Slingshot: Time-Critical Multicast for Clustered Applications*

Mahesh Balakrishnan Stefan Pleisch Ken Birman
Department of Computer Science
Cornell University, Ithaca, NY-14853
{*mahesh, pleisch, ken*}*@cs.cornell.edu*

Abstract

Datacenters are complex environments consisting of thousands of failure-prone commodity components connected by fast, high-capacity interconnects. The software running on such datacenters typically uses multicast communication patterns involving multiple senders. We examine the problem of time-critical multicast in such settings, and propose Slingshot, a protocol that uses receiver-based FEC to recover lost packets quickly. Slingshot offers probabilistic guarantees on timeliness by having receivers exchange FEC packets in an initial phase, and optional complete reliability on packets not recovered in this first phase. We evaluate an implementation of Slingshot against SRM, a well-known multicast protocol, and show that it achieves two orders of magnitude faster recovery in datacenter settings.

1. Introduction

A contemporary datacenter consists of hundreds to thousands of commodity machines collocated on a building-wide network. The software base running on such a datacenter is typically structured as a large collection of autonomous, fine-grained services. Each service is spread over several nodes, for purposes of fault-tolerance and load-balancing, and application tasks are carried out by multiple services acting in various degrees of coordination. For instance, a buy order coming into the backend for a commercial website triggers off a flurry of distributed updates involving different services, each responsible for a small, individual subtask - such as billing, inventory, user profiling, and so on. As a result of this service oriented style of computing, many-to-many communication - or multicast, with many senders - comprises a large fraction of the traffic within a datacenter. A multicast primitive is, thus, a very useful building block for constructing datacenter applications, provided it counters the failure-prone nature of COTS hardware and software with coherent, efficiently realized reliability guarantees.

In this paper, we consider the problem of time-critical reliable multicast within a datacenter. We characterize time-critical services as requiring low-latency reliable communication - and willing to pay coherent, tunable overheads for it. Such time-critical services coexist on datacenter nodes with other applications of a less urgent nature. During periods of overload, it is desirable that timely delivery of time-critical data is continued, even if this involves temporarily slowing down other protocols on the network. The requirements of time-critical applications can vary from hard real-time guarantees - absolute bounds on both reliability and timeliness - to probabilistic, best-effort estimates of these two metrics. In reliable multicast literature, gossip-style protocols have enabled a transition from absolute guarantees given failure bounds - such as a maximum of k nodes failing, beyond which all bets are off - to probabilistic guarantees that degrade gracefully in the face of any degree of failure. We believe that similar probabilistic guarantees on timeliness are of value to applications of a time-critical nature, especially in the COTS world of the datacenter, where the entire communication stack is composed of unreliable components oblivious to timing constraints. Traditionally, this space has been targeted by *real-time* protocols which offer deterministic guarantees under bounded failures, but slow down delivery in the average case to achieve this objective. For applications dealing in perishable data with short life-spans, such as stock quotes in a financial center or location updates coming from military sensors, there is a need for protocols that deliver a large percentage of data very quickly, offering performance that degrades gracefully under arbitrary failure conditions and providing optional recovery of remaining data at a higher latency. Our goal is to design a time-critical multicast primitive offering such probabilistic guarantees on timeliness.

Reliable multicast is a well-researched field, with a wealth of existing solutions for timely, guaranteed delivery.

*Our effort is supported by DARPA's SRS program, the AFRL/Cornell Information Assurance Institute, the Swiss National Science Foundation, and a MURI grant.

However, the unique combination of a service-oriented architecture and a datacenter networking topology is one that existing protocols are not designed to deal with efficiently. Communication between and within services results in multicast groups with many senders transmitting small messages at uneven rates, across nodes and over time. State-of-the-art reliable multicast protocols - especially those optimized for delivery time - are not designed to deal with such communication patterns. Further, datacenter topologies occupy a unique point in the networking problem space, scaling beyond the reach of conventional LAN protocols while escaping much of the complexity of WAN deployments. Exploiting the positive aspects of such topologies - such as high bandwidth, proximity, and flat routing structure - while retaining scalability is an open problem.

Our key idea is to employ Forward Error Correction (FEC) at the receivers of a multicast. FEC is a well-known technique for introducing reliability into multicast, involving the injection of redundant packets into a stream to enable receivers to recover from packet loss. Traditional FEC is strictly a sender-based mechanism, and has been used to good effect in single-sender settings involving steady constant-rate streams of data. We explore the idea of having *receivers* in a multicast group encode incoming data into FEC packets, which are then exchanged proactively to repair losses. We present a protocol called Slingshot, which layers unreliable IP Multicast with a gossip-style scheme for disseminating receiver-generated FEC packets. Slingshot offers applications tunable, probabilistic guarantees on timeliness, and allows users to choose either probabilistic or complete reliability.

We evaluate Slingshot on a rack-style cluster and compare its time-critical properties to Scalable Reliable Multicast [8], a well-known reliable multicast protocol. We believe SRM to be a closer match for time-critical, multi-sender settings than other existing protocols, and hence a valid baseline to measure Slingshot against. Slingshot achieves recovery of lost packets two orders of magnitude faster than SRM in our evaluation, a finding that highlights the value of receiver-based FEC in the datacenter. We supplement this evaluation on a real cluster with simulation results that highlight the scalability of Slingshot to larger system sizes.

The rest of the paper is organized as follows: In Section 2 we describe the operating conditions for a datacenter multicast protocol, in terms of workload and network behavior, and how existing reliable multicast protocols perform within these constraints. Section 3 describes the operation of Slingshot in detail, along with its overheads and assumptions, and provides an analysis of the protocol. Section 4 presents evaluation results for Slingshot. Section 5 places Slingshot in context with related work, and Section 6 concludes the paper.

2 Design Space

A reliable multicast protocol aimed at time-critical datacenter settings has to take into account the expected nature of communication within groups, as well as the characteristics of the underlying network. As mentioned in the introduction, the service oriented nature of datacenter computation results in multicast groups with large numbers of senders transmitting at varying rates. We obtain some insight into the characteristics of datacenter networks from a recent description of the Google File System (GFS) [10], which is set against a networking backdrop of large clusters of machines distributed across many machine racks, with switches facilitating inter-rack communication. The paper's evaluation setup involves 1 Gbps links between switches and 100 Mbps links to machines, indicating that high-bandwidth links are the norm. The combination of such flat, high-capacity routing structures and inexpensive commodity hosts results in communication characteristics in the datacenter being determined by the end-points, rather than the network. For instance, inter-node latency is dominated by the time spent by packets in either protocol stack, with a negligible quantum of time spent on the actual wire. More pertinently for reliable multicast protocols, packet loss occurs at the end nodes due to buffer overflows, and not at intermediary routers and switches. This allows a datacenter multicast protocol to assume that packet loss occurs independent of network structure (though other kinds of loss correlation will exist, such as nodes running the same services getting overloaded at the same time). Further, the fact that time-critical data entering a node comprises only a fraction of the total incoming traffic at that node allows us to assume that there is no significant temporal correlation of losses; a buffer overflow would affect applications across the board, and not just the time-critical segment. Hence, we can assume that the pattern of losses visible to the time-critical protocol is not bursty and optimize for the case where packets are lost singly.

With this set of assumptions in mind, we examine the current solution space available to reliable multicast protocol designers. Most reliable multicast protocols layer a packet loss discovery/recovery mechanism over an unreliable delivery primitive, such as IP Multicast [6] or some form of overlay multicast [4]. While IP Multicast is not widely deployed on the Internet, it is a viable alternative in a datacenter context, given the attendant administrative homogeneity and relatively lower number of dimensions in which scalability is required. Introducing reliability over this unreliable delivery primitive decomposes into two intertwined questions: detecting that a node has not received a packet, and recovering that lost packet. Existing reliability schemes can be divided based on delegation of responsibility into two classes: sender-based and receiver-based.

2.1 Sender-Based Reliability

In sender-based schemes [14, 12], the sender of a multicast is responsible for ensuring that the data is delivered to all receivers. The trivial extension of unicast reliability to a multicast scenario involves positive acknowledgements: each receiver sends an ACK for every packet back to the sender. However, if multicast is used heavily, this causes ACK implosion, where the sender is overwhelmed by acknowledgements from its many receivers. A standard way to avoid ACK implosion is to use ACK trees, which are used to aggregate acknowledgements before passing them on to the sender. For example, RMTP and RMTP-II [14, 12] use such a hierarchical structure to collect ACKs and respond to retransmission requests (both are designed for single sender settings). In a time-critical setting, though, any kind of hierarchical structure imposes unacceptable latency on the discovery process, since the sender has to wait for a reasonable amount of time to allow the acknowledgement aggregate to percolate up the tree before it declares packets lost. In general, sender-based reliability mechanisms have several disadvantages. In many cases, the sender is likely to be busy, and going back to it for retransmissions might overload it. Also, the round trip time to the sender might add unnecessary latency to the packet recovery process, particularly if there are less loaded receivers near the affected node from which lost packets could be recovered.

2.2 Receiver-Based Reliability

Receiver based schemes [1, 8] for reliable multicast place the burden of discovering and recovering from packet loss on the receivers. Many solutions use sender specific sequencing for discovering packet loss: the sender numbers the messages it multicasts, and a receiver knows it missed a message if it receives the next message in the sequence. If messages are delivered out of order by the transport subsystem, timeout thresholds are used to determine if a packet is truly lost. Once a packet is declared lost, one recovery mechanism is to send a NAK, or negative acknowledgment, requesting retransmission of the packet. The NAK can be sent back to the sender, to a nearby receiver, or multicast to the entire group, as in SRM [8]. SRM discovers loss only when it receives the next packet from the same sender; an alternative, such as in Bimodal Multicast[1], is to have receivers gossiping message histories with each other to expedite discovery of loss. Here, once a packet is discovered to be missing, a request for it can be sent back to the node who initiated the gossip exchange and the packet can be recovered.

2.3 Forward Error Correction

Forward Error Correction (FEC) [11, 13, 16] is used in scenarios where the latency of a two-phase discovery/recovery mechanism is unacceptable, and where contacting the sender for missing packets is undesirable or impossible. In its simplest form, FEC involves creating l repair packets from m data packets such that any m out of the resulting $(m + l)$ packets is enough to recover the original m data packets [11]. Traditional applications of FEC to reliable multicast have the sender generate l repair packets for every m data packets and inject them into the data stream. Hence, for every block of $m + l$ packets, a receiver is insulated from l packet losses. FEC is particularly attractive as a reliability mechanism as it imposes a constant overhead on the system and has easily understandable behavior under arbitrary network conditions. However, it is designed primarily for situations where a single sender is transmitting data at a high, steady rate: for example, bulk file transfers [2], or video and audio feeds [3].

3 Slingshot

Slingshot uses an unreliable multicast mechanism such as IP multicast for an initial best-effort delivery, and then injects point-to-point error correction traffic between receivers to proactively recover lost packets. Like conventional FEC, it generates a constant percentage of error correction packets, with the difference that each receiver encodes over all incoming packets within a group, as opposed to a sender encoding on the packets it sends. In some sense, Slingshot leverages the multiplicative increase in message density at multicast receivers — the fundamental cause of ACK implosion — to use error correction techniques that require a certain critical rate of input packets, decreasing the latency of packet loss recovery/discovery as a result. Below, we describe the basic operation of Slingshot, describe the nature of the overheads it imposes and offer analytical bounds on its performance.

3.1 Protocol Details

Message loss discovery and recovery in Slingshot occurs in two phases. Phase 1 involves the proactive dissemination of repair packets between receivers, and guarantees *fast* discovery and recovery of a high percentage of lost packets; this is the main contribution of this paper. Phase 2 is optional and configurable, and allows for complete recovery of all packets. Slingshot deals with two types of packets: *data packets*, which contain the original data multicast within the group and are uniquely identified by message IDs in the form of *(sender, sequence number)* tuples, and *repair*

packets, which contain recovery information for data packets.

In Phase 1, Slingshot introduces a constant percentage overhead on communication by having receivers send each other repair packets. For every r data packets that a node receives via the underlying unreliable multicast, it generates a repair packet using FEC and sends it to c other randomly selected receivers. Slingshot uses XOR, the simplest and fastest form of FEC, which allows the recepient of a repair packet to recover from one missing data packet. To ensure that both data and repair packets are sent in the network without fragmentation, data packets are limited to a size slightly smaller than the Maximum Transmission Unit (MTU); this ensures that a repair packet has space for the XOR, which is equal to the size of a single data packet, and a list of contents, comprised of r message IDs describing data packets which can be recovered using it. We say that a data packet is *contained* within a repair packet if it can be recovered from the latter. The fraction of overhead traffic and the resulting percentage of lost packets recovered by Phase 1 are directly determined by the 2-tuple parameter (r, c), which we call the *rate-of-fire*. We give an analysis for the expected percentage of lost packets recovered in Section 3.4.

To facilitate recovery of lost data packets from repairs, each node maintains a *data buffer*, where it stores incoming data packets. When a node receives a repair packet, it first checks the list of contents to determine if there are data packets included that it has not received. If only one such packet is included in the repair, it retrieves the other $r - 1$ data packets from its data buffer and combines them with the XOR contained in the repair packet to recover the lost packet. Further, each node maintains a *repair bin* where it stores pointers for upto r recently received data packets, to be included in the next repair packet it sends.

The probabilistic nature of Phase 1 results in a small percentage of lost packets going unrecovered. This usually happens when all incoming repairs at a node containing a particular lost data packet include other losses, making them useless, and also in the unlikely event that a node does not receive a repair containing the lost packet, due to the random manner in which nodes pick destinations for repair packets. The probabilities attached to these two cases depend on the *rate-of-fire* parameter, allowing us to tune the reliability provided by Phase 1 to desired levels. Even if the FEC traffic in Phase 1 does not enable packet recovery, it ensures that loss is discovered very quickly. In the case that the packet is not recoverable due to all relevant incoming repairs containing multiple losses, discovery takes place when the first such repair is received. At this point, the node can either wait for more repair packets to arrive, or run Phase 2 after some timeout period has elapsed.

In Phase 2, the node initiates recovery of the lost packet by sending an explicit request to some other node. If we use a scheme similar to Bimodal Multicast, this explicit recovery request is sent to some arbitrarily chosen receiver of the multicast - for instance, the sender of the repair packet - which services the request from its data buffer. Figure 1 illustrates the difference between Slingshot and Bimodal Multicast, if Phase 2 is structured in this manner. Another option is to send the retransmit request back to the sender, which requires senders to either buffer sent packets or delegate responsibility for reconstructing the packet to the application, as in SRM. Algorithm 1 gives the complete Slingshot algorithm, substituting the first option for Phase 2.

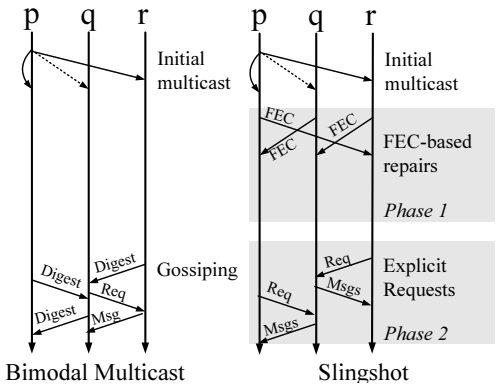


Figure 1. Comparison with Bimodal Multicast.

3.2 Overheads

In Phase 1, Slingshot imposes a constant percentage communication overhead on the data sent through, which depends on the rate-of-fire (r, c); for every r packets a node receives, it sends out c additional repair packets. The ratio of incoming repair packets to data packets over all nodes in the system is given by the simple formula: $\frac{(1-p)c}{r}$, where p is the probability - independent, consistent with our assumptions in Section 2 - of a packet being dropped at an end-node. Other than the rate-of-fire (r, c), the principal parameter to Slingshot is the size of its data buffer. Our experiments show that in an 8 Mbps group, lost packets are retrieved within a few milliseconds of the time they were originally sent at, which reduces buffering requirements to tens of milliseconds worth of data, or a few hundred Kbytes. An optional data structure is the *pending repairs buffer*, where repair packets containing multiple missing packets are temporarily stored, in case all but one of their included lost packets is recovered through other means; in practice, we achieve good results with the size of this buffer set to twenty packets. Finally, the computational overhead imposed by Slingshot is minimal and easily quantifiable: every incoming data packet is XORed incrementally to a buffer to pre-

Algorithm 1 Slingshot algorithm.

1: **Code of receiver** p_i**:**

2: Initialisation:
3: $DataBuffer \leftarrow \emptyset$ {*Contains received Data Packets*}
4: $RepairBin \leftarrow \emptyset$ {*Data packets to create next Repair Packet*}
5: $LastSeqNo[] \leftarrow \emptyset$ {*Last known data packet from each sender*}
6: $KnownLost \leftarrow \emptyset$ {*List of Message IDs thought to be lost*}

7: **upon** r-multicast(dp) **do**
8: send (dp) to all processes in v.

9: **upon** reception of data packet dp from sender p_j **do**
10: deliver dp to the application
11: $DataBuffer \leftarrow DataBuffer \cup \{dp\}$
12: $RepairBin \leftarrow RepairBin \cup \{dp\}$
13: markLost($dp.id$)
14: $KnownLost \leftarrow KnownLost - \{dp.id\}$
15: composeRepairPac()

{*Phase 1 Recovery*}
16: **upon** reception of repair packet rp from sender p_k **do**
17: **for all** message id $id \in rp.MsgIdList$ **do**
18: markLost(id)
19: **if** $|\{id | id \in KnownLost \wedge id \in rp.MsgIdList\}| = 1$ **then**
20: Recover data packet dp corresponding to this id
21: deliver dp to the application
22: $DataBuffer \leftarrow DataBuffer \cup \{dp\}$
23: $KnownLost \leftarrow KnownLost - \{dp.id\}$

{*Marks unreceived messages preceding passed-in id as lost*}
24: **procedure** markLost(id)
25: **for** $i = lastseqno[id.sender] + 1$ to $id.seqno$ **do**
26: $KnownLost \leftarrow KnownLost \cup \{(id.sender, i)\}$
27: $lastseqno[id.sender] \leftarrow id.seqno$

{*Constructs FEC repair packet*}
28: **procedure** composeRepairPac
29: **if** $|RepairBin| \geq r$ **then** {(r, c) *denotes the rate-of-fire*}
30: $DestinationSet \leftarrow$ select c processes q s.t. $q \in v$
31: generate repair packet rp
32: send rp to all q in $DestinationSet$
33: $RepairBin \leftarrow \emptyset$

{*Phase 2*}
34: **upon** receiving request message $rm(MsgIdList)$ from sender p_k **do**
35: **for all** data packet dp s.t. $dp.id \in MsgIdList \wedge dp \in DataBuffer$ **do**
36: send dp to p_k

{*Phase 2: runs periodically*}
37: **task**
38: **for all** Msg ID id in $KnownLost$ for longer than δt **do**
39: $MsgIdList \leftarrow MsgIdList \cup \{id\}$
40: send request message $rm(MsgIdList)$ to an arbitrary process p_l

pare the outgoing repair packet, and during recovery, $r - 1$ XORs are performed to extract the missing data packet from the repair packet. Phase 2 recovery imposes the extra overhead of an explicit request and response; we do not consider this significant, since we expect Slingshot's rate-of-fire to be tuned such that the percentage of packets left unrecovered in Phase 1 is very small.

3.3 Membership

Slingshot needs a weakly consistent membership service to provide each node with the list of other nodes in the group, known as a view, from which it can pick targets for repair packets randomly. The more accurately this view reflects actual group membership, the better Slingshot's discovery and recovery mechanism performs; the repair packets sent to nodes who are in a view but not in the group are wasted, and members of the group who are not accurately represented in views are less likely to receive sufficient repair packets. However, Slingshot's probabilistic nature allows it to perform well despite weakly consistent membership, allowing it to be layered over gossip-based membership protocols [15]. Slingshot also works well with partial views, where each node knows only a subset of the group; hence, a scalable, external membership service providing uniformly selected partial views, such as Scamp [9], can be used underneath it. In our implementation we include a simple state-machine replicated membership server that provides nodes with view updates, and have nodes perform failure detection in a ring. We believe such a solution to be appropriate for datacenter settings, where group sizes are limited to thousands of nodes and churn is not a concern. Because we do not require strong consistency, the overhead imposed by the underlying membership service is negligible: in our implementation, each node pings one other node once a second and membership updates are propagated lazily to nodes by the central service.

3.4 Analysis

We present a simplistic analysis to predict how the probability of a lost packet being recovered depends on the rate-of-fire parameter, the size of the group, the partial view size, and the probability of packet loss. We assume that packets are dropped at end node buffers with some fixed independent probability p, and that routers do not drop packets; this is consistent with our assumptions in Section 2. Given a group size N_G, a partial view size N_v and a rate-of-fire parameter (r, c), we want to predict the probability of recovering a given data packet dp lost at node n. We make the assumption that the partial view at a node is a uniformly chosen subset of the whole group membership, and always includes the node itself.

Now, let X be a random variable signifying the number of nodes in the system which have n in their partial views. Since each partial view is a uniformly picked subset of the group membership, the probability of n being included in a view other than its own is $p_v = \frac{\binom{N_G-2}{N_v-2}}{\binom{N_G-1}{N_v-1}}$. Thus, X has a binomial distribution with parameters N_G and p_v.

Given that the number of nodes including n in their views is a particular value i, let Y be a random variable denoting the number of repair packets originating at these nodes that include the data packet dp and are targeted at n. The upper bound on the total number of such packets is i, for the case that each of the i nodes receives dp without loss and sends out a repair packet to n. At any of the i nodes, the probability of n being selected as one of the c destinations is $p_c = \frac{\binom{N_v-2}{c-1}}{\binom{N_v-1}{c}}$. If we also consider the probability p of the sender of the repair packet dropping dp, then Y has a binomial distribution with parameters i and $p_c(1-p)$.

If we set the number of repair packets which include dp and are targeted at n to a value j, then the number of such packets received without loss by n is represented by a random variable Z that has a binomial distribution with parameters j and $1-p$. Let us set Z to the value k. Now, we need to compute the probability p_k of dp being recovered if n receives k repair packets containing it. We can recover dp if, for at least one of the incoming repair packets containing it, n has all the other $r-1$ data packets included in that repair packet. We derive inclusive upper and lower bounds p_{kL} and p_{kU} for p_k, where $k \geq 1$; it is equal to zero when $k = 0$. The lower bound corresponds to the case where all k repair packets have the same contents, and the upper bound is given by the case where all k repair packets are pair-wise disjoint; i.e they include completely different data packets. Hence, p_k is bounded by:

$$p_{kL} = (1-p)^{r-1} \;\leq\; p_k \;\leq\; p_{kU} = 1-(1-(1-p)^{r-1})^k$$

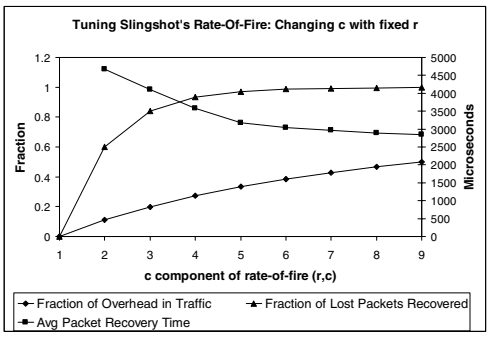
Hence, the final probability $P_{n,dp}$ of a data packet dp lost at node n being recovered successfully is bounded by:

$$\sum_{i=0}^{N_G} P(X=i) \sum_{j=0}^{i} P(Y=j) \sum_{k=0}^{j} P(Z=k) p_{kL} \;\leq\; P_{n,dp}$$

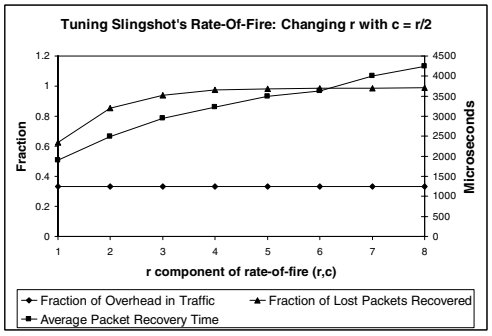
$$\leq \sum_{i=0}^{N_G} P(X=i) \sum_{j=0}^{i} P(Y=j) \sum_{k=0}^{j} P(Z=k) p_{kU}$$

4 Evaluation

We evaluated an implementation of Slingshot on a rack-style cluster of 64 nodes, consisting of four racks of blade-servers connected via two switches. We believe this cluster to be fairly typical of a datacenter setup, with average ping roundtrip time at around 100 microseconds. Unless indicated otherwise, our loss model involves packets being dropped at end nodes with .01 independent probability.



(a)



(b)

Figure 2. Tuning a Slingshot implementation on a 64-node cluster: tradeoff points available between reliability, timeliness and overhead by changing the rate-of-fire (r, c). Reliability and Overhead are plotted against the left y-axis, and Average Recovery Time on on the right y-axis.

4.1 Reliability and Timeliness *vs.* Overhead

Figure 2 shows how changing the rate-of-fire parameter affects the amount of overhead induced and the resulting reliability and timeliness characteristics. Figure 2a shows the effect of varying c for constant $r = 8$, and Figure 2b shows results for different values of r, keeping c at $r/2$ to maintain the same level of overhead. In these experiments, each node multicasts a packet once every 64 milliseconds, resulting in 1000 packets/second in the group. Note that the average recovery time includes discovery of packet loss. Most packet recovery mechanisms, including SRM [8], present only the time taken to recover packets in their results, ignoring discovery time. In a multi-sender setting, discovery time using sender-based sequencing would heavily dominate recovery time, averaging at 64 milliseconds in this experiment. In contrast, Slingshot performs both discovery and recovery within a few milliseconds. We run only Phase 1 of Slingshot

here, and the graph includes the fraction of packets recovered in this phase; running a second phase would provide complete reliability. After a certain point, increasing the amount of overhead by raising c has diminishing returns; for the rest of our simulations we set the rate-of-fire to be $(8, 5)$.

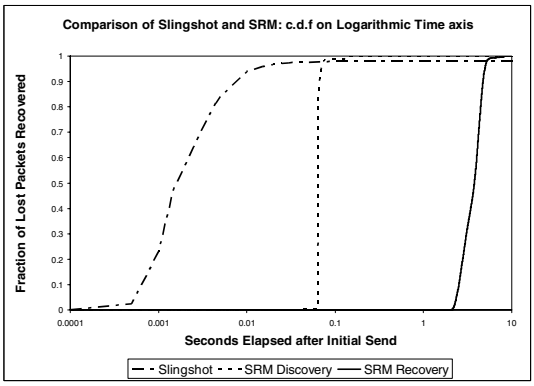


Figure 3. Comparison of Slingshot implementation against SRM on a 64 node rack-style cluster: c.d.f. of packets recovered for a 1000 packets/sec workload.

4.2 Comparison with SRM

Figure 3 shows a comparison of the time-critical properties of Slingshot against SRM. The experiment involves 64 nodes multicasting every 64 milliseconds, to achieve a data rate of 1000 packets per second within the group. The graph is a cdf of packets recovered against time taken (since the original unreliable send), on a logarithmic x-axis. SRM discovers packet loss through sender-based sequencing, which explains the steep rise of the "SRM Discovery" curve at around 64 milliseconds; a lost packet is not discovered until its sender multicasts again, 64 milliseconds later. In this particular run of SRM, roughly 53% of all traffic is repair overhead, while the Slingshot configuration used (rate-of-fire = $(8, 5)$) results in 38% of all traffic being overhead. Unless SRM is allowed to facilitate faster discovery by sending blank messages between actual multicasts, using up even more overhead, its recovery time is bounded below by the inter-send time. Hence, even if recovery were made faster by optimizing SRM's timing parameters for datacenter settings, it cannot take place faster than the inter-send time. Running only Phase 1 of the Slingshot protocol, we achieve recovery of almost all lost packets (97.5% in this run) two orders of magnitude faster than SRM.

4.3 Scalability

To assess the scalability of Slingshot beyond the limits of our 64-node cluster, we ran a simulation of the protocol on a 400 node network. Our topology consisted of 20 switches forming a star network around a gateway router, with each switch having 20 end-hosts under it. As in the implementation setup, end-hosts drop packets with 1% probability, and routers and switches do not drop packets. In Figure 4, we randomly select nodes from the 400 node topology to fill a group of a given size, and evaluate the percentage of packets recovered by Slingshot's Phase 1 in the resulting group. We also used the simulator to assess the impact of partial view size on Slingshot's performance. Figure 5 shows that Slingshot works as well with small, partial views as it does with global views.

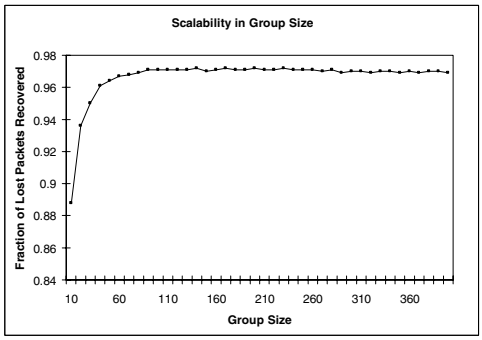


Figure 4. Effect of sroup size on Slingshot's Phase 1 recovery percentage in a 400 Node simulated datacenter.

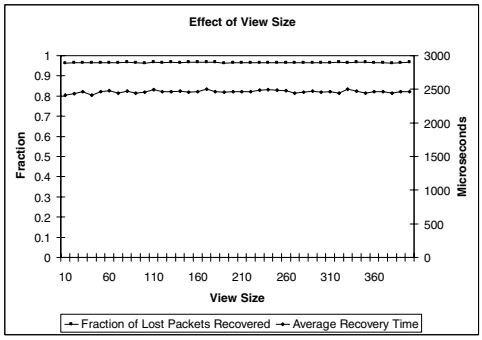


Figure 5. Effect of changing the partial view size in a 400 node simulated datacenter.

5 Related Work

Slingshot lies in the intersection of various distributed systems technologies, such as reliable multicast, FEC schemes, and real-time protocols. Amongst the reliable multicast protocols discussed in Section 2, the one closest to Slingshot is Bimodal Multicast (see Figure 1); they are both layered over IP Multicast, and involve receivers exchanging recovery information with each other. Slingshot has the obvious advantage over Bimodal Multicast of requiring only a single message send for recovery, as opposed to three.

Also, running Bimodal Multicast in time-critical settings would involve exchanging message digests at a very high rate. When digests are sent this frequently, the case where some receivers get a packet before others could trigger off many unnecessary two-step recoveries. Slingshot does not suffer from this problem, as recovering a packet from a repair is a very fast operation. The passing resemblance of Slingshot to Bimodal Multicast is partially by construction; we believe that this is one way of enhancing reliable multicast with receiver-based FEC, but one can imagine other alternatives, such as deterministically flooding repairs on an overlay, that offer guarantees of a different flavor.

Slingshot is closer to the reliable multicast space than to real-time protocols, due to the nature of its guarantees and operating assumptions. Real-time protocols either assume timing guarantees at a lower level or characterize the violation of timing assumptions as failures. For instance, the delta-t protocol [5] offers deterministic bounds on delivery, given a certain number of such failures. Our characterization of time-criticality as a need for very small, probabilistic time bounds is, hence, quite different from traditional notions of real-time requirements.

Lastly, FEC has long been a topic of extensive research; nonetheless, we are not aware of any existing work that proposes encoding repair packets at the receiver end. Most current research in FEC is focused on providing efficient ways to perform complex encodings that allow recovery from multiple losses in a stream. While we have restricted ourselves to using XOR for encoding, using more powerful forms of FEC is an avenue open to further exploration.

6 Conclusion

Slingshot offers unique probabilistic guarantees on timeliness in datacenter settings by placing FEC at the receivers of a multicast. It exploits the accumulated multicasting rates of multiple senders to achieve faster packet loss detection and recovery at the receivers. In our evaluation setup of a 64-node rack-style cluster, Slingshot recovers packets two orders of magnitude faster than Scalable Reliable Multicast, underscoring its utility as a building block for time-critical datacenter applications.

Acknowledgments

We thank Vidhyashankar Venkataraman for his help with the analysis of the protocol, Robbert van Renesse for his comments on Slingshot, and Luis Rodrigues for his pointers to real-time multicasts.

References

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[2] J. Byers, M. Luby, and M. Mitzenmacher. A digital fountain approach to asynchronous reliable multicast. *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[3] G. Carle and E. Biersack. Survey of error recovery techniques for ip-based audio-visual multicast applications. *IEEE Network*, December 1997.

[4] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12, 2000.

[5] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[6] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[7] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 443–452. IEEE Computer Society, 2001.

[8] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[9] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Computers*, 52(2):139–149, 2003.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43. ACM Press, 2003.

[11] C. Huitema. The case for packet level FEC. In *Protocols for High-Speed Networks*, pages 109–120, 1996.

[12] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, Apr. 1998. IETF Internet Draft.

[13] J. Nonnenmacher, E. W. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on Networking*, 6(4):349–361, 1998.

[14] S. Paul, K. K. Sabnani, J. C.-H. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal of Selected Areas in Communications*, 15(3):407–421, 1997.

[15] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, 28, 1998.

[16] L. Rizzo and L. Vicisano. A reliable multicast data distribution protocol based on software FEC techniques. In *The Fourth IEEE Workshop on the Architecture and Implementation of High Performance Communication Systems (HPCS'97)*, Sani Beach, Chalkidiki, Greece, June 1997.