Ricochet: Low-Latency Multicast for Scalable Time-Critical Services*

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

Abstract

Ricochet is a time-critical multicast protocol for use in clustered platforms and datacenters. Applications in such settings often are cloned for scalability and availability, hence updates involve multicast or publish-subscribe, with large numbers of heavily overlapping multicast groups. The applications that interest us are ones for which rapid response reflecting the most recent updates is important. Ricochet, a multicast protocol that combines native IP multicast with proactive forward error correction, achieves high levels of consistency with stable and tunable overhead. Evaluation on a 64-node rack-style cluster shows that existing technologies perform poorly relative to our goals, whereas Ricochet achieves extremely low and consistent delay, performs well with as many as 1024 multicast groups per node, and is not disrupted by packet loss. The price of this low latency is a modest loss in peak throughput.

1 Introduction

Clusters and datacenters play a growing role in the contemporary computing spectrum, providing back-end processing and storage in a wide range of commercial and military settings. Yet technology for building scalable applications to run on such platforms lags that available for building more traditional applications destined to run on a single machine, or on a single server node accessible over a network.

Of particular interest to us are mission-critical applications that combine *time-criticality* with other reliability requirements. For example, a cluster hosting an air traffic control service might need to provide rapid responses even when a node fails, and even if a surge in communications traffic is causing some packet loss within the network or host operating system. We would like to make it as easy as possible to build such a system.

A decade ago, technology supporting this kind of application might have represented a fairly esoteric, niche solution. But times have changed. One major factor is the emergence of time-criticality as a need in settings not normally understood to be mission-critical. Consider, for example, the importance of responsiveness and scalability in an e-tailer's datacenter. Companies such as Amazon, eBay and Google are betting their bottom lines that when a service is commercially successful and loads soar, they will be able to scale it up while maintaining interactive response. These companies are merely the first to encounter a need that will become ubiquitous.

Studies of best engineering practice make it clear that componentized software development reduces costs and results in better code. Tools supporting this style of development, and platform support for component integration and reuse, are increasingly standard. Thus a modern rapid-response system would surely adopt a componentized approach.

But when we take a componentized time-critical application and undertake to deploy it on a cluster or in a datacenter for reasons of scalability, we face significant challenges. The key steps are to build a non-clustered componentized solution, and then to scale-out the components by transforming them into sets of replicas. Next, reads can be load-balanced across replicas while updates are multicast to the entire replica set. Finally, one introduces self-monitoring and management logic, often in an event-notification or publish-subscribe style. In support of these steps we need a reliable, time-critical multicast that can guarantee good performance when subjected to the communication patterns generated by replicated components hosted on clusters. Unfortunately, existing multicast platforms are inadequate for this use.

Major clusters and datacenters may have enormous numbers of nodes (e-tailers routinely rent entire warehouses and fill them with tens or hundreds of thousands of machines) and each node might host multiple services. A heavily used component might need to be

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

scaled over dozens or even hundreds of nodes. A node may belong to large numbers of communication groups, each including tens or even hundreds of members, and they experience packet loss, application crashes, node failures, node restarts, etc. Against this tumultuous backdrop, our goal is to offer high quality time-critical multicast communication, and later to use it in our Tempest system, which automates many steps in the overall process.

Much of the research on time-critical communication has focused either on maximizing raw throughput without attention to latency[1] or on providing absolute guarantees of real-time response in embedded systems and other SCADA applications [9, 6]. Classic real-time mechanisms offer absolutely guaranteed response, but may be predictably slow and optimized for worst-case assumptions (see, for example, [9]). They generally assume that systems will be small and dedicated to one task, hence a typical node probably participates in just one (or very few) multicast groups, and those groups probably have just a few members each. Yet the applications of interest here require something different: a high probability of fast response, even at the cost of somewhat reduced throughput, guaranteed even when faults occur, that won't melt down even in large-scale deployments.

Lacking adequate platform support, the developer of a time-critical application faces a daunting problem. First, one must abandon the componentized style of development that yields the best quality solution: interactions between components represent a threat to interactive performance. Next, since the platform itself is also a threat to predictable responsiveness, the developer will need to minimize use of the SOA platform, interfacing to the technology base in the narrowest manner possible. At the end of the day, the developer will find him or herself using outmoded tools and developing a monolithic, proprietary solution.

We believe that the Ricochet protocol presented here, and the Tempest architecture we're building over it, is a better solution. Ricochet is layered over IP multicast and designed to minimize delay: a given packet is typically sent just once, delivered immediately upon receipt, and in the event of packet loss will generally be recovered and delivered in milliseconds, long before the loss could even be detected by the application. The system scales extremely well in realistic application scenarios. Finally, Ricochet is designed to fit naturally in service-oriented architectures (indeed, Tempest extends a popular web services platform). By making time-critical multicast easy and cheap, Ricochet facilitates the kind of high-productivity development style that has swept other aspects of the distributed computing community.

[1] Indeed, other members of our research group are working on a second system, QuickSilver Scalable Multicast (QSM), which flips the goals by emphasizing throughput in large-scale settings, treating latency as a less critical requirement. QSM, which will be described elsewhere, achieves about ten times the peak throughput of Ricochet, but incurs perhaps 2 orders of magnitude higher latency. We believe the tradeoff between guaranteed fast delivery and maximum throughput is fundamental.

1.1 Problem Statement

Accordingly, we focus on the problem of providing time-critical, reliable, scalable multicast. By *time-critical,* we mean that with high probability, the multicast message should be delivered rapidly and with low inter-arrival skew. We seek this goal even at the expense of some overheads and somewhat reduced peak throughput rates. By *reliable* we mean that the protocol should overcome bounded levels of message loss (which can arise in the operating system or on the network) and that it should tolerate bounded numbers of node crashes.

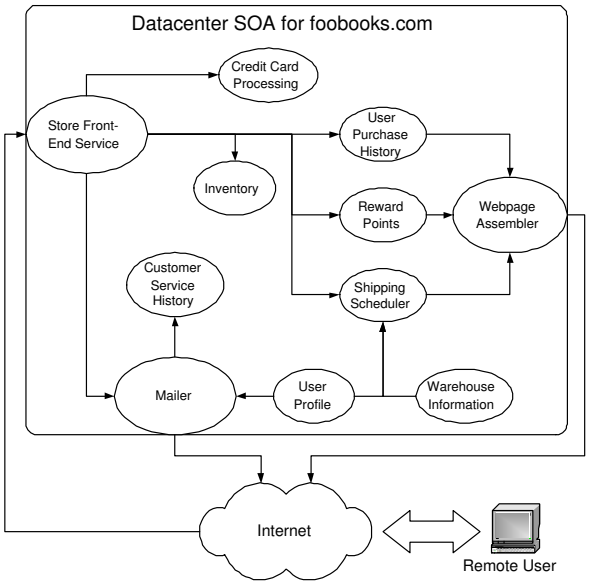

Figure 1. Example Micro-service Architecture

To understand precisely what we mean by *scalable*, consider Figure 1. Here we see a hypothetical service structure for an e-tailer's datacenter, with major components broken out as separate services. Such a system would need to access services concurrently to exploit parallelism, and each service would be cloned for fault-tolerance and availability, with its replicas running on multiple nodes. Multicast plays an important role here, in updating single services cloned at multiple machines, as well as updating multiple services in parallel.

Since a single node might host multiple services, we arrive at a structure of the sort seen in Figure 2. We will say that our target system is composed

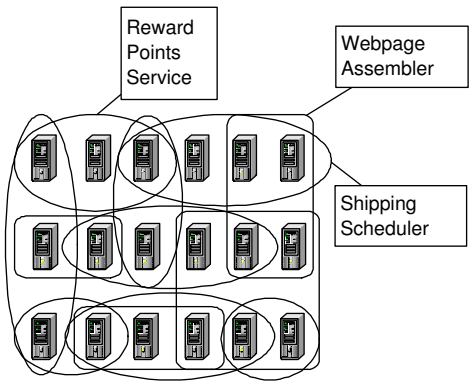

Figure 2. Each service is cloned on multiple nodes and each node runs multiple services, resulting in overlapping replication groups

of *micro-service components* organized into groups. Micro-service designs benefit from componentization - reusability, fault-isolation, and easy versioning; each micro-service can be re-used across multiple applications and developed/upgraded independently. The diagram in Figure 1 is a micro-service design for an e-tailer. A buy order triggers interactions between multiple small services within the back-end datacenter: the user history service logs the object bought, a credit card service processes the purchase, the rewards tracking service adds points to the user's account, etc.

To update a micro-service, the initiating process uses a reliable multicast; this will be automatic in the Tempest system. Moreover, reliable multicast arises when micro-services communicate with one-another using publish-subscribe or event notification architectures.

Scalability, then, has several dimensions. We need to support very large numbers of nodes hosting very large numbers of applications that have been cloned to increase capacity. These many clients will generate a heavy aggregated system load, hence our solution needs to be decentralized: clients should pay for messages they actually send and receive. Merely adding more clients shouldn't cause the system to degrade. We also wish to avoid bursty or unpredictable overhead: such spikes (often associated with failures or other rare events) can temporarily disable many popular platforms and products.

If we translate these goals to multicast communication, we see that the system will need to support large numbers of communication groups, and must expect them to overlap extensively. Even if the individual application doesn't belong directly to enormous numbers of groups, a node hosting many applications might do so, and Ricochet will be shared among the applications. The groups themselves probably won't be huge, since they arise from replicated micro-services, but they might often have a few tens of members.

With respect to all of these goals, we seek solutions with good but probabilistic properties. Ideally, the developer should be in a position to specify the tolerable level of overhead, and should have a way to find out what the consistency implications of such a choice will be. By *consistency*, we mean the following. Suppose that an update is issued (via a multicast) to some service at time t, and yet reads continue to respond on the basis of state data until time $t + \delta$. We would say that the system has a window of inconsistency of length δ. Ricochet actually offers *eventually consistent* delivery: unless a message has a deadline and that deadline expires, Ricochet will eventually deliver it; with high probability, it will do so very rapidly. Messages are delivered at most once, and may arrive out of order; in effect, we assume that application updates are commutative. The natural metric of time-criticality in this model is a measure of the window of potential inconsistency - the time taken for an update to be reflected accurately at all replicas.

This is a relatively weak notion of consistency. Traditional multicast protocols have often sought much stronger properties, such as guarantees of consistent delivery ordering, ordering with respect to changing views of group membership, and failure atomicity [6]. However, recent studies of large datacenters suggest that weak consistency might suffice for time-critical uses.

For example, Jim Gray et. al. [13] argue for a combination of commutative updates with eventual consistency in the context of disconnected applications, and Fox et al. [12] suggest that eventual consistency is sufficient for clustered applications. A similar argument is made by Porcupine [21], a cluster-based mail service designed around eventually consistent semantics. We believe that a large class of time-critical applications fit this consistency model, or can be made to with little effort; for instance, inventory changes are incremental in nature and do not have to arrive in order at replicas.

1.2 Contributions

This paper has the following contributions:

- We propose a new form of multicast in support of micro-service replication - with eventual consistency and commutative updates.
- We review existing real-time communication technologies and show that they do not scale adequately for these settings.
- We describe Ricochet, a unique time-critical multicast primitive that introduces a layer of FEC traffic over multiple IP Multicast groups, allowing for

rapid packet recovery in settings involving thousands of overlapping groups. By using IP Multicast Ricochet ensures that messages travel from source to destination as rapidly as possible; our proactive FEC protocol detects and fills most gaps due to packet loss or failure before the application at the receiver has even discovered the event.

- We evaluate Ricochet on a 64-node rack-style cluster and show that it achieves two orders of magnitude faster recovery of packets than SRM [11], an existing reliable multicast protocol with a number of similarities to Ricochet.
- We link multicast to eventually consistent replication by showing that, in conditions with 1% packet loss, 60% of all updates to micro-services are reflected across replica sets of size 10 within 1 millisecond, 99% within 100 milliseconds, and 100% by 125 milliseconds.

The rest of this paper is structured as follows: Section 2 examines the design choices available for a time-critical multicast primitive, and motivates a new multi-group protocol. Section 3 introduces Ricochet and describes the core algorithm used by it, and provides details of a Java implementation. Section 4 provides an evaluation of Ricochet. Section 5 gives a summary of related work, and Section 6 concludes the paper.

2 Time-Critical Multicast in Datacenters

The multicast problem we've posed differs from prior problem statements in its strong emphasis on time-critical delivery in systems with large numbers of heavily overlapping but relatively small communication groups. As mentioned earlier, we believe this pattern of communication will be important in datacenters. It is not, however, a pattern of communication or requirements of "universal" value, nor does our solution appear to be one that would displace prior solutions in settings with other objectives. In fact, our protocol will turn out to make other sacrifices in the interest of faster delivery: the solution is not designed to scale to large numbers of recipient processes. Moreover, it has not been optimized for exceptionally high data rates; as mentioned earlier, we lose about a factor of ten in peak throughput compared to a different system also being developed in our group (QSM), but gain several orders of magnitude improvements in the latency metrics of interest here. We've come to believe that time criticality and peak throughput conflict: by optimizing for one objective, we've accepted unavoidable sacrifice in others.

Multicast is a mature, well-researched field, and before designing a new protocol it makes sense to think of retrofitting an existing reliable multicast protocol for use in time-critical settings. However, multicast research has traditionally focused on scaling different properties - such as throughput - along different dimensions, notably the number of receivers in a single group. Moreover, it has traditionally focused on consistency semantics stronger than the ones we adopt here – for example, the virtual synchrony model [5], or the Paxos model proposed by Lamport [17].

For example, the Spread system [2] implements virtual synchrony using a central set of agents as a server through which all messages are relayed. Spread latencies necessarily grow as a function of the numbers of clients in the system and the aggregate multicast rates; particularly noticeable is the potential for interference, whereby the timing characteristics of a multicast from process p_1 to p_2 are affected by traffic in other groups to which neither belongs. Data-Aware Multicast [3] focuses on exploiting hierarchical relationships between publish-subscribe groups, but without special attention to low latency; indeed, latency can be quite high if a process finds itself in an obscure corner of the hierarchy. Lightweight Process Groups, again a virtual synchrony solution, involves the substitution of a single, large transport group for a collection of smaller groups [15]; broadcasts in the large group are filtered to simulate multicast in the small groups. These approaches suffer from interference, similar to Spread.

TAO DDS [1] is an implementation of DDS (a specification for a real-time CORBA communications bus), offering the easy-to-use and flexible abstraction of QoS-aware topic-based publish-subscribe to developers. It consists of a central server that orchestrates the creation of direct point-to-point TCP channels between a sender and each of the receiver processes in a publish-subscribe group. The architecture is such that one TCP connection will be established for each $(sender, receiver, group)$ triple. Thus in the scenarios of interest here, where there could be many receivers in some groups, and where there will be many heavily overlapped groups, a sender may find itself managing thousands of TCP connections, each with its own acknowledgement and flow control protocols. A single notification sent to a group of, say, 25 nodes would result in 25 separate TCP writes, and at the underlying operating system level, these will contend not just with one another but also with traffic on all the other connections; both phenomena introduce latency and variability in delivery roughly linear in the number of clients. As demonstrated below, one can do much better than this using IP multicast.

The Ricochet protocol takes considerable inspiration from Scalable Reliable Multicast [11], a data distribution protocol originally created to support collaboration over the Internet White Board and multicast on the Inter-

net mbone. SRM, uses a very similar reliability model to the one we adopt here: a best-effort delivery mechanism that seeks to recover lost data rapidly or not at all, notifying the application if an unrecoverable loss is detected. Ricochet, like SRM, makes extensive use of IP multicast; SRM does this for scalability, while we do it because IP multicast is the fastest data path available from a multicast source to a set of recipients. Where we part company from SRM is that SRM assumes that any single multicast will have a huge number of destinations. Ricochet assumes that the datacenter may have a huge number of nodes running our protocols, but that any given multicast targets a relatively small set of nodes - perhaps as many as 25 or even 100, but not thousands. Moreover, SRM doesn't worry very much about low-latency communication; in their setting, latency could easily be concealed from the user by buffering data in the playback system. Ricochet, of course, targets applications for which low-latency is a big win.

These considerations motivate our solution:

- We use IP Multicast to deliver data rapidly to as many recipients as possible
- We employ proactive Forward Error Correction between multicast receivers to allow rapid reconstruction of lost packets.
- Our solution exploits knowledge of the patterns of overlapping groups in the system to utilize FEC overhead intelligently, by planning the composition and propagation of repair packets.

2.1 The Timeliness of (Scalable) Reliability Schemes over IP Multicast

Reliable multicast protocols typically consist of three logical phases: *transmission* of the message, *discovery* of packet loss, and *recovery*. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols, the time taken to discover packet loss dominates recovery time heavily in the kind of settings we are interested in. The key insight is that the discovery time of reliable multicast protocols is usually inversely dependent on *data rate*: for existing protocols, the rate of outgoing data at a single sender, in a single group. Existing schemes for reliability in multicast can be loosely divided into the following categories:

ACK/timeout: RMTP [18], RMTP-II [19]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [18]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger time-outs at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

Gossip-Based: Bimodal Multicast [7], lpbcast [10]. Receivers periodically gossip histories of received packets with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery time is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [7]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact. Thus Bimodal Multicast is bimodal in several senses: delivery probability (the origin of the protocol name) but also delivery latency, and a loss can result in delays of several hundred milliseconds.

NAK/Sender-based Sequencing: SRM [11]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery time is thus proportional to the inter-send time at any single sender within a single group - a receiver can't discover a loss until it receives the next packet from the same sender - and consequently depends on the sender's data transmission rate within the designated group. In SRM, Lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

Sender-based FEC [16, 20]: Forward Error Correction schemes involve multicasting redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating c repair packets for every r data packets, such that any r of the combined set of $r + c$ data and repair packets is sufficient to recover the original r data packets; we term this (r, c) parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC

data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out r data packets to that group. Traditionally, sender-based FEC schemes tend to use complex encoding algorithms to obtain multiple repair packets, placing computational load on the sender. However, faster encodings have been proposed recently, such as Tornado codes [8], which make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, Internet Radio.

Of the above schemes, ACK-based protocols are intrinsically unsuited for time-critical multi-sender settings, while sender-based sequencing and FEC limit discovery time to inter-send time at a single sender within a single group. Ideally, we would like discovery time to be independent of inter-send time, and combine the scalability of a gossip-based scheme with the tunability of FEC. In earlier work, we introduced **Receiver-based FEC** in the context of the Slingshot protocol [4]. In Slingshot, receivers generate FEC packets from incoming data and exchange these with other randomly chosen receivers. Since FEC packets are generated from *incoming* data at a receiver, the timeliness of loss recovery depends on the rate of data multicast in *the entire group*, rather than the rate at any given sender, allowing scalability in the number of senders to the group.

Slingshot was developed for single-group settings, recovering from packet loss in time proportional to that group's data rate. Our goal with Ricochet is to achieve recovery time dependent on the rate of data incoming at a node *across all groups*. Essentially, we want recovery of packets to occur as quickly in a thousand 10 Kbps groups as in a single 10 Mbps group, allowing applications to divide node bandwidth among thousands of multicast groups while maintaining time-critical packet recovery.

3 The Ricochet Protocol

In Ricochet, each node belongs to a number of groups, and receives data multicast within any of them. The basic operation of the protocol involves generating FEC packets from incoming data and exchanging them with other randomly selected nodes. Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 3 shows the structure of these two packets types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from r data packets allows recovery of one of them, if all the other $r-1$ data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets.

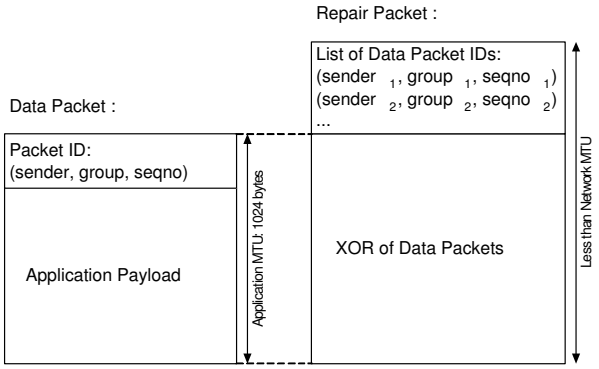

Figure 3. Ricochet Packet Structure

Each Ricochet group has an (r, c) rate-of-fire parameter that determines the amount of overhead expended to ensure time-critical delivery of its data. All the groups that a particular node belongs to must have the same r component; however, c can vary from group to group, allowing tunable time-criticality. The rate-of-fire parameter determines the fraction of lost packets that are recovered using FEC traffic; in almost all of the remaining cases, *discovery* of loss takes place through FEC traffic, but the existence of multiple losses in a single FEC packet makes immediate recovery impossible. In such cases, the packet is retrieved using a negative acknowledgment to either the sender of the repair packet, or the sender of the original data packet.

At the core of Ricochet is a probabilistic engine running at each node that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind this engine is the notion that repair packets sent by a node to another node can be composed from data in any of the multicast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur within time that's inversely proportional to the aggregate rate of data in all these groups. Figure 4 illustrates this idea: n_1 has groups A and B in common with n_2, and hence it can generate and dispatch repair packets that contain data from both these groups. n_1 needs to wait until it receives 5 data

packets in either A *or* B before it sends a repair packet, allowing faster recovery of lost packets at n_2.

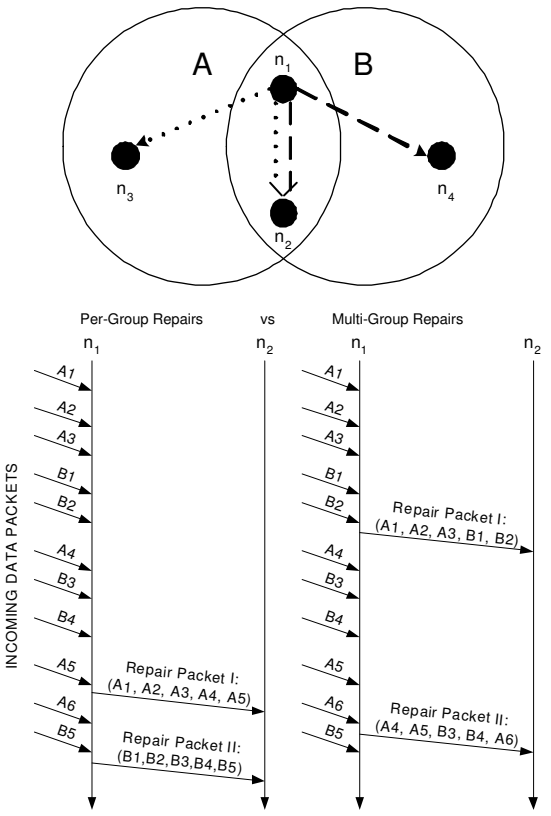

Figure 4. Optimizing Receiver-based FEC in multiple groups

While combining data from different groups in outgoing repair packets drives down recovery time, it destroys the coherent tunability that receiver-based FEC provides. The *rate-of-fire* parameter in single group receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every r data packets, c repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for Ricochet is to combine repair traffic for multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. Coherent, Tunable Per-Group Overhead: For every data packet that a node receives in a group with rate-of-fire (r, c), it sends out *an average of* c *repair packets* including that data packet to other nodes in the group.
2. Randomness: Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

Ricochet supports overlapping groups with the same r component and different c values in their rate-of-fire parameter. In Ricochet, the rate-of-fire parameter is translated into the following guarantee: For every data packet d that a node receives in a group with rate-of-fire (r, c), it selects an average of c nodes from the group randomly and sends each of these nodes exactly one repair packet that includes d. In other words, the node sends an average of c repair packets containing d to the group. In the following section, we describe the algorithm that Ricochet uses to compose and dispatch repair packets while maintaining this guarantee.

3.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same code runs at every node - and hence, we can describe its operation from the vantage point of a single node, n_1.

3.1.1 Regions

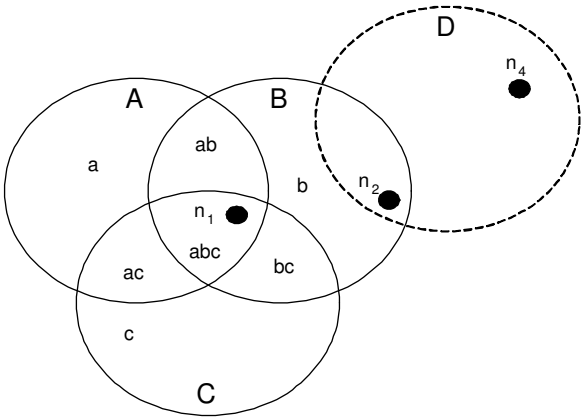

Figure 5. Regions: n_1 belongs to groups A, B, C; it divides them into disjoint regions abc, ab, bc, ac, a, b, c

The Ricochet engine running at n_1 divides n_1's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that n_1 belongs to. Figure 5 shows the regions in a hypothetical system, where n_1 is in three groups, A, B and C. We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e, abc is a region formed by the intersection of A, B and C. In our example, the neighborhood set of n_1 is carved into seven regions: abc, ac, ab, bc, a, b and c, essentially the power set of the set of groups involved

(readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case, since we are only concerned with non-empty intersections: see Section 3.1.4). Note that n_1 does not belong to group D and is oblivious to it; it observes n_2 as belonging to region c, rather than cd, and is not aware of n_4's existence.

3.1.2 Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, Ricochet selects targets randomly from *each region*. The number of targets selected from a region is set such that:
1. It is proportional to the size of the region
2. The total number of targets selected, across regions, is equal to the c value of the group
Hence, for a given group A with rate-of-fire (r, c_A), the number of targets selected by Ricochet in a particular region, say abc, is equal to $c_A * \frac{|abc|}{|A|}$, where $|x|$ is the number of nodes in the region or group x. We denote the number of targets selected by Ricochet in region abc for packets in group A as c_A^{abc}. Figure 6 shows n_1 selecting targets for repairs from the regions of A.

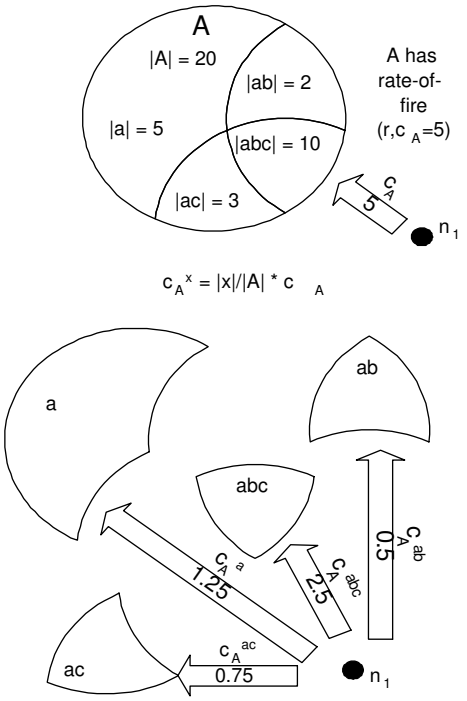

Figure 6. n_1 selects proportionally sized chunks of c_A from the regions of A

Note that Ricochet may pick a different number of targets from a region for packets in a different group; for example, c_A^{abc} differs from c_B^{abc}. Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size c has to be selected from the group, selecting targets from regions corresponds to *stratified sampling*, an existing technique from statistical theory.

3.1.3 Why select targets from regions?

Selecting targets from regions instead of groups allows Ricochet to construct repair packets from multiple groups; since we know that all nodes in region ab are interested in data from groups A and B, we can create composite repair packets from incoming data packets in both groups and send them to nodes in that region.

Single-group FEC [4] is implemented using a simple construct called a *repair bin*, which collects incoming data within the group. When a repair bin reaches a threshold size of r, a repair packet is generated from its contents and sent to c randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems trivial; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size r, it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups A, B, and C, how many targets should we select from region abc for this repair packet - c_A^{abc}, c_B^{abc}, or c_C^{abc}? One possible solution is to pick the maximum of these values. If $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$, then we would select c_A^{abc}. However, a data packet in group B, when added to the repair bin for the region abc would be sent to an average of c_A^{abc} targets in the region; resulting in more repair packets containing that data packet sent to the region than required (c_B^{abc}), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group B than required by its (r, c_B) value; a similar argument holds for data packets in group C as well.

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups A and B than required, resulting in a lower fraction of packets recovered from FEC traffic. To rectify this we send the additional compensating repair packets to the region abc from the repair bins for regions a and b. The repair bin for region a would select $c_A^{abc} - c_C^{abc}$) destinations, on an average, for every repair packet it generates; this is in addition to the c_A^a destinations it selects from region a.

A more sophisticated version of the above strategy involves iteratively obtaining the required repair pack-

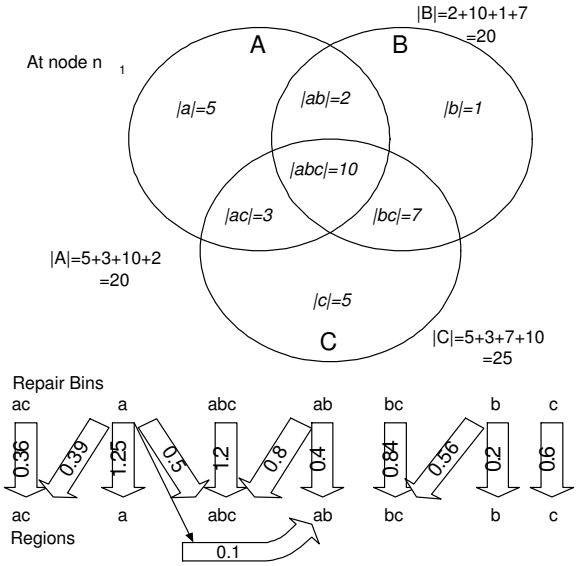

Figure 7. Mappings between repair bins and regions: For e.g., the repair bin for ab selects 0.4 targets from region ab and 0.8 from abc for every repair packet.

ets from regions involving the remaining groups; for instance, we would have the repair bin for ab select the minimum of c_A^{abc} and c_B^{abc} - which happens to be c_B^{abc} - from abc, and then have the repair bin for a select the remainder value, $c_A^{abc} - c_B^{abc}$, from abc. Algorithm 1 illustrates the final approach adopted by Ricochet, and Figure 7 shows the output of this algorithm for an example scenario. Note that a repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

Algorithm 1 Algorithm for Setting Up Repair Bins

1: **Code at node** n_i**:**
2: **upon** Change in Group Membership **do**
3: **while** L not empty {*L is the list of regions*} **do**
4: Select and remove the region $R_i = abc...z$ from L with highest number of groups involved (break ties in any order)
5: Set $R_t = R_i$
6: **while** $R_t \neq \epsilon$ **do**
7: set c_{min} to $min(c_A^{R_t}, c_B^{R_t}...)$, where {A,B,...} is the set of groups forming R_t
8: Set number of targets selected by R_i's repair bin from region R_t to $c_m in$
9: Remove G from R_t, for all groups G where $c_G^{R_t} = c_m in$

3.1.4 Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that n_1 is part of. The algorithm has complexity $O(I \cdot d)$, where I is the number of populated regions (i.e, with one or more nodes in them), and d is the maximum number of groups that form a region. Note that I at n_1 is bounded from above by the cardinality of the set of nodes that share a multicast group with n_1, since regions are disjoint and each node exists in exactly one of them. d is bounded by the number of groups that n_1 belongs to.

3.2 Implementation Details

Ricochet is implemented in Java. We assume that each node executes a single JVM, containing all the micro-services running on the machine. Further, we assume that there are no other processes running on the node at a priority higher than that of the JVM's. We believe these assumptions are reasonable given the administrative homogeneity of the typical datacenter.

3.2.1 Repair Bins

A Ricochet repair bin is a lightweight structure holding an XOR and a list of data packets, and supporting an add operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in the example in Figure 7, the repair bin for abc has to select 1.2 targets from abc, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

3.2.2 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [6]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and

contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the algorithm described previously. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at n_1 - and consequently the group and intersection sizes - does not include n_1 itself.

3.2.3 Membership and Failure Detection

The Ricochet infrastructure includes Group Membership (GMS) and Failure Detection (FD) services, which execute on high-end server machines. If the GMS receives a notification from the FD that a node has failed, or it receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s). The GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node n_{55} joins group A, the update sent by the GMS to every node in A would be a 3-tuple: *(Join, A, n_{55})*. Individual Ricochet nodes process these updates to construct multi-group views relative to their own membership. Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this is easily done by partitioning the service on group id. For instance, one machine is responsible for groups A and B, another for C and D, and so on. Similarly, the FD is partitioned on a topological criterion; one machine on each rack is responsible for monitoring other nodes on the rack by pinging them periodically. The GMS is made fault-tolerant by replicating each partition on multiple machines, using a strongly consistent protocol like Paxos. The FD has a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain. While this semi-centralized solution would be inappropriate in a high-churn environment, such as a WAN or ad-hoc setting, we believe that it is reasonable in datacenters, where membership is relatively stable.

3.2.4 Performance

Since Ricochet creates FEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to; instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to c (from the rate-of-fire) in the worst case - which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

3.2.5 Buffering and Loss Control

One of the major arguments against FEC mechanisms is that losses are typically caused by buffer overflows and hence bursty, rendering error recovery packets useless beyond a threshold burst size. This is certainly a valid argument in WAN settings, where loss occurs at intermediary routers and is beyond the control of applications running on end hosts. However, datacenter networks have flat routing structures, with no more than two or three network hops on any end-to-end path, and are typically over provisioned and of high quality; hence, packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are cheap and inexpensive; even with high-capacity network hardware, the commodity operating system (Windows or Linux) often drops packets due to traffic spikes or high-priority threads occupying the CPU. In practice, draining input packets from the kernel buffer into application space often suffices to control the problem.

Ricochet maintains an application-level buffer within the JVM. A thread running at the highest Java priority picks up data from the kernel and places it in the application-level buffer; hence, we can reasonably expect little loss to occur at the kernel level, unless our assumption of the JVM being the highest priority process is violated. When the application buffer's threshold is reached, we drop a randomly selected packet from the buffer and accommodate the new packet instead. In practice, this results in a sequence of *almost* random losses from the buffer, which are easy to recover from using FEC traffic. In our current implementation, we set the size of this buffer to be 1024 packets, or 1 MB.

4 Evaluation

Our evaluation of Ricochet was on a 64-node rack-style cluster, comprised of 4 racks of 16 blade-servers each, interconnected via two levels of switches. Each blade-server has a single 1.33 Mhz Pentium processor and 512 MB RAM. In all the experiments, a 1% rate of packet loss is simulated by dropping with uniform probability 0.01 at end-host buffers. A detailed evaluation

of the tunability and behavior of receiver-based FEC in a single group has been provided in [4], and hence we focus exclusively on multi-group settings. Aside from Figure 8, the graphs in this section involve all 64 nodes in the cluster. Error bars shown for averages indicate the minimum and maximum value over ten runs.

4.1 The Two-Group Case

First, we show that the basic technique used by Ricochet succeeds in driving down recovery delays while maintaining per-group overhead and recovery characteristics. Figure 8 involves 16 nodes and 2 groups; group A, with rate-of-fire $(8, 5)$, and group B, with rate-of-fire $(8, 3)$. The x-axis denotes the number of nodes belonging to both groups. When the x value is zero, each node belongs to exactly one of the two groups, and the packet recovery characteristic for each group is determined by its rate-of-fire; for instance, in group A, 97% of lost packets are recovered via FEC, and 37% of all traffic in that group is repair overhead, numbers which match the single-group results given in [4]. As we make more nodes join both groups, these nodes divide their membership into three regions, and send other nodes in the region ab repair packets containing data from both A and B. When $x = 16$, all nodes belong to the region ab formed by the intersection the two groups, hence every repair packet includes data from both groups. Note that recovery delay goes down without disrupting per-group overhead and recovery percentage. Recovery delay is defined as the time between the initial multicast and the recovery of the lost packet.

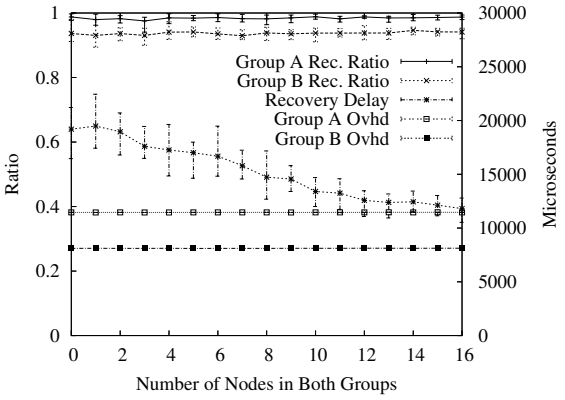

Figure 8. Two Group Example: As more nodes occupy the overlap region, recovery delay goes down, while per-group overhead and recovery rate are maintained. Recovery Delay is plotted on the right y-axis.

4.2 Scalability

This experiment is key to verifying that Ricochet provides time-critical packet recovery that scales in the number of groups in the system. The workload used in this experiment involves each node sending a packet every ten milliseconds in any one of the groups it belongs to. We set the average group size to 10 - which we believe to be a reasonable number if micro-services are replicated aggressively within the datacenter for availability and caching purposes. Consequently, a sending throughput of 100 packets/sec at each node results in an average receive throughput of 1000 packets per second, as each sent packet is received at an average of ten nodes. We use the following formula for increasing the number of groups: $d * n = g * s$, where d is the degree of membership, i.e the number of groups each node joins, n is the number of nodes in the system, g is the total number of groups, and s is the average size of a group. Setting s to 10 and n is 64, changing d is sufficient to examine the scalability of Ricochet in the number of groups, per node and overall. The main reason for using this formula is to allow the random creation of groups of size 10, such each node is in a fixed number of groups.

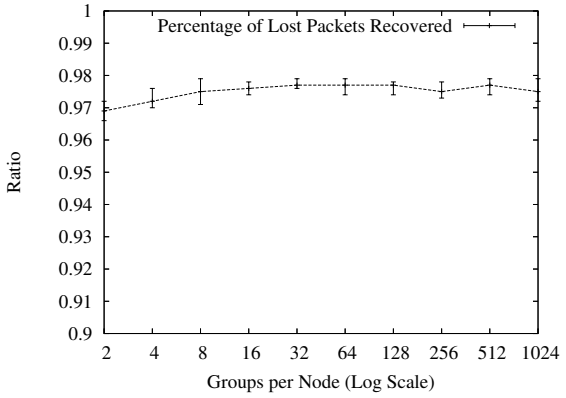

Figure 9. Fraction of lost packets recovered via FEC

Accordingly, the x-axis in figures 9 and 10 is equivalent to d, the number of groups each node belongs to. In our setup, we had each node join d randomly chosen groups from a set of g groups, computing g using the formula above; for example, if each node joins $d = 8$ groups, the total number of groups in the system is $g = \frac{d*n}{s} = 52$. Each group has rate-of-fire $(8, 5)$. Figure 9 plots the average percentage of lost packets recovered via the FEC mechanism; it shows that this percentage stays constant as we increase the degree of membership from 2 to 1024. Between 2 and 3 percent of lost packets are not recoverable using the FEC mechanism; these are later recovered by sending NAKs back to the sender. Figure 10 shows the average delay in recovering

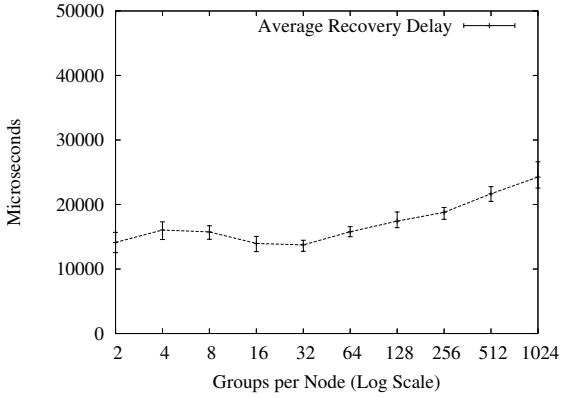

Figure 10. Average Delay in Recovering Lost Packets

lost packets via FEC. Note that the x-axis in both these graphs is in log-scale, hence a straight line increase is actually *logarithmic* with respect to the number of groups, and represents excellent scalability. The slow increase in packet recovery delay as we move towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section. In the context of a 1% loss rate, these two graphs together indicate that around 99.97 of all packets are delivered at an average of 25 milliseconds. To observe the distribution of recovery delays around the average, we provide a histogram for a single run involving each node joining 64 groups.

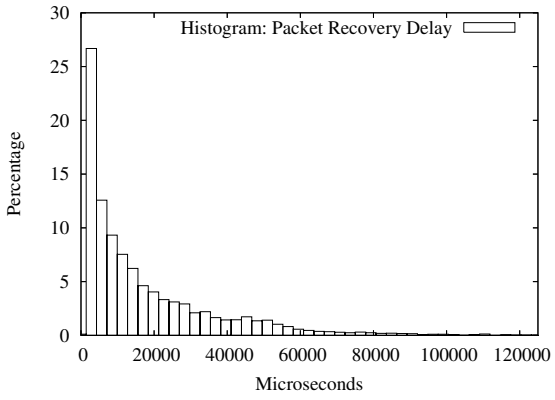

Figure 11. Distribution of Packet Recovery Delay

4.2.1 Comparison with SRM

To compare Ricochet to existing multicast techniques, we run Scalable Reliable Multicast - the NAK based reliable multicast protocol mentioned previously - in exactly the same settings. To run SRM in multiple groups at a node[2], we executed a separate SRM process for each multicast group that the node joined. Figure 12 shows the recovery characteristics of SRM as we increase the number of groups per node from 1 to 128; beyond 128 groups, the number of processes per node made it difficult to extract coherent performance measurements. SRM recovery is more than two orders of magnitude slower than Ricochet in two groups, and this difference expands by an additional order of magnitude at 128 groups per node. The SRM discovery curve on the graph shows the average time within which packet loss is discovered at the affected receiver through the NAK mechanism; as expected, it doubles as the number of groups per node is doubled and the rate of incoming data per group at a node gets halved. The difference between discovery of packet loss and recovery in SRM - roughly 4 seconds - seems to arise from timing assumptions built into the protocol, and could certainly be reduced by tuning its internal parameters. In the limit, however, SRM's recovery is bounded below by its discovery curve, which is around 180 milliseconds for 2 groups, and grows linearly with the number of groups per node, exceeding Ricochet by 3 orders of magnitude at 128 groups.

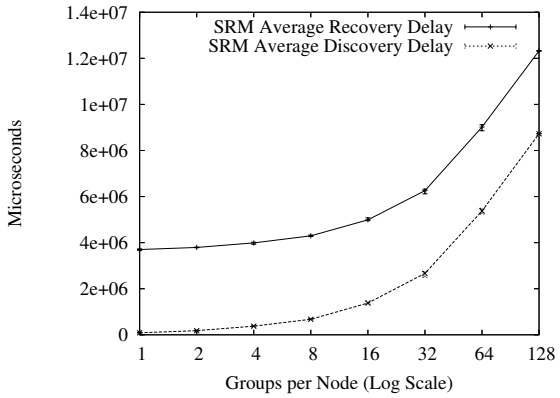

Figure 12. SRM: Average Delay in Discovery and Recovery of Lost Packets

4.2.2 Protocol Performance

Even though the recovery characteristics of Ricochet scale well in the number of groups per node, it is important that the computational overhead imposed by the protocol at any node stays manageable, given that time-

[2]SRM was not designed for this configuration. However, it remains a well known and popular protocol, was designed to scale well, and has significant architectural similarity to Ricochet. For these reasons, we felt that comparison isn't unreasonable.

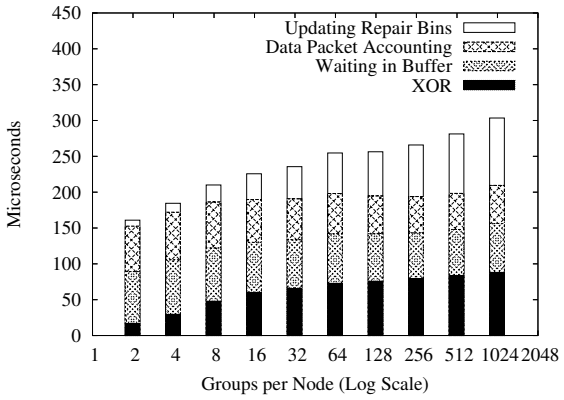

Figure 13. CPU time per incoming Data Packet

critical applications are expected to run over it. Figure 14 shows the average number of XORs taken for each incoming data packet; as stated in Section 3.2.4, the number of XORs stays under 5 - the value of c from the rate-of-fire - as we increase the number of groups per node. Figure 13 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a logarithmic x-axis shows that per-packet processing time increases logarithmically with respect to the number of groups; doubling the number of groups results in a constant increase in processing time. The increase in processing time in the latter half of the graph is due to the increase in the number of repair bins as the number of groups increases. While we did not address this problem in our current implementation - 1024 was considered adequate scalability in the number of groups we could easily remove this scalability "hit" by creating bins only for occupied regions. With the current implementation of Ricochet, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second.

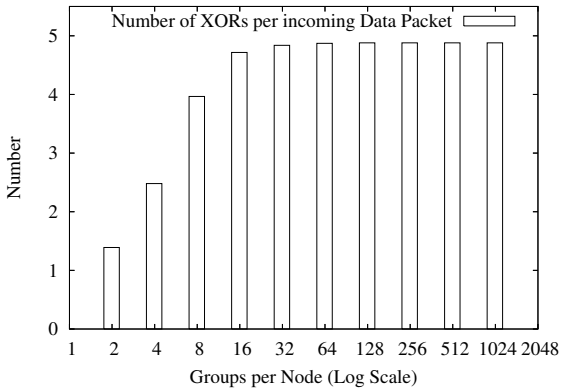

Figure 14. XORs per incoming Data Packet

4.3 Consistency Properties

Thus far, we have seen that Ricochet achieves rapid recovery from packet loss in the presence of large numbers of groups. However, recovery time from packet loss is a multicast metric; to get an idea of what these numbers signify in the context of a replicated micro-service, we plot a histogram of the *inconsistency windows* of updates in Figure 15. An inconsistency window for an update is defined as the period during which a query to the replicated micro-service can return a value that does not reflect the update; its length is calculated as the difference in time between the first and last deliveries of the update, across replicas. Around 65 percent of all updates are reflected at all replicas within 1.25 milliseconds, 90 by 18 milliseconds, and 99 percent within 77 milliseconds. Full consistency is achieved in all cases within 125 milliseconds, off the right edge of this graph.

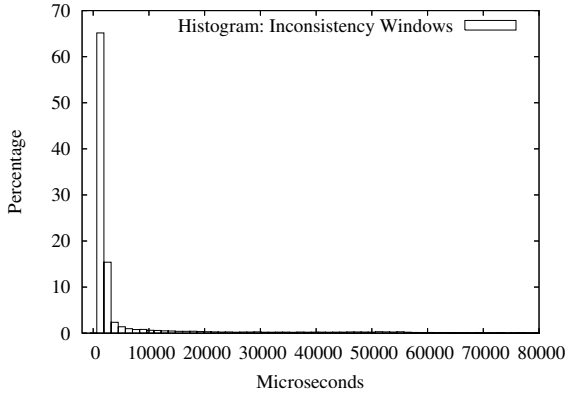

Figure 15. Histogram of Inconsistency Periods for updates: 64 groups per node

5 Related Work

The Ninja Project at Berkeley [12], [14] describes an extensive architecture for running internet services within clusters. While replication of components is mentioned as a fundamental building block for ensuring availability and fault-tolerance, the work focuses on the execution environment within the cluster; our work is at a level below, and Ricochet can potentially be used as a replication primitive within any clustering framework.

Multicast can be used via many different abstractions, such as process groups or publish-subscribe [6]; our focus is on the technology used for implementing many-to-many communication, and is orthogonal to the choice

of abstraction. We discussed the existing multicast literature in depth in Section 2.

6 Conclusion

Ricochet is a new multicast protocol that achieves time-critical eventual consistency and scales well, particularly in the numbers of communication groups to which a node belongs. The protocol achieves several orders of magnitude lower delivery latency, with tunably high probability, than SRM, a prior protocol with some similarities to ours. Our plan is to incorporate the protocol into Tempest, a new drag-and-drop tool and runtime environment to assist developers in building componentized time-critical services that scale well in clusters or datacenters.

7 Acknowledgments

We would like to thank Tudor Marian for his invaluable help in setting up the cluster, and for his comments on the paper. We would like to thank Robbert van Renesse and Krzys Ostrowski for their comments. Douglas Schmidt and Paul Calabrese provided useful information about the state-of-the-art in DDS systems.

References

[1] Object computing inc. tao dds. `http://www.ociweb.com/products/dds`, 2005.

[2] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins University, 2004.

[3] S. Baehni, P. T. Eugster, and R. Guerraoui. Data-aware multicast. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 233, Washington, DC, USA, 2004. IEEE Computer Society.

[4] M. Balakrishnan, S. Pleisch, and K. Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*, 2005.

[5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[6] K. P. Birman. *Reliable Distributed Systems*. Springer Verlag, 2005.

[7] 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.

[8] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 56–67, New York, NY, USA, 1998. ACM Press.

[9] 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.

[10] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[11] 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 Trans. Netw.*, 5(6):784–803, 1997.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 78–91, New York, NY, USA, 1997. ACM Press.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM Press.

[14] S. D. Gribble, M. Welsh, E. A. Brewer, and D. E. Culler. The multispace: An evolutionary platform for infrastructural services. In *USENIX Annual Technical Conference, General Track*, pages 157–170, 1999.

[15] K. Guo and L. Rodrigues. Dynamic light-weight groups. In *ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 33, Washington, DC, USA, 1997. IEEE Computer Society.

[16] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[17] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[18] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

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

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

[21] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1999. ACM Press.