of supportable receivers by each of the different classes, relative to processor speed requirements.

This number is obtained by normalizing all classes to a baseline processor. As described in [8, 9], the baseline uses a sender-initiated protocol and can support exactly one receiver. As in [8, 9], let $\mu^{\omega}[R]$, be the speed of the processor that can support at most R receivers under protocol ω, where $\omega \in \{A, N1, N2, R, H1, H2\}$ representing sender-initiated, receiver-initiated, RINA, ring-based, tree-based, and tree-NAPP, respectively. If we set $\mu^{A}[1] = 1$ as a baseline it is shown in [8] that

$$\mathrm{E}[X^A]\Bigg|_{R=1} = \frac{1}{\mu^A[1]}\frac{3-p}{1-p} = \frac{3-p}{1-p}$$

The speedup of tree-NAPP protocols can be calculated as the ratio of their expected cost (Eq. 13) to the baseline

$$\begin{aligned}\mu^{H2}[R] &= \frac{1}{\mathrm{E}[X^A]}\,\mathrm{E}[H^{H2}] \\ &= \frac{1}{\mathrm{E}[X^A]}((4-p)\,\mathrm{E}[M] - 1.9 + 0.1B + p^2(\frac{3-2p}{1-p} - 2)\end{aligned}$$

In [8, 10], the number of supportable receivers derived for sender- and receiver-initiated, RINA, ring-based, and tree-based protocols are shown to be

$$\begin{aligned}\mu^{A}[R] &= \frac{1}{\mathrm{E}[X^A]}\,\mathrm{E}[M](2 + R(1-p)) \\ \mu^{N1}[R] &= \frac{1}{\mathrm{E}[X^A]}(1 + \mathrm{E}[M] + Rp/(1-p)) \\ \mu^{N2}[R] &= \frac{1}{\mathrm{E}[X^A]}(2\,\mathrm{E}[M]) \\ \mu^{R}[R] &= \frac{1}{\mathrm{E}[X^A]}\left(3 + \frac{2(R-1)p}{(1-p)}\right) \\ \mu^{H1}[R] &= \frac{1}{\mathrm{E}[X^A]}(-1 + \mathrm{E}[M](4 + B - (2+B)p))\end{aligned}$$

Because the exact value of $\mathrm{E}[M]$ is difficult to compute for large values of R, as in [8, 9], we use the following approximation

$$\mathrm{E}[M] \approx a + \frac{(H_{35} - H_R)}{\ln(p)}$$

where a is the value of $\mathrm{E}[M]$ for $R = 35$ and H_k is the harmonic series. When evaluating $\mu^{H1}[R]$, or $\mu^{H2}[R]$ an exact value for $\mathrm{E}[M]$ is used because the number of receivers is always $R = B = 10$.

From Figure 3, it is clear that only the tree-based classes can support any number of receivers for the same processor speed bound at each node. It is also clear that, in terms of performance, tree-NAPP protocols are superior to other classes. Of course, our model constitutes only a crude approximation of the actual behavior of reliable multicast protocols. In the Internet, an ACK or a NAK is simply another packet, and the failure to deliver a given packet correctly to a receiver is correlated with what happens at other receivers, because packets are distributed along multicast routing trees. Nevertheless, our approximate model is still a valuable tool as a first-order comparison of reliable multicast protocols and produces results that should be expected. Because tree-based protocols delegate responsibility for retransmission to receivers and because they employ techniques applicable to either sender- or receiver-initiated protocols within local groups (i.e., a node and its children in the tree) of the ACK tree only, any mechanism that can be used in a receiver-initiated protocol can be adopted in a tree-based protocol, with the added benefit that the throughput and number of supportable receivers is completely independent of the size of the receiver set, regardless of the likelihood with which packets are received correctly at the receivers. The rest of this paper describes how to construct shared ACK trees in a scalable and fault-tolerant manner.

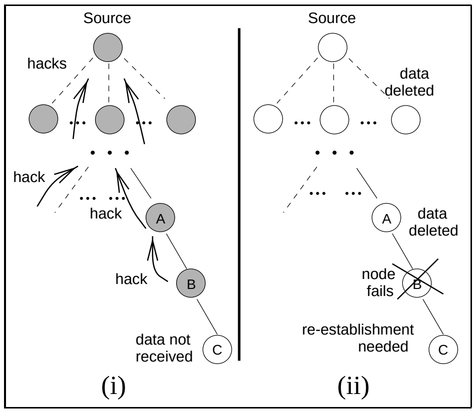

Figure 4: Packet loss for a subtree due to a hop node failure.

4 DEALLOCATING MEMORY

The ACK tree structure greatly improves throughput over other classes. However, an inherent weakness in the basic approach of delegating the responsibility of retransmissions to group leaders is that the failure of one or more hop nodes can cause an entire subtree to lose a series of packets during re-establishment of the tree.

After node failure, applications could either (a) terminate the session, (b) continue the session without providing a reliable service to nodes while they are temporarily disconnected, or (c) allow nodes to rejoin and catch up with the session. For the first two cases, changes in the ACK tree do not create a problem, and the *cw* and *mw* can advance together at the source and each hop node. The rest of this section considers necessary extensions for the last case. Figure 4 illustrates the problem with an example. Packets are multicast from the source to the receiver set; nodes that have received the data correctly are shaded. Packets are acknowledged as they are received, rather than waiting for the acknowledgments from children. Node A, and all other nodes that have received HACKs from all their children, delete packets; however, B fails before it is able to confirm that all its children have correctly received the data. If we assume that at least one child C has not received the data, then there is no node with whom to re-establish contact that will definitely have a copy of the data. One solution to this problem is to buffer the entire session in a secondary store, as is done in RMTP and SRM. However, this solution can become unscalable.

Ideally, we would like to keep data in a finite secondary store only until all current receivers have correctly received the data, without having any node keeping track of who all the receivers are. Fortunately, deadlocks due to receiver failures or reconfigurations of the ACK tree or the underlying multicast routing tree can be easily avoided in tree-based protocols by introducing aggregate ACKs that propagate from the receivers up the tree to the source. The ACK sent from a node to its parent in the tree consists of its own ACK and the aggregated ACKs from all its children. Just as in the generic tree-based protocol, correctly received data packets are acknowledged using HACKs. However, packets are not deleted at this point, they are kept in a secondary store or partition of memory. When a parent of a leaf node confirms that all its children have correctly received the data, it deletes the data from secondary store and sends an aggregated ACK to its own parent. Hop nodes do the same procedure. In terms of our taxonomy, aggregate ACKs are used to move the *mw* and HACKs (and negative HACKs) to move the *cw*.

The following two additional mechanisms are used together with aggregated ACKs to ensure that a disconnected node or subtree is never allowed to rejoin the ACK tree *after* the source has erased data from memory that the rejoining node or subtree never received.

First, a node that perceives one of its children as disconnected assumes the reception of any pending aggregated ACK from that child and sets a topology-change notification flag in its own aggregated ACK. The setting of the flag is preserved as the aggregated ACK travels back to the source. The flag instructs the source to wait for an even longer period of time before erasing the associated data from memory *after* receiving all the aggregated ACKs from its children.

Second, an orphan node is given a finite amount of time to reconnect to the ACK tree. This time is much shorter than the time set in a "connect timer" at the source. Once an orphan node times out, it cannot join the session and catch up without application-level support. The next section provides more details on the handling of orphans within the context of Lorax.

5 SHARED ACK TREES

For a *concurrent* multicast session, it is not reasonable to manage a separate ACK tree for every source. To remedy this, Lorax supports the proper dissemination of all acknowledgments in a multicast group along a single *shared* ACK *tree* of the concurrent multicast session. The routing scheme used in Lorax is adapted from a technique developed for the routing of messages between multiple processor elements described in [24].

Consider an ACK tree created for one original source, in which nodes have at most B children. The tree can be re-hung as an acknowledgment tree with any other node as the root, and all nodes will still have at most B children. This is a well known property of trees that allow us to apply the constant-cost results of Section 3 to any protocol utilizing a shared-tree for concurrent multicast sessions.

When an ACK tree is created for a single-source multicast session with the source as the root of the tree, the routing of aggregate ACKs to the appropriate hop node towards the source is simple: each node HACKs to its designated parent. However, the situation is more complicated for shared ACK trees. The introduction of multiple sources clashes with the inherent anonymity of the tree: receivers in the ACK tree lack knowledge of where each source is located, knowledge that can be used to route ACKs to the appropriate hop node leading to a particular source. In this paper we refer to the actions a node takes to discover which adjacent node on the shared ACK tree lies on the path to a particular source node on the shared ACK tree as *routing*.

Routing in an anonymous ACK tree can be done efficiently using *implicit routing*, with which each node is labeled based on its position in the tree. All packets from a source include this label, from which receivers can infer to which child or parent to route towards that source. One such labeling scheme is presented in [25], but the algorithm requires the entire tree to be relabeled when any node is added or deleted. Our adaptation of [24] involves only two nodes for a completely new addition (the added node and its parent), and deletions require re-labeling of the subtree of the deleted node when patched back into the ACK tree.

First we describe the construction of the shared ACK tree, then the labeling and routing scheme. We then describe tree maintenance, including how nodes can split off children and how node deletions are handled.

5.1 Ack Tree Construction

Our approach assumes the existence of the multicast routing tree(s) provided by the underlying multicast routing protocols. In the Internet, these trees will be built using such protocols as Distance Vector Multicast Routing Protocol (DVMRP) [17], Core Based Trees (CBT) [19], Ordered Core Based Trees (OCBT) [20], or Protocol Independent Multicast (PIM) [18].

To construct the ACK tree, Lorax utilizes a combination of root-based and off-tree schemes to grow the tree. These schemes are based on the common expanding ring search (ERS) technique over the underlying multicast routing tree(s) and mechanisms intended to limit the cost of each ERS.

The ACK tree is grown from a single root node using either the source multicast routing tree of the root node or the common multicast routing tree of the multicast group. The root node may be selected before the session starts and advertised together with the multicast address, or may be selected when the session begins using an election algorithm.

After joining the IP multicast address, all nodes are considered *off-tree* except for the root node of the ACK tree. The root immediately begins multicasting invitation-to-join messages (INV) using the underlying multicast routing tree with a time-to-live (TTL) value of zero in the IP header, and sets a timer T_{INV}. An off-tree node that hears an INV message unicasts a request (REQ) to be adopted back to the inviting node. If an inviting node does not hear a REQ before T_{INV} expires, it multicasts a new INV with a larger TTL value and resets T_{INV} to a longer timeout. When a REQ is received correctly at the inviting node, a bind message (BIND) is sent to the new child confirming the adoption. Once the new child receives the BIND, it becomes an on-tree node and starts the same process again by multicasting an INV. This process stops at any on-tree node (i.e., a node that is "growing the ACK tree") when the node has several children, or the TTL field of its INV reaches a maximum value.

Note that the maximum TTL of an INV is much smaller than the TTL of data packets or the TTL needed to cover the entire underlying multicast routing tree. This root-based strategy to create the ACK tree is used to avoid excessive traffic over the multicast routing tree. In practice, this scheme should suffice to create the ACK tree, because most if not all members of a reliable multicast group will want to participate in the ACK tree (i.e., receive information reliably).

In the unlikely scenario in which a large number of members of the reliable multicast group does not want to receive information reliably, growing the ACK tree from the root only may result in the formation of a "frontier" of leaf nodes on the ACK tree that may not reach nodes interested in receiving packets reliably but who are beyond the maximum allowed TTL of INVs from frontier nodes. To account for this case, Lorax includes an off-tree scheme for off-tree nodes to reach the ACK tree.

Allowing off-tree nodes to freely multicast until they find a parent may cause the underlying multicast routing tree to become congested with search messages. This method is similar to the single-source tree construction method presented in [6]. Lorax solves this problem by limiting the scope of ERS multicasts needed to reach the ACK tree. More specifically, consider an off-tree node that joins the multicast session and call it orphan node o. This node starts multicasting query (QRY) messages looking for a new parent in an ERS fashion after one of the following outcomes occurs: (a) A timeout expires after joining the multicast session without the reception of an INV message from a node in the ACK tree, or (b) having sent its REQ a number of times, a timeout expires without receiving the corresponding BIND.

When an off-tree node h_1 receives node o's QRY, it responds to o with a DIF message. Node o may receive multiple such replies, and can pick any one of the responding nodes as its helper in joining the ACK tree. If o chooses h_1 as its helper, the two nodes then periodically send nexus messages (NEXUS) to each other verifying that there is a *nexus* from o to h_1. A nexus is a directed connection from o to h_1 corresponding to o's attempt to reach the ACK tree; it can be terminated only by o or a resource failure. Node o need not multicast more QRYs as long as its nexus with h_1 is valid. Node h_1 diffuses the ERS towards the ACK tree by multicasting QRYs according to ERS. After a number of ERS attempts, node h_1's search may either be successful and reach the ACK tree, or be unsuccessful and reach another off-tree node h_2 willing to help. In the latter case, a nexus from h_1 to h_2 is established, and h_2 helps diffusing node o's ERS towards the ACK tree.

Note that the chain established by the diffusion of node o's ERS by one or various helpers is not part of the ACK tree; it is only used to contain the span of the ERS multicasts needed for the orphan node to reach the frontier of the ACK tree.

Once an on-tree node hears a QRY message for the ERS started by node o, it unicasts a response message (RSP) to node o. This message indicates that the sending node is willing to adopt the orphan o. All on-tree nodes are required to respond to any QRY, and nodes that end up having more children than they can handle go through a process of *fission*, described subsequently.

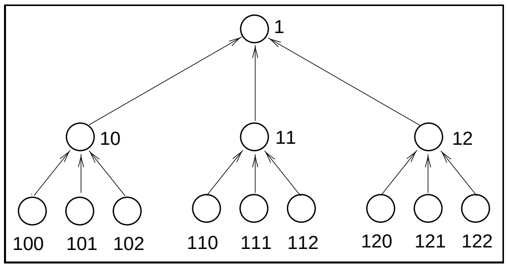

Figure 5: An example of the labeling scheme.

An orphan node may receive more than one RSP, in which case the orphan unicasts a REQ to one of the on-tree nodes willing to adopt and the process outlined above continues. If the orphan node already started an ERS to join the ACK tree, it also sends a terminate message (TERM) to its helper to erase the nexus it had established to reach the ACK tree. Any node that is active helping node o that receives a TERM from its incident nexus sends a TERM on its outgoing nexus. This process continues until the chain started by node o to reach the ACK tree is erased.

A node declares a nexus built to help orphan o to be invalid after timing out without receiving a NEXUS from the other node in the nexus. In that case, the helper closer to the orphan node o (or node o itself) starts sending QRYs again (i.e., it attempts to get a new helper to reach the ACK tree) and the node to which the nexus was incident sends a TERM on its outgoing nexus to erase the rest of the chain of helpers.

Note that a node may participate in the diffusion of multiple ERSs, each for a different orphan. Each search is is treated independently. Once a node is on-tree, it must receive a label used in routing of acknowledgments, for restructuring of the tree during node deletions, and for the fissioning of local groups.

5.2 Labeling and Routing

The algorithm used in Lorax for routing HACKs and aggregate ACKs is based on the following simple scheme: if a source is not in a receiver's subtree, then HACKs should be sent to the parent; otherwise, the HACK should be sent to the child who heads the appropriate subtree.

Some common definitions are used for our formal description of the protocol. As an initial framework, we represent the network as an undirected graph $G = (V, E)$, where V is the finite set of nodes, and $E = V \times V$ is the set of edges representing the (currently operational) bi-directional links between nodes. We require each node $x \in G$ to have a unique name $n(x)$ (e.g., an IP host address).

Let $T \subset G$ be the ACK tree over which acknowledgments are routed. The protocol assigns a unique integer label $l(x)$ to each node x such that all nodes descendent from x contain $l(x)$ as the prefix of their respective labels. Figure 6 describes the hierarchical labeling algorithm, where $\circ$ is the concatenation operator. When the labeling algorithm terminates, every node in the ACK tree has a unique label, illustrated in Figure 5. This label is used by the ACK routing algorithm also shown in Figure 6. Basically, a receiver must check if the source is in its subtree. If it is, then the label of

```
LABEL (graph G)
        Construct a tree $T \subseteq E$ from $G$ rooted
          at some node $s$, as above;
        label $l(s) = 1$;
        Call LABEL-SUBTREE($s$,T,1);

LABEL -SUBTREE (node r, tree T, integer $\delta$),
        Let $\omega = 0$;
        For each child $c$ of $r$ do:
            label $l(c) = \delta \circ \omega$;
            Let $\omega = \omega + 1$;
            Call LABEL-SUBTREE($c$,T,$l(c)$);
ROUTE -PACKET (tree T, node n, node s, packet p)
        If   $|l(n)| > |l(s)|$
            Then route the packet to the parent of $n$;
            Else compare the first $|l(n)|$ low order bits
              of $l(n)$ and $l(s)$;
                If they are not equal
                  Then route the packet to the
                    parent of $n$;
                  Else the next $\lceil \log_2 B \rceil$ explicitly
                    states to which child of $n$ the packet
                    should be routed.
```

Figure 6: Algorithms for ACK tree labeling and routing of ACKs.

the receiver node will be a prefix of the label of the source node. The ACK routing algorithm routes packet h acknowledging source S's data to the proper hop node for receiver R. Let $|l(x)|$ denote the cardinality of the label of node x, i.e., the number of bits it contains. Each source node includes its label in all packets it transmits. It is trivial for each receiver to store the label in the table containing retransmission information (e.g., the last sequence number received correctly). If at any point the source is assigned a new label, it must be multicast to all the receivers.

On average, nodes closer to the root of the tree have to compare fewer bits than leaf nodes. The cardinality of the label grows well with an increasing receiver set. Each level adds an additional $(\log_2 B)$ bits, and a tree of n receivers has $(\log_B n)$ levels. The number of bits needed is therefore exactly $(\log_2 B)(\log_B n) = \log_2 n$. Consequently, if 32 bits were used in the packet headers for this label, then a tree could handle 2^{32} nodes.

5.3 Tree Maintenance

5.3.1 Group Fission The analysis of Section 3 assumes a constant B which bounds the degree of each node in the tree. In practice, the value of B can be chosen independently at each node; i.e., some machines are more capable than others. For reasonable performance, nodes should not set B so low that only a few nodes can be supported as children, and B should not be set any higher than the nodes can support efficiently.

It is clear, then, that an algorithm is needed to keep the number of children at or below B at each node; we refer to this process as *fission*. Arbitrarily assigning a new parent to children would not preserve grouping well, and so we propose again using the ERS heuristic for fissioning a group. An easy heuristic is to simply disconnect extra children and let them ask nodes in other local groups for adoption; however, this may create unscalable amounts of work at some nodes.

Our fission algorithm requires that parent nodes keep track of how many additional nodes its current children may take. This information is easily included periodically in HACKs, or as part of a NAPP algorithm. The parent node sends an adopt message (ADOPT) to the child with the most free space. The ADOPT forces the adopting node to multicast to all the nodes in its local group in the ACK tree a request-for-children message (RFC). The nodes that respond first with a QRY message are currently closer and are sent an RSP message by the adopting node and the fission is completed. From here, the labeling algorithm must be run on the new subtrees of the adopting nodes.

The question remains of how many new children to force onto the adopting node. Initially it is advantageous to just reduce the number of nodes to slightly below B. If three fissions happen close to each other, measured by a timer T_{growth}, a heuristic is to require that the third fission reduce the number of nodes down to half B, and the timer is reset. This drastic fissioning is motivated by the fact that if many fissions happen in such a short period of time, then the tree is most likely in a period of growth, and needs to be expanded. Each additional fission within the T_{growth} period also separates half the children and resets the timer. When no fissions happen within a full T_{growth} period, the node initializes to small fissions again.

5.3.2 Deletions We use the same algorithm for accidental and intentional deletion of nodes from the ACK tree; only the initiating conditions are different. The algorithm is motivated by the need for a fast distributed algorithm that would not force all disconnected nodes to be children of one parent, causing fission. We assume that all children of a deleted node are operating close to their B limit, and cannot take on B more children.

To describe a simple method of restructuring the ACK tree, we present the following relation. Define "at least as old as", with operator "$\geq$" for two children x, y of a common parent: x is *at least as old as* y if integer values $l(x) \geq l(y)$.

The algorithm is simple and starts when the parent node multicasts a deletion message (DEL) to the members of its local group, or when a node detects that it is an orphan. Since all nodes have labels before the deletion starts, all even-labeled nodes become the child of the next lowest (that is, next eldest) even-labeled node. All odd-labeled nodes become the child of next lowest (even-)labeled node. Since all nodes have a list of their siblings, unicast QRYs are sent directly to the proper node, and an RSP is expected in response (A BIND is not required). If new parents do not respond, then the node joins the ACK tree as if it were a new node.

While this algorithm is completing, the eldest node starts multicasting QRYs to all nodes in the multicast group using the groups's multicast address. Note that as long as this node does not join with descendents, the partial ordering of the tree is preserved. In support of this, we also require that nodes retain their label until a new parent is found. The reason is that the eldest node may join with a disconnected node previously in its subtree. A loop is formed if that disconnected node then re-joins with a node in the eldest node's subtree before it can be relabeled. As long the disconnected node retains the old label, this scenario cannot happen.

Because all other maintenance in the tree is ERS-heuristic-driven, it is likely that the new parents are close in the network topology. However, this is not guaranteed, and a node may wish to keep a counter of how many times its parent has been deleted; it can then rejoin the ACK tree as a new node when a certain value has been exceeded.

When the root of the ACK tree becomes disconnected from the tree, the eldest child becomes the new root.

5.3.3 Orphans If a node has lost contact with its parent for a time T_{orphan}, long enough so that the cause is clearly not congestion, it considers itself an orphan, and will have to rejoin the ACK tree by initiating the method described above. Clearly, all descendents of the orphaned node do not have to rejoin the ACK tree, but must be relabeled with the orphaned node's new prefix.

When a node is orphaned it may choose to enact the secondary acknowledgment protocol described in Section 4. The only thing an orphan must do is contact each source in the ACK tree, instructing them that the node might be orphaned, and to not delete data until it receives a new parent. If there are enough sources, the orphan may choose for efficiency to multicast this information. A node may choose to contact sources at time $T_{paranoid} < T_{orphan}$ to be sure the hold message reaches the source in enough time. If the sources are following the secondary protocol, then normally they will not delete data until the second ACK is received from all children, or a certain very long timeout T_{source} has been reached. In practice, nothing is guaranteed, but if T_{source} is much longer than T_{orphan} we expect the protocol to work.

Showing that Lorax is loop-free is simple, because the relation "$\geq$" is reflexive, transitive, and anti-symmetric and is also a partial ordering. Let $x \rightarrow y$ denote that x is the parent of y because there exists an edge in the tree T between x and y. Assume that at some time during the operation of Lorax, there is a path of nodes in the ACK tree such that $a \rightarrow b \rightarrow \cdots \rightarrow c \rightarrow a$, and $a \neq b \neq c$. Lorax requires that $x \geq y$ if x is the parent of y, then it must be true that $x \geq y$. It follows that both $a \geq c$ and $c \geq a$ must be true. The only way in which this can be true is if $a = c$, which is a contradiction, and it follows that Lorax produces loop-free routing of ACKs at all times.

6 COMPARISON WITH RELATED WORK

As we have summarized in our taxonomy of Section 2, there is a growing body of work on reliable multicast protocols for internetworks. Our results in Section 3 clearly indicate that tree-based protocols are the first choice in terms of performance, with RINA protocols being the second. Not surprisingly, RINA and tree-based protocols are the two prominent approaches for the implementation of reliable multicasting today.

The main motivation for RINA protocols is that using NAKs frees the sender from having to process every ACK from each receiver. Two additional advantages are that the source is not supposed to know the receiver set and and the receivers pace the source. However, RINA protocols suffer from a number of limitations.

First, the RINA protocols that have been proposed to date (e.g., SRM [11] and LBRM [12]) have no mechanism for the source to know when it can safely release data from memory [10]. LBRM uses a hierarchy of log servers that store information indefinitely and receivers recover by contacting a log server. Using log servers is feasible only for applications that can afford the servers and leaves many issues unresolved. If a single server is used, performance can degrade due to the load at the server; if multiple servers are used, mechanisms must still be implemented to ensure that such servers have consistent information. On the other hand, SRM simply requires that data needed for retransmission be rebuilt from the application. Since the application is never informed when data has been successfully delivered to all receivers, all data is stored at all sources (and at all willing receivers) for the length of the session.

Second, if error recovery in a RINA protocol depends solely on timeouts at the receivers, end-to-end delays can become arbitrarily large. For example, SRM requires every receiver to multicast periodic "session messages" specifying the highest sequence number accepted from a source and a time-stamp used by the receivers to estimate the delay from the source. The sequence number in a session message is in effect an ACK to the last packet from the source, and a receiver can keep "polling" the source periodically to ensure that the source eventually delivers missing packets not caught by the NAK scheme. This clearly limits the scalability of SRM, because the persistence of session messages forces every node to know the receiver set.

Third, NAKs and retransmissions must be multicast to the entire multicast group to allow suppression of NAKs. The NAK-avoidance was designed for a limited scope, such as a LAN, or the small number of Internet nodes that can be expected in a local group of an ACK tree. This is because the basic NAK-avoidance algorithm requires that timers be set based on updates multicast by every node. As the number of nodes increases, each node must do increasing amount of work! Even worse, nodes that are on congested links, LANs or regions may constantly bother the rest of the multicast group by multicasting NAKs.

On the other hand, tree-based protocols eliminate the ACK implosion problem and free the source from having to know the receiver set, provide maximum end-to-end delays that are bounded, and operate solely on messages exchanged in local groups (between a node and its children in the ACK tree). As we show in Section 3, the amount of work required at each node for tree-NAPP protocols does not increase with the number of group members, i.e., the throughput of such protocols is not dependent on the number of group members.

The only two concerns regarding the practicality of tree-based protocols are whether finite memory can be used and the effort needed to build and maintain a "reasonable" structure for the ACK tree that can be modified in a dynamic and scalable manner. Our approach addresses all prior concerns with tree-based protocols. We have shown in section 4 how to make tree-based protocols work correctly

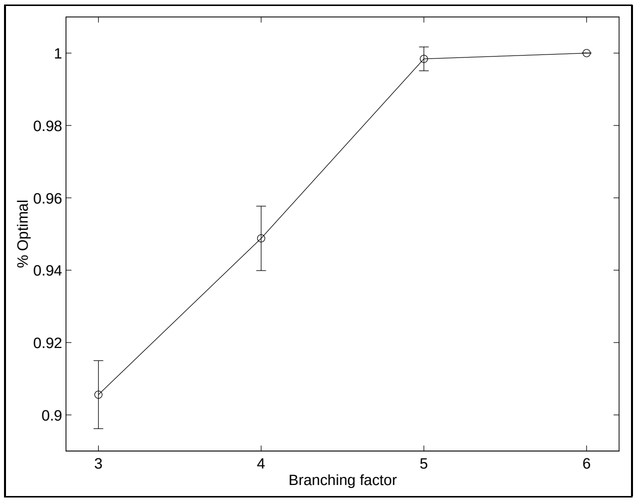

Figure 7: Optimality with 95% confidence intervals shown as vertical lines.

with finite memory. Second, Lorax is the first protocol that provides a shared ACK tree for efficient use among multiple sources and receivers in a concurrent multicast session, and the ACK tree is maintained dynamically in the presence of changes to the receiver set or the underlying multicast routing tree.

7 QUALITY OF ACK TREES

Throughout this paper we have described the construction of ACK trees making no assumptions regarding the structure of the underlying multicast routing tree(s). However, there is much to be gained by using a shared multicast routing tree such as created by CBT or OCBT. With such an underlying routing tree, packets are multicast from each source to the receivers through the same structure; receivers closer to the source receive packets before receivers down the tree do, and there is a correlation of packet loss at nodes hanging from the multicast routing subtree of a router. The more these relationships are preserved in the ACK tree, the better the ACK tree performs, because latencies and retransmissions within each local group of the ACK tree have a direct correspondence with delays, congestion, and errors that occur in the routing tree.

We define an ACK tree as *optimal* if, for all paths in the underlying multicast routing tree that start from the router adjacent to a parent node in the ACK tree and terminate at a router adjacent to its child node in the ACK tree, any receivers adjacent to a router lying on that path necessarily are children of the same parent node in the ACK tree. Unfortunately, obtaining an optimal ACK tree may be at odds with the number of children in the ACK tree that any given host can support in practice.

To gain insight on the optimality of the ACK trees built with Lorax, we performed a number of simulations.[2] A single routing tree was created using a simulation of CBT running on top of the Distributed Bellman Ford algorithm in a network of 25 nodes. The routing tree has its core (root) at node 10, and each routing node in such a tree has a maximum degree of 6. A node of the ACK tree was attached to each routing node, and each such node was selected as the root of the ACK tree. For each placement of the ACK tree root, each node of the ACK tree was allowed to have a maximum degree of 3, 4, 5, and 6, and Lorax was run to obtain the corresponding ACK tree in each of the 100 cases. For each ACK tree obtained by Lorax, we counted the number of nodes in the ACK tree that adhere to our definition of ACK tree optimality. An ACK tree node adheres to our optimality principle if its router is a descendant (on the routing tree) of the router of its ACK tree parent.

The results from this simulation experiment indicate that Lorax tends to build an ACK tree that is optimum according to our definition. Lorax always built optimum ACK trees when nodes in the ACK tree can support up to 6 neighbors, and for the rest of the cases it builds ACK trees with more than 90% of their nodes adhering to the optimality principle. The simulation results are graphed in Figure 7. The type of topologies considered in [11] to analyze SRM's performance correspond to the case in which Lorax produces optimum ACK trees, and Lorax with a tree-NAPP protocol should provide performance better than or at least equal to the best performance that can be expected from SRM.

As the node degree needed for an optimum ACK tree exceeds the maximum degree that can be supported by nodes in the ACK tree, the structure of the ack tree deviates from the routing tree structure. This corresponds to the case in which receivers of the multicast group are sparsely distributed over a routing tree.

8 CONCLUSION

We have established that tree-NAPP protocols have better performance than all other classes of reliable multicast protocols using a maximum throughput model. We have also presented solutions to several open questions concerning the implementation of shared ACK trees. Preserving reliability during restructuring of the ACK tree is easily guaranteed using aggregated acknowledgments that propagate from each leaf towards the source. It is not necessary to use aggregate ACKs in conjunction with a tree-based congestion window scheme. It is possible to use a (shared) tree for deallocating memory and an unstructured receiver-initiated scheme for retransmission requests. This class of tree-based receiver-initiated NAK-avoidance (TRINA) protocols can be viewed as an extension to RINA protocols like SRM and LBRM so that packets can be deleted safely. Our future work continues to define instances of this new subclass of protocols.

Lorax maintains scalable operation with multiple sources by constructing and maintaining a shared ACK tree. Overhead traffic is contained during initial ACK tree construction by growing the ACK tree from a known root. Impatient nodes are quieted down and allowed to join the ACK tree by means of expanded ring searches that are narrow in scope. Hierarchical labeling of each node makes implicit routing of acknowledgments simple and preserves loop-free routing of such acknowledgments over the ACK tree at all times. For the case in which a shared multicast routing tree is used at the network layer, the ACK trees built with Lorax mirrors the multicast routing tree.

Although our empirical evidence shows that Lorax creates ACK

[2]We thank Rooftop Communications Corporation for donating the C++ Protocol Toolkit.

trees that are reasonably close to an underlying shared multicast routing tree, changes in routing tables and group membership can make the two trees differ from one another over time. Furthermore, more efficient mechanisms could be adopted in Lorax if hosts were allowed to know more about the structure of the underlying multicast routing trees. Our work continues to address the opportunities presented by the hierarchical labeling of routers, namely the ability to provide a directed-multicast service over an existing IP-multicast routing tree. With a small change in the protocols now being proposed for the creation of multicast routing trees, Lorax can make intelligent choices when constructing and maintaining the ACK tree. Multicast routing protocols such as CBT and OCBT can create a single tree from which an arbitrary root node can easily be picked (e.g., one of the cores) to start the labeling algorithm. It is trivial to incorporate the labeling scheme presented in Section 5.2 into these multicast routing protocols. Forthcoming publications define this service more formally and the associated protocols, and address the dynamics of Lorax in large multicast groups.

(Positive) ACKNOWLEDGMENTS

We would like to thank Clay Shields for his generous help with the C++ Protocol Toolkit, as well as for allowing us to use his implementation of the CBT protocol in our simulations. This work was supported in part by the Office of Naval Research under Grant N00014-94-1-0688, and by the Defense Advanced Research Projects Agency (DARPA) under Grant F19628-96-C-0038.

REFERENCES

1. S. Deering, “RFC-1112: Host extension for ip multicasting.” Request For Comments, August 1989.
2. S. Ramakrishnan and B. N. Jain, “A negative acknowledgment with periodic polling protocol for multicast over lan,” in *Proc. IEEE Infocom*, pp. 502–511, March 1987.
3. J.-M. Chang and N. F. Maxemchuk, “Reliable broadcast protocols,” *ACM Transactions on Computer Systems*, vol. 2, pp. 251–273, August 1984.
4. S. Paul, K. K. Sabnani, and D. K. Kristol, “Multicast transport protocols for high speed networks,” in *International Conference on Network Protocols*, pp. 4–14, 1994.
5. J. C. Lin and S. Paul, “RMTP: A reliable multicast transport protocol,” in *Proc. IEEE Infocom*, pp. 1414–1425, March 1996.
6. R. Yavatkar, J. Griffioen, and M. Sudan, “A reliable dissemination protocol for interactive collaborative applications,” in *Proc. ACM Multimedia*, pp. 333–44, 1995.
7. B. Whetten, S. Kaplan, and T. Montgomery, “A high performance totally ordered multicast protocol.” Available from research.ivv.nasa.gov by ftp `/pub/doc/RMP/RMP_dagstuhl.ps`, August 1994.
8. S. Pingali, D. Towsley, and J. F. Kurose, “A comparison of sender-initiated and receiver-initiated reliable multicast protocols,” in *Performance Evaluation Review*, vol. 22, pp. 221–230, May 1994.
9. S. Pingali, *Protocol and Real-Time Scheduling Issues for Multimedia Applications*. PhD thesis, University of Massachusetts Amherst, September 1994.
10. B. N. Levine and J.J. Garcia-Luna-Aceves, “A comparison of known classes of reliable multicast protocols,” in *Proc. IEEE International Conference on Network Protocols*, October 1996.
11. S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, “A reliable multicast framework for light-weight sessions and application level framing,” in *Proc. ACM SIGCOMM'95.*, pp. 342–356, August 1995.
12. H. Holbrook, S. K. Singhal, and D. R. Cheriton, “Log-based receiver-reliable multicast for distributed interactive simulation,” in *Proc. ACM SIGCOMM'95*, pp. 328–341, August 1995.
13. W. T. Strayer, B. Dempsey, and A. Weaver, *XTP: The Xpress Transfer Protocol*. Addison-Wesley Publishing Company, 1992.
14. A. Koifman and S. Zabele, “RAMP: A reliable adaptive multicast protocol,” in *IEEE Infocom'96*, pp. 1442–1451, March 1996.
15. M. Grossglauser, “Optimal deterministic timeouts for reliable scalable multicast,” in *IEEE Infocom'96*, pp. 1425–1441, March 1996.
16. Y. Ofek and B. Yener, “Reliable concurrent multicast from bursty sources,” in *IEEE Infocom'96*, pp. 1433–1441, March 1996.
17. S. Deering and D. Cheriton, “Multicast routing in datagram inter-networks and extended lans,” *ACM Transactions on Computer Systems*, vol. 8, pp. 85–110, May 1990.
18. S. Deering, D. Estrin, D. Farinacci, V. Jacobson, and others., “An architecture for wide-area multicast routing,” in *Proc. ACM SIGCOMM'94*, pp. 126–135, 1994.
19. T. Ballardie, P. Francis, and J. Crowcroft, “Core based trees (CBT): An architecture for scalable inter-domain multicast routing,” in *Proc. ACM SIGCOMM'93*, pp. 85–95, October 1993.
20. C. Shields, “Ordered core based trees,” Master's thesis, University of California — Santa Cruz, Santa Cruz, California, June 1996.
21. Dr. Seuss, *The Lorax*. Random House, 1971.
22. Jon B. Postel, ed., “RFC-793: Transmission control protocol.” Request For Comments, September 1981.
23. D. D. Clark, M. L. Lambert, and L. Zhang, “NETBLT: A high throughput transport protocol,” in *Proc. ACM SIGCOMM'93*, pp. 353–359, Aug. 1987.
24. D. Summerville, J. Delgado-Frias, and S. Vassiliadis, “A high performance pattern associative oblivious router for tree topologies,” in *Proc. Eighth International Parallel Processing Symposium*, pp. 541–545, April 1994.
25. N. Santoro and R. Khatib, “Labeling and implicit routing in networks,” *The Computer Journal*, vol. 28, pp. 5–8, February 1985.