The Power of Indirection: Achieving Multicast Scalability by Mapping Groups to Regional Underlays

Krzysztof Ostrowski
Cornell University

Ken Birman
Cornell University

Amar Phanishayee
Cornell University

Abstract

Reliable multicast is a powerful primitive, useful for data replication, event notification (publish-subscribe), fault-tolerance and other purposes. Yet many of the most interesting applications give rise to huge numbers of heavily overlapping groups, some of which may be large. Existing multicast systems scale scale poorly in one or both respects. We propose the *QuickSilver Scalable Multicast protocol* (QSM), a novel solution that delivers performance almost independent of the number of groups and introduces new mechanisms that scale well in the number of nodes with minimal performance and delay penalties when loss occurs. Key to the solution is a level of indirection: a mapping of groups to regions of group overlap in which communication associated with different protocols can be merged. The core of QSM is a new regional multicast protocol that offers scalability and performance benefits over a wide range of region sizes.

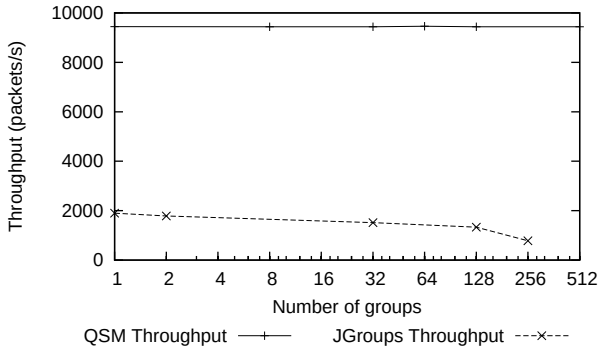

Figure 1: Scalability of multicast rate with the number of groups. All groups completely overlap on the same 25 members. A single source sends messages in all the different groups in a round-robin fashion.

1 Introduction

1.1 Motivation

In this paper we report on a new multicast substrate, QSM, targeting very large deployments of client computers that communicate using publish-subscribe or event notification architectures. Were assuming that there may be thousands of users, many running Windows, and distributed over a WAN. For the present paper, our goal is simply to support the highest possible throughput[1] and "best effort" reliability. In future work, we'll extend QSM with a stackable protocol extension architecture supporting protocols that customize reliability or other properties on a per-group (e.g. per-topic) basis.

Existing reliable multicast support for multiple groups falls into two categories: "lightweight group" solutions that map application groups into broadcasts in an underlying group spanning all receivers, then filter at the receivers to drop unwanted messages, and those that run separate protocols for each group independently.

Lightweight groups work best in smaller systems; with scale, data rates in the underlying group become excessive, receivers are presented with huge numbers of undesired packets that waste resources and fill up receive buffers, and the overload triggers increased packet loss. The Spread system [1] works around this using lightweight groups in combination with an agent architecture: clients relay multicasts to a small group of agents; these broadcast each message, filter them on reception, and then relay matching messages back to the client, but the approach introduces two extra message hops, and the agents experience load linear in the system size and multicast rate.

Running a separate protocol for every group brings different issues. Such an approach incurs overhead lin-

[1] In other work (the Tempest system and its Ricochet multicast protocol), our group is looking at clustered applications with time-critical behavior; the architecture is completely different.

ear in the number of groups for ACKs, NAKs and other control traffic. Moreover, contention between groups for communication resources, both within individual nodes and on the wire, emerges as an issue. To quantify such effects, we measured the performance of JGroups [2], a popular communication package that supports multiple groups. We configured JGroups to provide only weak reliability guarantees. Nonetheless, as shown in Figure 1, the overhead associated with running multiple protocols in JGroups decreases the achievable aggregated throughput by 6% with 2 groups, 20% with 32 groups and almost 60% with 256 groups.

Here, we explore a third option. QSM runs per-group protocols over a lower layer implementing protocols on a per-region basis. Regions are designed to have a fairly regular structure, and this lets us innovate in the regional protocols. The performance so obtained is strong across the board, even with very large numbers of groups. As seen in Figure 1 and explored further below, we also achieve low latency and excellent scalability in group size.

1.2 Group Overlap and its Implications

Our goals emerge from communication patterns seen in demanding real-world multicast systems. One of us developed the multicast technology used in the current New York and Swiss Stock Exchange systems, the French Air Traffic Control System, and the US Navy AEGIS warship [4], and we are in dialog with developers of very large data centers, such as Amazon, Google, Yahoo! and Lockheed Martin. If developers of such systems are forced to work over an unreliable event notification or publish-subscribe architecture, they typically implement stronger properties by hand, a difficult, error-prone and inefficient approach. In contrast, by treating these in terms of multicast to large numbers of groups, we take a major step towards offering customizable communication properties and protocol stacks on a per-group basis. This opens the door to a whole class of applications that previously could not be supported.

We assume a system with a large number of nodes, each belonging to a number of potentially overlapping groups (left side of Figure 2). Nodes multicast asynchronously (without waiting for replies) and may do so at a high rate; a single node may multicast to multiple groups concurrently. We want to optimize for the highest possible throughput while keeping latency reasonably low, and achieving a simple reliability property similar to that in systems such as SRM or RMTP. Specifically, the system should attempt to deliver every pending message still buffered at the sender or another node, to all nodes that have not crashed or left the group to which the message was addressed.

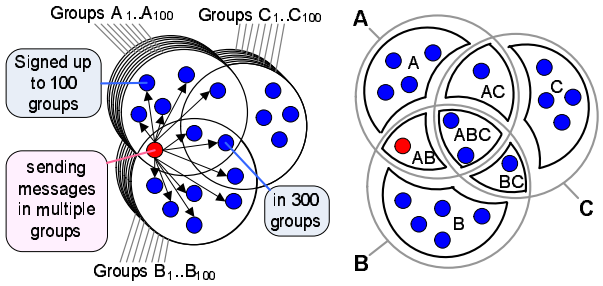

Figure 2: Left: a pattern of overlapping groups. Right: regions of overlap.

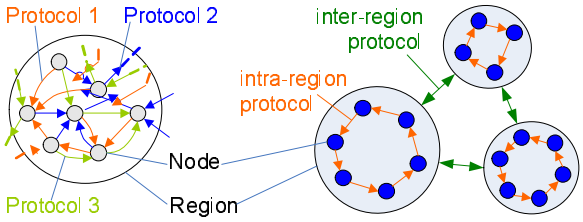

Figure 3: Left: In a typical system, nodes that belong to multiple groups participate in multiple protocols. Right: Our system splits protocols into two levels: a single "bottom half per region, and a per-group "upper half.

If we rule out lightweight group approaches as fundamentally non-scalable, the scenario depicted in Figure 2 poses real problems for existing technologies. Since each group is operated independently, the sending node will multicast data separately in a large number of groups, perhaps using a separate IP multicast group in each. Each group will implement its own flow control and reliability protocol, hence a node that participates in many groups independently exchanges ACKs, NAKs and other control messages with its counterparts. Not only does traffic rise linearly in the number of groups, but the communication pattern sabotages efforts to prevent ACK and NAK implosion, typically using aggregation on trees. For example, in Figure 3 (left) we've drawn multiple superimposed acknowledgement trees of the sort used in RMTP [10] and SRM [6]. Unless tree creation is coordinated across groups, each node will communicate with far more neighbors than in a single-group configuration.

1.3 Our Approach

Key to our approach is the recognition that a system with large numbers of overlapping groups typically has a much smaller number of *regions of overlap*. In the example on Figure 2 (right), 18 nodes have subscribed to 300 groups but these overlap in just 7 regions. Regions

of overlap have a few important properties that we refer to as *fate and interest sharing*, and exploit in our system.

Interest sharing arises when nodes in a region receive the same multicast messages, which makes it natural to consider message batching and similar tactics. If regions are large scalability becomes an issue, and it makes sense to create IP multicast groups and attempt loss recovery at a regional granularity. In the example in Figure 2, the transmitting node, instead of multicasting in 200 groups, might do so in 6, and can pack small messages targeted to a given region in larger packets, increasing network utilization and reducing the frequency with which receiver overheads. Latency suffers, but as noted earlier, QSM isnt optimized to minimize latency.

Fate sharing captures a related observation: nodes in a region experience the same workload, suffer from the same bursts of traffic, miss the same packets dropped at the sender, which makes it desirable to implement flow control and certain parts of the loss recovery mechanisms at this level. If we multicast at the granularity of regions, nodes will acknowledge the same messages and can recover missing data on a peer-to-peer basis.

Suppose that all members of a region drop some packet. Rather than having the nodes individually seek a retransmission from the sender, it makes sense to report their aggregated loss in a manner similar to recovery in RMTP. Similarly, it makes sense to coordinate control traffic relating to flow control on a regional basis.

Jointly, such considerations argue that if we view nodes as members of regions rather than of groups, and design regions to achieve high levels of interest and fate sharing, we can gain efficiencies not available at the group level. We can also avoid the inefficiency that limits lightweight group schemes: in our terminology, lightweight group schemes map large numbers of groups to a single enclosing region, but often end up with mappings in which there is little interest and fate sharing.

The efficiency of this approach depends on the number and sizes of regions, which in turn is a function of system structure. With lots of small regions, we end up with large numbers of small IP multicast groups and senders may have to issue a great many IP multicasts to send a single group multicast; this is clearly not desirable. In the extreme case a node would need to send a separate packet for each destination. Yet at the other extreme, where all nodes are members of the same groups and the whole system is covered by a single region, performance can be virtually independent of the number of groups. The greater the degree of group overlap, the easier it will be to find regions that work well.

We see the selection of regions as a machine assisted task in which the system designer plays a significant role. Automating this task may ultimately be possible, but is not a current goal of our work.

Designer-assisted region discovery seems entirely practical. Consider, for example, publish-subscribe systems in which subjects are mapped to groups. Each instance of a given application, operated by a similar kind of user, is likely to result in similar patterns of subscriptions and hence extensive overlap. Equities traders who trade high-tech stocks share interest in similar sets of equities; traders focused on the services sector also share interests, yet the two categories of traders have little overlap. One can easily visualize the corresponding regions.

Operators of e-commerce datacenters implement farms consisting of very large numbers of scalable services by cloning the service and using multicast (or publish-subscribe) to replicate updates [5]. Since a single server may be involved in replicating many objects, sets of servers end up subscribing to large numbers of heavily overlapping groups. In effect, regularity of the datacenter architecture makes it easy to identify regions that can be exploited by our protocols.

1.4 Technical Challenges

We've suggested that revisiting lightweight group multicast in an architecture based on multiple regions, rather than a single underlying multicast group, could be the key to a major advance in scalability. What makes the problem hard? Several practical challenges stand out:

- *Avoiding inbound traffic implosion.* The risk of ACK or NAK implosions associated with increased group size was recognized long ago and motivated protocols such as SRM and RMTP, but weve identified a secondary risk associated with node membership in large numbers of groups. Prior work hasn't explored this issue.

- *Maintaining efficient runtime structure.* Much of the benefit of the concept revolves around the potential to amortize communication costs and overheads over nodes that share interests and fates. Implementing protocols that exploit these shared characteristics is hard.

- *Scalable multicast.* Regions aren't necessarily small, and most existing reliable multicast protocols scale poorly. In the past we've worked on this problem, but the protocol we proposed (Bimodal Multicast) turns out to be a poor choice in systems with large numbers of groups: latency of recovery from packet loss was too high. Moreover, if a region happens to be small, gossip communication of the sort used in Bimodal Multicast makes no sense.

- *Choosing the right software architecture.* Software architectures for systems dealing with large num-

bers of concurrently active protocol stacks are surprisingly difficult to implement, debug and tune. For example, in an early stage of our work, we implemented a modestly multithreaded solution. Performance was terrible; ultimately we realized this was due to scheduling anomalies. Elimination of threading helped eliminate the issue but made our code more complex (and this is just one example among many).

- *Limiting resource usage.* With large numbers of groups and potentially large regions, there are a great many ways resource consumption can spike and cause a performance collapse. There has been little attention to the design of protocols that are predictably sparing of resources.

- *Efficient management of membership information.* Throughout our system, we need to track membership: of groups, of regions, of the underlying set of nodes. A single event (notably a failure) can trigger huge numbers of updates unless the associated event handling logic is designed for scalability.

Our contributions thus range from new protocols that can deliver a multicast reliably in a region whether it happens to be small or large to new data structures that let us efficiently update membership when a process joins or leaves thousands of groups.

The bottom line is that investment in this mixture of pragmatic and fundamental problems has a huge payoff. As will be seen below, our system achieves three times the performance of JGroups even at small scales, scales independently of the number of groups and incurs control overhead at the sender actually *decreasing* with the number of nodes, resulting in excellent scalability in this dimension. A primary reason for this is that in QSM, control overhead is not a major factor limiting scalability.

2 Protocol Overview

Our scalable multicast protocol is structured around a series of key ideas. Several are familiar from prior work, but the combination is new.

- *Leave strong reliability properties to higher-level protocols.* Our ultimate plan is to support complex reliability properties through protocol stacks in the manner of the Horus and Ensemble systems. QRM would be a low-level protocol in such a stack, and we limited ourselves to a best-effort guarantee of the sort used in SRM and RMTP. Our reasoning is basically end-to-endian: sophisticated reliability protocols often involve running non-trivial end-to-end protocols, for example when a failure occurs. Given that these are needed in any case, and that they often have reliability needs peculiar to their own reliability goals, these higher level layers might as well also overcome infrequent packet loss not addressed by QRM itself.

- *Rate based admission control.* In keeping with a philosophy that avoids burdening the sender with unnecessary incoming traffic, we use a rate-based flow control scheme that dynamically estimates the rate at which each sender can transmit. In conjunction with the mechanisms described below, this leaves the sender largely decoupled from the receivers, greatly improving scalability.

- *Group and regional membership service.* Many group communication systems employ a service to detect failures and recoveries. We go further, and employ a GMS that tracks membership in groups, regions, and the system as a whole, and handles such tasks as assigning IP multicast addresses to regions. Reporting is consistent (all nodes see the same information), and this greatly simplifies the design of protocols used in the nodes themselves.

- *Regular tiling with regions.* After struggling to develop solutions for arbitrary group overlap, we settled on an approach in which a human developer is expected to assist us by designing the system in a way that facilitates a regular tiling, with regions of relatively uniform size. If a region is very large, we partition it into smaller subregions that share a single IP multicast address but handle error recovery independent from one-another.

- *Token passing.* To avoid unpredictable bursts of ACK and NAK packets, we circulate tokens in each region to gather this and other control information. Tokens circulate rapidly, but on a predictable schedule, an innovation that works extremely well for us.

- *Local repair.* We pick some nodes in each region as loggers and are usually able to recover lost packets without involving the sender. The sender buffers packets until the loggers acknowledge them, but doesn't worry about delivery in the remainder of the region.

The subsections that follow provide additional details on these mechanisms.

2.1 Membership

As noted, we introduce a specialized 2-level group membership service (GMS). The GMS maintains not only in-

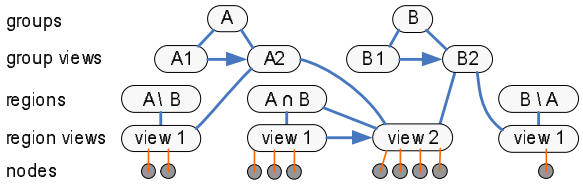

Figure 4: Group and region views in 2-level membership.

formation about groups, but also regions. Regions, like groups, experience membership change as nodes join and leave or crash, hence associated with every region is a sequence of what we call *region views*, each listing membership at some point in time. The different types of structures and their relationships are shown on Figure 4. A group has a sequence of associated group views (one of them being current), each of which maps to a set of region views, which in turn map to sets of nodes. Group and region views are immutable, while for every group and region, new group and region views will be often created and marked as current. The GMS listens to client requests as well as messages from a failure detector, updates the whole structure and sends every node affected by the change an appropriate piece of information, incremental whenever possible (containing only the updates relevant to that node). The GMS does not process every crash or client request separately; it groups requests in batches and performs changes at a predefined maximum rate. This reduces the number of view changes and the number of messages sent to nodes.

When a region has more than one recipient node, we use IP multicast to send messages, and the GMS is also responsible for assigning multicast IP addresses. If a region is very large, we break it into multiple smaller partitions that share a single IP multicast address. We adopt this approach for several reasons. First, the pool of addresses is limited. Second, we experimented with network adapters and determined that the mapping of Ethernet multicast to IP multicast is such that if IP multicast groups are assigned profligately, a node will receive interrupts and waste CPU resources even if it is not a member of any of these groups, the level of CPU consumption being proportional to the number of groups. Third, IP multicast membership changes are expensive and time consuming. Finally, since new requests are always sent to the most recent views and old messages are eventually delivered, the overhead of this approach (data delivered where it should not be, for example when a node leaves a group and time elapses before it drops out of the IP multicast address) is negligible.

Having the GMS maintain information about regions and assign version numbers (views) to their subsequent incarnations allows nodes to rely on this information as a form of common knowledge. In particular, nodes in our system can construct trees, rings and other structures spanning regions and connecting the various regions based directly from membership information, without running additional consensus protocols. Our current GMS is centralized; in the future we will replicate it to eliminate the resulting single point of failure[3].

2.2 Reliability Protocol

As noted earlier, our emphasis has been on achieving a scalable but best-effort form of reliability over which stronger models can be layered in the future.

Accordingly, before transmission every message to a group is assigned a sequence number within a particular group view. Every group view maps to some fixed set of region views, hence we now know the set of receivers. The system will try to deliver the message to this set. We do so by creating, for each request to send a message in a group (*group request*), a set of subrequests (*regional requests*), one for each region, and processing them separately (when a very large region is partitioned we still send just one IP multicast).

Recovery from packet loss associated with regional requests occurs on a region-by-region basis. In the rare case of sender failure, receivers in each region recover the message only among themselves, and it may happen that some regions deliver a message but others do not. Consistent with an end-to-end perspective, higher level protocols can and should deal with this failure case. Our mechanism thus provides a strong but not absolute form of best-effort reliability.

Regional requests are handled similarly to the group level: we assign sequence numbers within the relevant regional views, and the system keeps trying to deliver a message until every member of the relevant view either has acknowledged it or is dead (reported by the GMS as faulty).

2.3 Multicasting in Regions

The regional multicast protocol proceeds as follows. We transmit packets into the whole region using a single IP multicast group. Nodes in the region recover from packet losses from peers within the region, using the token ring protocol described in section 2.3.2 to ACK stable data and to NAK dropped packets. Receivers also provide feedback that the sender uses to adjust its multicast rate for maximum throughput. The sender participates in recovery only if an entire region drops a packet.

As a consequence of the reduced feedback, the sender will experience a greater latency in receiving acknowledgements and thus pay a greater overhead of buffering. To offload the sender, for every message we des-

ignate R nodes as *caching servers* that will buffer the received data for the purpose of peer-to-peer loss recovery, described in detail in section 2.3.3, until all nodes in the region have acknowledged it. R is a small *replication factor*, typically 5 or less in the experiments reported here. The sender considers a message to have been delivered (and stops buffering it) once it has been acknowledged by all caching replicas. This mechanism reflects the pragmatic observation that with a few nodes caching each message it is extremely unlikely that all of them will crash before the message is delivered. As long as at least a single receiver in the region has the message cached, our peer-to-peer loss recovery protocol will guarantee that it ultimately is delivered to every node that missed it in that region.

In order to provide a structure for the token passing, loss recovery and caching mechanisms mentioned above, we split every region into multiple small partitions, as described in section 2.3.1. This partitioning simplifies our design and provides additional benefits that will be discussed further. As noted before, we construct partitions and the structures built on top of them based solely on the membership information, and every node does it independently. The GMS eliminates the need for elections, consensus or other similar methods, and guarantees consistency among the views on which decisions are based.

2.3.1 Partitioning

A region of size n is divided into p partitions of size no smaller than R, $p = \lfloor n/R \rfloor$, by assigning nodes to partitions in a round robin fashion, i.e. k-th node in the region becomes a member of the $(k \bmod p)$-th partition. Each partition serves as a set of caching replicas for $1/p$ of the received packets: a packet with a sequence number i is cached by all nodes the $(i \bmod p)$-th partition.

Nodes in each partition cooperate in recovering from losses of the packets they cache and may forward (*push*) or request (*pull*) data from each other. This is achieved by our token protocol. They are also responsible for sending NAKs to the sender to request retransmission of data missed by the whole partition. Packets missed by receivers in other partitions can only be requested from nodes in the partition caching these packets. This offloads the sender by reducing the control traffic. In fact, in our experiments, with replication factor $R = 5$ the sender rarely receives NAKs and never retransmits anything after a timeout, all losses can be efficiently repaired among the receivers.

2.3.2 Token Passing

The token passing protocol on which our loss recovery scheme is based is built on a structure that resembles

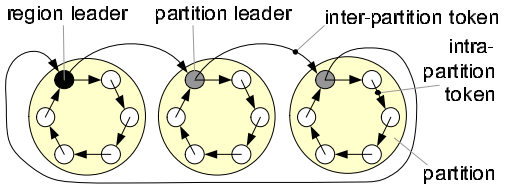

Figure 5: The torus protocol for loss recovery in regions.

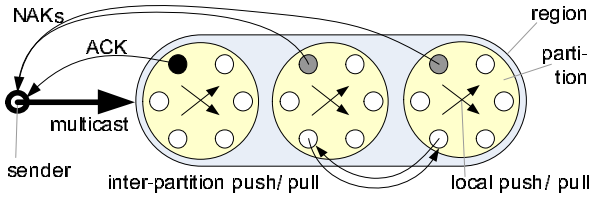

Figure 6: Non-token control traffic in loss recovery. Region leader generate ACKs, partition leaders send NAKs, nodes perform *push* and *pull* inside or across partitions.

a torus, depicted on Figure 5. Nodes in every partition form a small *intra-partition token ring*. All partitions within a region, in turn, form a single global *inter-partition token ring*. A node with the smallest address in each partition is a *partition leader*, and the leader of the first partition in the region is also a *region leader*.

At constant intervals the region leader generates a token to travel across the region. First, it circulates the token around its own partition as an *intra-partition token*. Once the token is received by the leader, it passes it to the leader of the next partition as an *inter-partition token*. The token then circulates around the second partition as an intra-partition token, then again it is passed to a yet another partition leader. This continues until the leader of the last partition passes the token back to the region leader. Tokens are passed using a simple TCP-like reliable unicast protocol. While generated at constant intervals, once released, the token progresses throughout the region at maximum possible speed (generally, a few milliseconds per partition).

Tokens serve the following purposes:

- Determine which were the latest packets received throughout the whole region, so that process can learn which packets they may have missed.
- Report losses to other nodes in the region and exchange packets. We use both the *push* model (node forwards packets without request from the other node as it learns that the other node is missing packets) and the *pull* model (a node may request forwarding of packets if it learns that other node has them).

- Find out which packets have been received by all nodes throughout the region and report them to the sender as a collective regional ACK.
- Find out which packets were missed by all nodes responsible for caching and are not recoverable and report them to the sender as a collective NAK.
- Distribute information about packets that can be purged from cache.

There is a single token per region for all the senders. Data relative to each sender actively multicasting into the region simply occupies a part of the token. This ensures scalability in the number of senders: when more senders multicast into the region, the token size grows, but the rate at which control packets circulate stays the same. Information for a given sender starts being included in the token when it multicasts data into the region and stops being included when all packets known to have been transmitted are acknowledged by all region members, up until some new data is received. For brevity, in the discussion below we refer as *token* only to the portion occupied by a specific sender.

Note that the leadership might change as the GMS notifies the receivers about changes in membership. Nodes update their status and assume leadership in a distributed manner. Since updates from GMS may arrive at different times, there might occasionally be two self-elected leaders. This does not affect the correctness of our protocol. The only information we rely on is that the GMS provides eventually consistent information about failures. A node that the GMS considers as dead in the region will be excluded by others from the protocol; if that node is actually not dead, it would need to rejoin the system.

2.3.3 Loss Recovery

In order to determine which packets have been transmitted, nodes use the token to calculate, in each round, the largest sequence number among packets received in the region, which is then distributed in the subsequent round as a *cutoff* point for loss recovery: in a given round nodes in the region consider as missing all packets with this or lower sequence numbers that they have not yet received. By introducing a full round of delay we account for the fact that recently transmitted packets might still be buffered by some nodes in their receive queues. Without this delay, our recovery protocol tends to unnecessarily generate duplicated packets.

Every time a node generates or receives a token, it creates a compressed NAK set representing the ranges of packets with numbers up to *cutoff* cached in the local partition and missed at this node and, if the token was received from another node in the same partition, the node

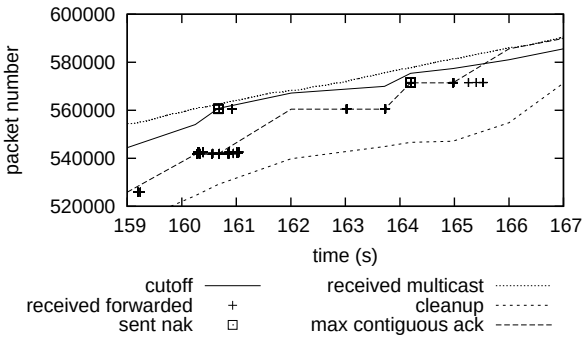

Figure 7: The loss recovery protocol at work.

compares its NAK set with a similar NAK set placed in the token by its predecessor. All NAKs reported by the predecessor, but not reported by the successor are to be forwarded (*push*) by the successor to the predecessor. All NAKs reported by the successor, but not the predecessor are to be requested (*pull*) from the predecessor (in the latter case the successor sends only a single *pull* request with a list of NAK ranges). Finally, if the token is to be passed to another node in the same partition, the node stores its NAK set in it. There is always only one such NAK set in the token, the *push* and *pull* requests mentioned above can be issued only between neighbors on the ring. As will be explained further, this decision was driven by the need to reduce the size of the token. Note that in this scheme it is possible for a node to pull a packet from a predecessor, and at the same time have it forwarded by a successor on the ring. Other schemes, including the node making its own decision as to whether it would pull or push the packet, are possible, but generally require putting more information into the token. Our experiments with such schemes show that space in the token is a scarce resource that must be used sparingly and such optimizations are generally not worthwhile.

This protocol ensures recovery of the cached packets among nodes within a single partition provided that the packet was delivered to at least one of the caching replicas. To recover from cached packets missed by the whole partition, nodes in the partition use the token to calculate the intersection of their NAK sets, which is then included in the collective NAK forwarded to the sender by the partition leader. Every partition reports its own losses independently (see Figure 6).

While processing the token, a node also creates NAKs for packets cached in other partitions and sends *pull* requests to the leaders of other partitions (one request per partition containing a compressed list). These requests are satisfied as soon as data is available. We contact only partition leaders, as the leaders usually contain the most

up to date packets (note that while *push* involves simply transferring data between neighbors, *pull* requires that it be requested first, hence forwarding towards the partition leader is generally faster). Other schemes, including choosing a random node in every partition, are possible.

Finally, the token is used to calculate the maximum value such that all packets with numbers up to this value are stable within the region. This single number is sent by the region leader as the only form of ACK to the sender (as well as passed around in the subsequent round to purge messages from cache). We experimented with adding more feedback for the sender, such as isolated ACKs, but found that it generally increased the amount of details about the different partitions that need to be passed in the token all around the region, which noticeably increases the token size without a clear benefit. Even with this minimal feedback, in our experiments messages are typically reported to the sender as acknowledged by the region within a few token rounds, the retransmission mechanism on the sender essentially never kicks in and NAKs sent to recover from partition-wide losses are rare. We favor simplicity and consider this a strength of our protocol. Offloading the sender in any way possible turned out to be good for performance.

Our description omitted some details for brevity. Among these, the more important issues were:

- We found it essential to bound resource utilization throughout the system, hence nodes are only allowed to report a limited number of NAK ranges, which requires additional handling when calculating NAKs, *pull* and *push* requests.

- We settled on a solution where requests for a given packet by nodes outside its caching partition is delayed until it is stable on all of its caching replicas. This reduced the amount of state maintained, ensured that *pull* requests can be satisfied immediately and simplified handling of the case where the same packet is requested twice in different token rounds. We found delaying of *pull* requests to be good for throughput, especially in case of massive losses.

In our experiments we generate a token once per second. This proved sufficient across a variety of configurations.

All control traffic is handled by a reliable TCP-like unicast protocol and rate-controlled. The rate is set to a fixed small value to minimize the impact it might have on multicast traffic. Multicast is controlled by a separate, adaptive rate control scheme described in section 2.4.

2.4 Rate Control

We managed to achieve effective rate control with minimal feedback, where the sender adjusts its multicast rate μ_{snd} based solely on a single value μ_{rec} received from nodes in the region and representing the lower bound on the rate at which all nodes in the region can receive the data multicast by this sender.

The rate μ_{rec} is calculated as follows. Each node i in the region maintains a smoothed estimate $\mu_{rec}^{(i)}$ representing the rate at which it is currently receiving new, previously unseen packets (we use a k-sample moving average of the node's rate history calculated in fixed intervals Δ, typically $k = 10$ and $\Delta = 1\text{s}$). The region leader, via the token protocol, periodically calculates a minimum of these rates and divides it by m, the number of senders currently multicasting into the region, according to the following formula:

$$\mu_{rec} = \frac{\min_i \mu_{rec}^i}{m} \tag{1}$$

This value, piggybacked on the ACKs sent to all the m senders, represents a lower bound on a fair share of the currently available bandwidth. Each sender then sets its multicast rate according to the following formula:

$$\mu_{snd} = \max(\mu_{min}, (1+\lambda)\mu_{rec}), \tag{2}$$

where λ is a *growth coefficient* that provides a tradeoff between the latency at which the controller tunes up to the maximum available rate and losses resulting from tuning it too high, and μ_{min} is some minimum rate, necessary to kick the system out of stagnation after long period of nonactivity or a series of massive losses.

This formula makes the senders multicast at a rate just slightly higher than that declared by the slowest receiver, which makes it possible for the rate to grow over time. At the same time, by keeping λ low we avoid overloading the system. Note that since different nodes process at different speeds in different intervals of time, the average rate among receivers will generally be considerably higher than the minimum in the region. This is precisely the subtle property that ensures the stability of our system: the actual capacity of the region is higher than the $m\mu_{rec}$ communicated to the senders and therefore we will not experience loss rates comparable to λ, as one might initially think.

In order to tune the sender to precisely the desired rate μ_{snd}, we use a simple rate control scheme enhanced with an adaptive adjustment mechanism. Specifically, we use a component that, given a parameter α, tunes the system to a sending rate of $f(\alpha)$ where f is monotonically increasing, and adjust parameter α in fixed intervals (typically at most once per second) according to the following formula, loosely inspired by the Newton's method:

$$\alpha \leftarrow \alpha + \rho(\frac{\mu_{snd}}{\mu'_{snd}} - 1), \tag{3}$$

where μ'_{snd} is the measured *actual* sending rate, calculated by the sender in a way similar to the manner whereby receivers calculate μ^i_{rec}. The parameter ρ controls the inertia of our mechanism (typically 0.5).

As the controlled component, we enhance sender with a simple credit system. Sending a message consumes a credit out of a fixed pool of size C, sender can send only if credit level is positive. Credits are recreated over time. Specifically, when credit goes below threshold C_{low}, we setup a timer for interval τ to recover at a level C_{high} using the following formula:

$$\tau = \frac{\min(C_{high} - c, \Delta_{max})}{\mu_{snd}}, \qquad (4)$$

where c is the current credit level and Δ_{max} is the maximum number of credits to recover, added to reduce burstiness of the system. When the timer expires, we increase the number of credits by $\Delta T \mu_{snd}$, where ΔT is the actual amount of time that has elapsed. If the credit is still below C_{high}, we keep setting a timer using the above formula until we reach that level. We found this scheme to work well in practice, in particular on one 3.8GHz machine with a gigabit connection this scheme was able to achieve the rates as high as 20,000 packets/s exactly as requested without the adaptive adjustment component. On the other hand, on slower 1GHz 100Mbit platforms, neither this nor any of several other sender rate control schemes we tried was able to match the desired rates without adaptive adjustment and without leading to high burstiness, the reason being that all the mechanisms we tested proved easily disturbed by scheduling.

Distributed rate limitation represents a significant research issue for us, and this preliminary solution is only a first step. In the future, we hope to explore other options, such as schemes allowing senders to lease bandwidth from the region. We believe the token passing scheme is flexible enough to support a wide range of algorithms.

2.5 Architecture

Our system is implemented in 99% in C#[2] for the managed .NET framework, currently only for the Win32 platform, as a library with a single-threaded core, optimized for fast event processing and implementing its own simple event scheduling. The relationship between the core and the rest of the system is shown on Figure 8. A single I/O completion port (*I/O queue*) is created to handle all I/O events, including confirmation of packets received, completed transmissions and errors, for unicast and multicast traffic, for all sockets created by our system. The

[2]We only use a small portion of C++ code to access I/O completion port APIs unavailable through the .NET interfaces or P/Invoke in C#. Because neither I/O completion ports nor a similar efficient mechanism are currently available on Linux, we only support the Win32 platform.

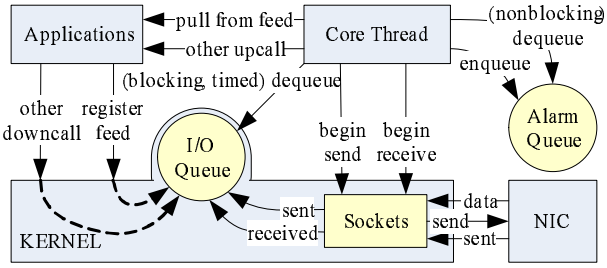

Figure 8: The overall architecture of our system: a single thread in the core managing its own scheduling policy.

core thread is continuously processing events from the I/O queue as well as from its own *alarm queue* (a priority queue implemented as a splay tree) where it stores timer-based events. The two types of events are processed in a round-robin fashion, in batches to minimize overhead (typically with a quantum of 100ms for any batch of I/O events and 50ms for batches of timer events). Additionally, in order to minimize losses, when a data is found on a socket, that socket is drained of all the received packets queued on it before any other processing takes place. In the rare case where there are no events to process, core is waiting for I/O in a blocking system call. All our protocols are based on UDP. We use a single socket for all incoming unicast traffic.

In order to reduce buffering inside our system, streamline flow control between the various components and the OS, and optimize management of resources such as buffer space, number of sockets etc., we employ a *pull* scheme. A component that intends to send data creates an *output channel* by registering a *data feed*, essentially a callback that returns objects to be transmitted. Data is pulled from the feed asynchronously, as rate control and resource limitations permit. When no more data is available, the feed is internally deactivated to eliminate useless polling. The component must then signal the feed when new data is available in order to reactivate it.

This *pull* model is used throughout our protocol stack and exposed to the applications using our library. We also provide wrappers that allow the application to use a more standard *push* interface, i.e. the familiar *SendMessage* call. Similarly, our system includes convenience features such as buffering components, message batching components etc. However, performance with the *pull* interface is superior, and the experiments reported here used it.

We support multithreaded applications. Calls made by an application to the core (*downcalls*), e.g. to signal a channel, are implemented via posting special requests to the I/O queue and/or inserting them into non-

blocking queues[3], implemented with *compare-and-swap* (CAS) operations (available on most of today's architectures). Calls made by the core to applications (*upcalls*), e.g. to pull data from a feed, are made directly, it is up to the user to protect data structures, possibly in a non-blocking manner (nonblocking queues are one option).

We also provide a serialization mechanism, through which the user can send arbitrary types of objects that implement our simple serialization interface. The interface requires a class to have a uniquely assigned number, and objects to be able to store their contents by appending data to a header or buffers to a scatter-gather pool, and to load its contents from a given header and buffer. This is necessary to avoid the enormous overheads of serialization in the managed framework, where names, signatures and other bulky information on the order of 150 bytes or more are typically appended even to the tiniest amounts of data sent in binary mode, and to enable scatter-gather mode of operation without an extra copies. A wrapper capable of transmitting arbitrary objects is available for developers who favor simplicity over efficiency.

The structure of our protocol stack reflects the structure of our protocols. Accordingly, we have components responsible for maintaining local view of membership, *message sinks* sending in groups, region views or to individual nodes as well as modules that manage resources or control concurrency. We omit details for brevity.

3 Evaluation

3.1 Setting

We evaluate performance on two sets of nodes [4], first a 71-node set of Pentium III 1.3GHz 512MB machines, and a the second a 25-node set of Pentium III 1GHz 2GB machines, both on a switched 100 Mbps network, running Microsoft Windows Server 2003 Enterprise Edition.

For JGroups evaluation we use JGroups version 2.2.8 running on Sun JDK 1.5.0.03 runtime configured with the same JVM flags as used by the author of this package. We used a *fc-fast-minimalthreads* protocol stack as recommended by JGroups developer, with ACK/NAK-based loss recovery and flow control on UDP, but with batching and higher-level protocols, including virtual synchrony, disabled[5]. Results of our own performance tests at small scales were similar to results obtained from a test written by the JGroups developer and run on our hardware. QSM was evaluated on the Microsoft Visual Studio 2005 Release Candidate.

[3] A restricted version that allows applications to enqueue one element, but to dequeue only all elements at once is fast, it requires only two CAS operations.

[4] Power and cooling problems prevented us from being able to run all tests on the same hardware before the submission deadline. For the same reason, some data points are missing. We are in the process of deploying our code on a 250-node cluster that will enable us to run experiments at a large scale long before the camera ready copy is due.

[5] However, we left message batching enabled for all control traffic.

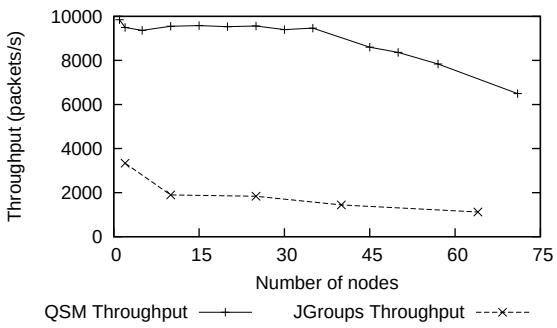

Figure 9: Multicast rate as a function of the number of nodes in a single group spanning the whole system.

In all our tests senders (usually one sender, with the exception of section 3.7) multicast small packets consisting of a few integers. Including TCP/IP headers as well as any headers specific to JGroups or QSM, JGroups packets are 124-bytes and QSM packets 90-bytes in length. With one exception, both the senders and all the receivers subscribe to the same set of groups, spanning the whole system. Each sender multicasts in all groups, round robin. The reported throughput rates refer to the combined rate for all groups, ignoring warmup and cooldown periods.

JGroups throughput is stable, we use runs typically a few minutes long. QSM throghput varies over time, the runs used for the results reported here last 10-40 minutes.

3.2 Throughput

As shown in Figure 1, performance in our system does not directly dependend on the number of groups, the only way the presence of multiple groups directly manifests is at the sender, where the amount of state involved grows linearly and consumes memory (the growth is slow and not noticeable in our experiments). Indirectly, as discussed in section 3.5, performance drops as the regions that the groups consists of become smaller, which is indeed more likely with a large number of irregularly overlapping groups unless care is taken to prevent that. In comparison, while scaling from 1 to 256 groups, JGroups performance degrades by almost 60%.

Our system also scales very well with the number of nodes. As shown in Figure 9, scaling up from 1 to 71 nodes results in a drop of performance by 1/3, from 9850 to 6500 packets/s. As explained in section 3.8, we believe that this drop in throughput is in large part due to the difficulty in estimating the rate at which the group

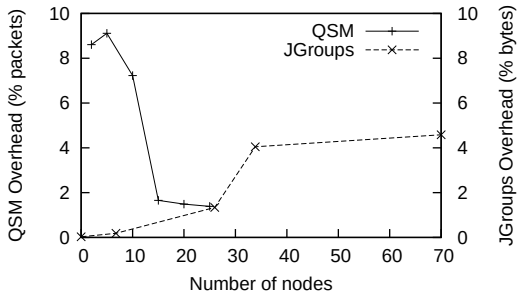

Figure 10: Overhead as a function of the number of nodes (1 group, 1 sender). QSM overhead shown as a fraction of control packets. JGroups overhead, due to message batching, shown as a fraction of bytes in those packets.

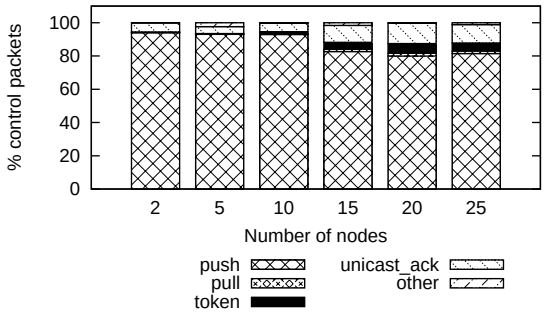

Figure 11: Overhead in QSM as a function of the number of nodes, for a single group and 1 sender.

can receive packets rather than just the capacity of the network and limitations of our loss recovery mechanism.

At the same time, even with just 2 nodes in a single group JGroups achieves a multicast rate of 3333 packets/s, half that of QSM with 71 nodes; with 40 members JGroups degrades to 1441 packets/s, more than 6 times lower than that of QSM with the same group size.

3.3 Overhead

We measured overhead at the sender in both JGroups and QSM by capturing packets in Ethereal and calculating the percentage of those carrying control messages, and then further decomposing these into subclasses based on message type. Because JGroups batches its control messages, we also looked at the sizes of its messages. For brevity, we do not discuss overhead at the receivers.

The network overhead in QSM is independent on the number of groups. Also, as shown on Figure 10, it decreases as more nodes are added to the system because the burden of packet forwarding is being distributed

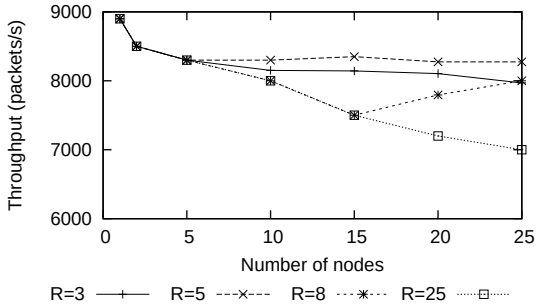

Figure 12: Throughput in QSM as a function of the number of nodes for different partition sizes.

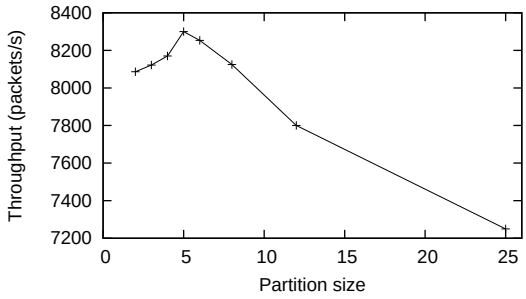

Figure 13: Throughput in QSM in a single 25-node group as a function of partition size.

across a larger set. Indeed, as shown on Figure 11, at smaller scales forwarded packets (*push*) account for over 90% of all QSM traffic and the ratio decreases with scale, giving place to tokens and ordinary unicast ACKs (*ack*).

In JGroups, as Figure 10 suggests[6], the fraction of data transmitted in control packets grows at least linearly with the number of nodes. Thanks to message batching, at smaller scales the resulting overhead is actually quite low.

Our experiments suggest that JGroups control overhead grows relatively slowly as a function of the numebr of groups. When scaling from 1 to 128 nodes, the overhead grows by the factor of 2.

Note that in both scalability scenarios, the increase in overhead in JGroups is present despite the radical drop in throughput. This is not the case for QSM.

3.4 Partitioning

The choice of partition size has a noticeable impact on performance. Because information about losses in

[6]We did not analyze JGroups source code and therefore JGroups overhead results discussed here, based entirely on an analysis of captured packets, may not be 100% accurate.

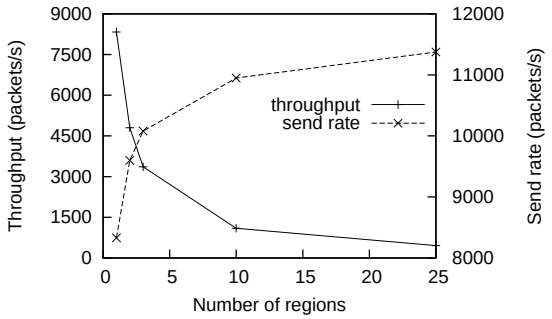

Figure 14: Throughput and the rate at which packets are physically transmitted as a function of the number of regions in a group. Multicasting in a single group, with artificial groups created to break it into smaller regions.

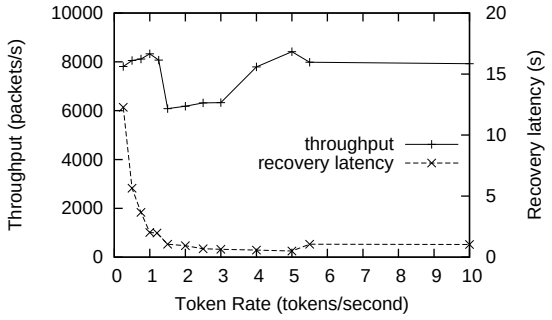

Figure 15: Throughput and average time to recover lost packets (see section 3.9) as a function of the rate at which tokens are generated by the region leader.

caching partitions is exchanged only between neighbors, loss recovery occurs on a hop by hop basis, i.e. a packet missed by a chain of nodes on the ring will be recovered in time linear to the distance between these nodes. This effect gains in significance when partitions become more than a few nodes in size. On the other hand, having too many small partitions increases the number of potential NAK sources, and the smaller number of caching replicas increases the likelihood that packets cannot be recovered just among the receivers.

As shown in Figure 12, while with partition size 5 throughput degrades very slowly, much larger partition sizes lead to a fast decrease in performance. Throughput in a 25-node region sliced into 5 partitions is almost 20% higher than when the whole region forms a single 25-node partition.

The inherent tradeoff between the number of caching replicas and the size of token rings for a 25-node region is illustrated on Figure 13. Increasing the number of replicas beyond 5 is not worthwhile.

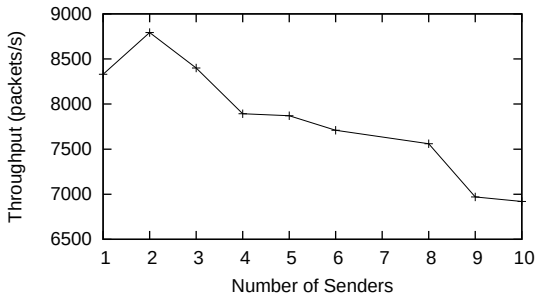

Figure 16: Scaling with the number of senders in our fair sharing scheme in a single 25-node group.

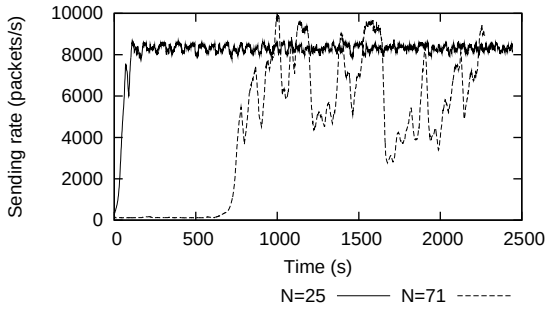

Figure 17: Sending rate as a function of time in 25-node and 71-node groups. Rate calculated in 1s intervals, smoothed by a 20-sample moving average for clarity.

3.5 Region Size

As mentioned earlier, performance of our scheme decreases with the number of regions: since every region forms a separate multicast group, multicasting at a rate μ in a group consisting of k regions requires the sender to generate $k\mu$ packets per second. This quickly becomes a bottleneck. As Figure 14 shows, throughput degrades slightly less than inversely proportionally to the number of regions. With more packets the rate at which they are sent is actually increasing, asymptotically approaching the maximum rate at which the system is able to transmit.

Since it is only the number and sizes of regions that bear any significance, we omit discussion of performance with different overlap patterns.

3.6 Token Rates

The token generation rate is a crucial internal parameter with broad impact on the performance of QSM. Other important parameters include rate controller settings, retransmission timeouts, scheduling quanta, and the vari-

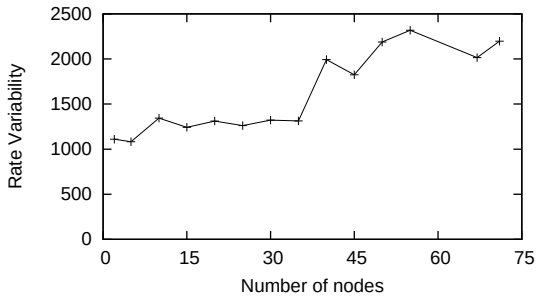

Figure 18: Sending rate variability expressed as a standard deviation of rate samples taken in 1s intervals.

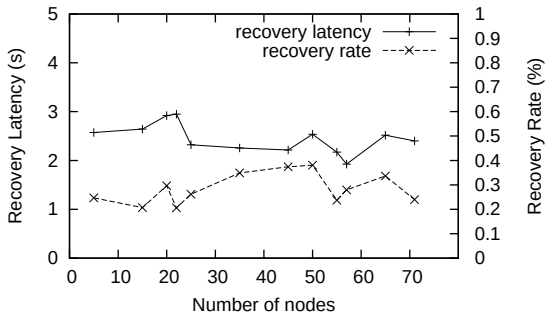

Figure 19: The average time to repair packets lost during IP multicasting (*recovery latency*) and the percentage of packets that needed to be repaired (*recovery rate*).

ous bounds on resource utilization, but token rate dominates.

Token rate should be high enough to ensure timely repair: the QSM token is the only means of communicating losses among partition members and the amount of information that we allow in the token is limited, hence limiting the token rate may cause a bottleneck. Limiting token rates also increases latency, buffering overheads and reduces feedback for the rate control purposes. On the other hand, if the token rate is set too high, delivery latency drops but we incur excessive control overhead. This dependency is shown[7] on Figure 15.

3.7 Multiple Senders

On Figure 16 we present throughput for a test with multiple senders using fair sharing; all senders belong to the same group and receive each-other's messages. The reported throughput represents the total across all senders. As we can see, two senders can actually multicast faster,

[7]We haven't had time to investigate the drop in performance for rates between 1 and 5.

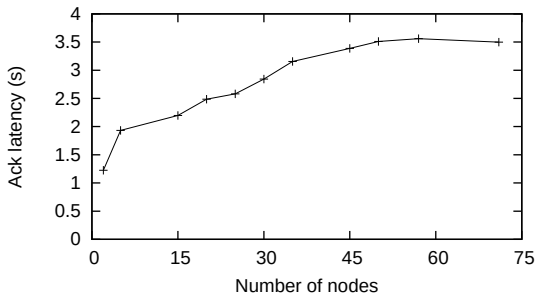

Figure 20: Acknowledgement latency as a function of the number of nodes.

because the fluctuations in sending rate partially cancel out, leading to smoother throughput, and the presence of two uncorrelated sources reduces burstiness of the traffic. As the number of senders increases, throughput drops, but with 10 senders it is still at 80% of its original value.

3.8 Rate control

Our rate control scheme generally requires between tens of seconds and several minutes to achieve maximum throughput, depending on the initial and minimum rates it is configured with. In our experiments we used the lowest settings to maximally stress the system by allowing it to almost stop. The rate generally grows slower in larger systems, where minimum rate among receivers will generally be considerably lower than average, and takes more time to recover, leading to more variability and lower throughput. We believe this to be the major reason for decrease in performance with larger systems mentioned earlier.

Figure 17 shows how sending rate in one larger and one smaller system changes over time. In the smaller system, the rate changes smoothly, slightly increasing and falling by a few percent in cycles spanning tens of seconds. In the larger system, the rate fluctuates between 3000 and 9000 packets/s in cycles that span several minutes. This effect has been quantified on Figure 18. This observation makes it clear that more work on rate control in large configurations will be needed.

With a more sophisticated heuristic and more aggressive rate control, or by manually bounding the rate to keep it from dropping too much, it should be possible to sustain higher throughputs. However, our results already show that even with a minimal feedback once per second, a 70-node system can achieve high performance.

3.9 Recovery Latency

As mentioned earlier, our loss recovery scheme repairs most lost packets within a few rounds. Figure 19 shows the length of time measured at one receiver between the moment when a delayed packet should have arrived (as calculated by looking at neighboring packets) and the time when it actually is received. On average, recovery takes 2.5 token rounds, independent of system size. Recovery latency multiplied by the percentage of packets delayed (also shown on Figure 19) represents the contribution of packet loss to end-to-end latency. In our experiments, this typically averaged a few milliseconds. It may be reduced by increasing token rate (see Figure 15).

3.10 Acknowledgement Latency

Figure 20 shows the average time between sending a message and full-group acknowledgement. In larger systems, this takes up to 3.5 token rounds, while in smaller systems it takes slightly over 1 round, much less than the time to recover packets. The latter is possible because packets are considered by the sender delivered as soon as they are stable at all the caching replicas.

4 Related Work

Our ideas build upon a rich body of work in best-effort scalable reliable multicasting based on the exchange of ACKs and NAKs, including RMTP [10], SRM [6] and LBRM [7]. A survey of these and other techniques can be found in [8], [12] and [13]. Our work differs from those systems in the explicit support and optimizations targeted at multiple groups, hence we confront our design with that of JGroups [2] and Spread [1], which offer similar support. We also differ from other best-effort approaches in the novel way in which we perform flow control and loss recovery.

Our work was inspired in part by the idea of sharing ACK trees among multiple senders, proposed in [9]. We extend this idea by merging protocols not just among senders, but also among groups, a technique that required developing new ways to view at group membership and novel loss recovery and flow control techniques that supported this framework.

The idea of using token rings in the context of multicast communication has been proposed e.g. in [11]. While much earlier work used token rings for total ordering, we use them simply as the lowest-overhead mechanism for aggregating state among multiple nodes. Our sets of caching replicas resemble other log-based approaches [7], but our partitioned torus-like structure for loss recovery differs significantly from prior work.

Rate admission approach to multicast flow control has been explored e.g. in [14]. Our work differs from this and similar techniques, mostly using some form of loss-based feedback, in that we adjust rates exclusively based on perceived rates and estimates of system capacity, and with minimal feedback.

5 Acknowledgments

This work was supported by DARPA/IPTO under the SRS program and by the Rome Air Force Research Laboratory, AFRL/IF. Additional support was provided by the NSF, AFOSR, and by Intel.

We want to thank our colleagues Robbert van Renesse, Mahesh Balakrishnan, Tudor Marian and Maya Haridasan for the invaluable feedback they provided, and to Bela Ban for helpful hints during our JGroups evaluation.

References

[1] AMIR, Y., DANILOV, C., MISKIN-AMIR, M., SCHULTZ, J., AND STANTON, J. The spread toolkit: Architecture and performance. Tech. Rep. CNDS-2004-1, Johns Hopkins University, 2004.

[2] BAN, B. Design and implementation of a reliable group communication toolkit for java, 1998.

[3] BIRMAN, K. P. *Reliable Distributed Systems*. Springer Verlag, 2005.

[4] BRIMAN, K. P. A review of experiences with reliable multicast. *Software Practice and Experience 29*, 9 (1999), 741–774.

[5] DEVLIN, B., GRAY, J., LAING, B., AND SPIX, G. Scalability terminology: Farms, clones, partitions, and packs: Racs and raps. Tech. Rep. MS-TR-99-85, Microsoft Research, 1999.

[6] FLOYD, S., JACOBSON, V., LIU, C.-G., MCCANNE, S., AND ZHANG, L. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking 5*, 6 (1997), 784–803.

[7] HOLBROOK, H. W., SINGHAL, S. K., AND CHERITON, D. R. Log-based receiver-reliable multicast for distributed interactive simulation. In *SIGCOMM* (Cambridge, MA, Aug. 1995), pp. 328–341.

[8] LEVINE, B. N., AND GARCIA-LUNA-ACEVES, J. J. A comparison of reliable multicast protocols. *Multimedia Systems 6*, 5 (1998), 334–348.

[9] LEVINE, B. N., LAVO, D. B., AND GARCIA-LUNA-ACEVES, J. J. The case for reliable concurrent multicasting using shared ack trees. In *ACM Multimedia* (1996), pp. 365–376.

[10] LIN, J. C., AND PAUL, S. RMTP: A reliable multicast transport protocol. In *INFOCOM* (San Francisco, CA, Mar. 1996), pp. 1414–1424.

[11] MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM 39*, 4 (1996), 54–63.

[12] OBRACZKA, K. Multicast transport protocols: a survey and taxonomy, 1998.

[13] PINGALI, S., TOWSLEY, D., AND KUROSE, J. F. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *SIGMETRICS* (1994), pp. 221–230.

[14] YAVATKAR, R., GRIFFOEN, J., AND SUDAN, M. A reliable dissemination protocol for interactive collaborative applications. In *MULTIMEDIA '95: Proceedings of the third ACM international conference on Multimedia* (New York, NY, USA, 1995), ACM Press, pp. 333–344.