QuickSilver Scalable Multicast[1]

Krzysztof Ostrowski
Cornell University

Ken Birman
Cornell University

Amar Phanishayee
Cornell University

Abstract

Reliable multicast is useful for replication and in support of publish-subscribe notification. However, many of the most interesting applications give rise to huge numbers of multicast groups with heavily overlapping sets of receivers, large groups, or high rates of dynamism. Existing multicast systems scale poorly in one or more of these respects. This paper describes QuickSilver Scalable Multicast (QSM), a platform exhibiting significantly improved scalability. Key advances involve new ways of handling time and scheduling, adaptive response to observed traffic patterns, and better handling of disturbances.

1. Introduction

In this paper we report on QSM, a new multicast substrate targeting large deployments of computers that communicate using publish-subscribe or event notification architectures. The work described here focuses on configurations that might include thousands of nodes, each belonging to many multicast groups; in aggregate, there might be tens or hundreds of thousands of groups. Groups can be large, although our protocols work best if large groups either have a small set of senders or low data rates. A single node may be a sender and/or a receiver in multiple groups. Groups thus exhibit extensive overlap. Figure 1 illustrates this, although overlap will rarely be so regular.

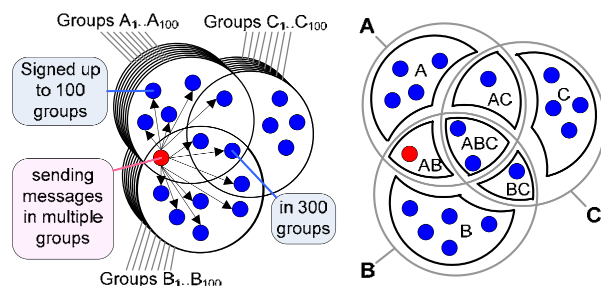

Figure 1. Left: A large number of heavily overlapping groups. Right: their "regions of overlap".

Our goal is to offer the highest possible throughput while keeping latency reasonably low. QSM's reliability property is similar to that of SRM [3] or RMTP [4]: the system should deliver each pending message to all live members of the target group, despite disruptions such as packet loss, load surges, or node failures.

QSM responds to the 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 [2]. These are older systems, but we are also in dialog with developers of very large data centers, such as Amazon, Google, Yahoo!, Lockheed Martin and Microsoft. Developers uniformly express frustration with the lack of a communications substrate having the mixture of properties targeted by our effort.

A robust and scalable mechanism for efficient, reliable, group multicast opens powerful new design options for developers. If groups can be used casually, separate groups could be created for individual events, or individual data items being tracked by the system. In a stock exchange, a separate group could exist for each stock being traded. In a data center, a separate group could exist per a category of products or for each individual service etc. We believe that the ability to declare and use multicast groups as cheaply as one declares and uses files or other objects will lead to entirely new ways of building distributed systems. Contributions include a new region-based multicast protocol, its evaluation, and a number of insights gained in "evolving" the system.

Although we have been working on it for almost two years, QSM is still a work in progress. The initial release runs on Windows .NET and comprises some 75,000 lines of code (in C#), plus debugging and instrumentation support. QSM requires IP multicast, although [1] discusses an extension to WAN settings supporting other dissemination frameworks. In future work, we hope to port QSM to other operating systems, strengthen fault-tolerance and security and to integrate QSM with strongly-typed interface features of popular service oriented architectures. This paper limits itself to features of the system that are complete and have been fully evaluated.

[1] Our work was supported by grants from AFRL, AFOSR, DARPA, Intel and NSF. Contacts: {krzys,ken,amar}@cs.cornell.edu

How do existing systems perform? To set context for our work, we evaluate JGroups [8], a popular group communication package available as part of the JBoss platform. JGroups is generally considered to be a solid platform and a good performer within the space. This said, our evaluation is not intended as a comprehensive side-by-side comparison with QSM, but rather as an illustration of challenges we faced. Much research has been done on scalable reliable multicast (e.g. [3], [4]), and JGroups may not be the most scalable of protocols. Even so, we believe that the observations and conclusions that follow are common to a wide range of contemporary systems.

In all experiments reported here we use a cluster of 110 nodes, with Pentium III 1.3 GHz CPUs, 512 MB memory, on 100 Mbps LAN, running Windows 2003 Server Enterprise Edition. Our experiments on JGroups use a performance test and JVM settings recommended by the author, version 2.2.8 on Sun JDK 1.5.0.03. JGroups lacks a good rate control mechanism, hence we experimented with several sending patterns (in burst, batches etc.), and report the best-performing scenarios.

Our graphs show averages from hundreds of samples taken in a steady state lasting 5-20 minutes. Error bars are included, but very small, and therefore might not be visible. In many cases, short term rates or individual latencies exhibit high degrees of variance; indeed, this is a topic discussed in Sections 2 and 3 of the paper.

Our experiments on QSM run on .NET Framework 2.0. For clarity, we limit ourselves to 1000-byte messages and disable "batching": small packets are not bandwidth efficient, but batching would obscure the core communications behavior. Nonetheless, we allow JGroups to batch control messages. We do not pre-allocate objects, and left garbage collection running. Consequently, both JGroups and QSM spend much of their time in JVM or CLR, respectively, mostly on memory allocation and garbage collection. In light of the contemporary preference for managed runtime environments, we believe these results to be realistic and that the comparison is a fair, apples-to-apples story.

In Figure 2 a single sender multicasts in a single group of varying size at the maximum sustainable rate. JGroups does not scale well with group size. Throughput with 110 nodes is 1/7th of throughput with 2, and less than 3.5% of the nominal 100Mbps fabric speed.

In Figure 3 a single sender multicasts in several groups concurrently in a round-robin fashion, i.e. message **k** is published in group **k** mod **g**, where **g** is the total number of groups. In this scenario, we want to measure the impact that the number of groups alone has on system performance; consequently, we make all groups completely overlap on the same set of members (we explore other patterns of overlap in section 2). JGroups scales poorly in the number of groups. Even with 2 nodes, performance is roughly halved as we shift from 2 to 256 groups. Although performance degrades more slowly with larger groups, the effect does not appear to be significant.

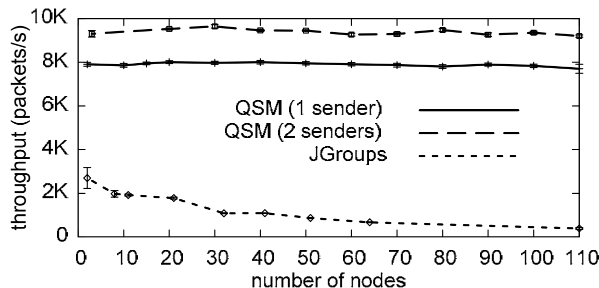

Figure 2. Scaling in group size (with a single group).

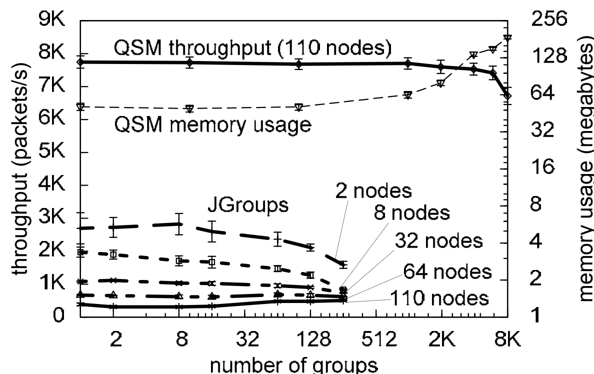

Figure 3. Scaling in the number of groups. For this experiments, each group has identical membership.

Can we do better? What factors limit performance? These questions inspired our work on QSM. As shown on Figure 2 and Figure 3, the answer to the first question will turn out to be positive. Within the range of group sizes tested, QSM scales almost perfectly with the number of nodes, achieving throughput of 8000 packets/s with one sender or 9500 packets/s with two senders, supports thousands of groups with minimal performance penalty (only 4% with 6000 groups), mostly resulting from the increased memory consumption[2]. We believe that with enough nodes, QSM performance would eventually deteriorate, but our cluster simply isn't large enough to expose its limits.

[2] In the same test on a cluster of nodes with 2GB memory, throughput does not degrade with 8192 groups.

QSM is a complex system, and brevity prevents us from describing it in detail here. Like JGroups, QSM is coded in a managed language (Java in their case, C# in ours). The first versions of QSM performed very much like JGroups, too. Only by simultaneously addressing a number of seemingly independent issues was it possible to achieve dramatically better scalability and performance. In the remainder of this paper, we describe our story by focusing on three major aspects: how to design a protocol that scales in multiple dimensions, how to get the system to work at the highest speeds and how to maintain good performance despite disturbances such as crashes, loss, garbage collection, or scheduling.

2. Scalability

2.1. Dissemination

Scalability has long been an issue for multicast protocols, but typically in just a single dimension at a time. For example, one can point to work on multicasting to "lightweight" groups by multicasting in some kind of a covering group and then filtering to discard undesired messages [9]. There has been work on protocols that scale with the number of nodes, avoiding ACK/NAK implosion through hierarchy (as in RMTP), support shared recovery mechanisms (as in SRM), share workload to improve scalability in the number of senders [6], or even weaken the reliability property (as in our own Bimodal Multicast). But no existing system scales well in several dimensions at once.

Let's begin by considering systems that support large numbers of multicast groups. Solutions to this problem fall roughly into two categories: (1) lightweight groups, and (2) solutions that simply run separate protocols for each group independently.

Lightweight groups were introduced in the Isis Toolkit, but Spread [5] was the first platform optimized for this model. Spread clients relay multicasts to a small group of agents; these broadcast each message, filter them on reception, and then relay matching messages to the clients. The approach introduces two extra message hops, to and from the agents. Agents experience load at least linear in the system size and multicast rate, and can also suffer from contention if each must support a really huge number of clients. Thus while Spread can support huge numbers of *groups*, it works well only if they overlap cleanly and if the number of *processes* using the system is reasonably small. Many commercial publish-subscribe systems also fall into this category ("topics" play the role of "groups"). Typically, they work much like Spread. We believe that the scalability limits just summarized are fundamental.

If we rule out lightweight group approaches, the scenario explored in Figure 1 poses real problems for the other existing technologies. If each group is operated independently, the sending node will multicast data separately in a large number of groups, often using a separate IP multicast group in each. Each group implements its own flow control and loss recovery protocol; hence a node that participates in many groups independently exchanges large numbers of ACKs, NAKs, and other control messages with its neighbors. Not only does traffic rise linearly in the number of groups, but the communication pattern sabotages efforts to prevent ACK/NAK implosion. For example, in Figure 4 (left) we have drawn multiple superimposed acknowledgement trees of the sort used in RMTP. RMTP uses hierarchy to avoid such problems, but unless all groups share a single tree, a node will have neighbors in multiple trees, and the benefit might be lost. Such effects can be reduced by clever engineering (for example, JGroups, batches control messages to amortize overhead), but the problem is basic and suggests that cross-group optimization and coordination is needed.

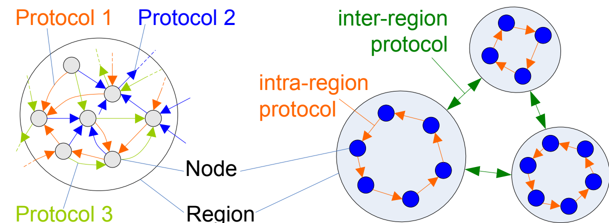

Figure 4. Left: Separate protocols running in different groups overlapping on a set of subscribers. Right: two-level protocol architecture used in QSM.

Systems that run multiple side-by-side copies of a protocol stack also miss opportunities for batching messages sent to groups that overlap on a set of receivers. In systems where each group is unaware of the others, separate sockets, IP multicast addresses, and receive buffers will be used for each group, hence memory consumption grows linearly, and less buffering space is available per socket, resulting in more dropped packets. We've performed experiments that revealed high overheads when a node subscribes to a very large number of IP multicast groups. Such a node will be disturbed even by traffic in IP multicast groups to which it isn't subscribed because filtering in network adapters is approximate. If an interface is asked to join too many

groups, most incoming packets must be received, then filtered in the device driver.

In developing QSM, we started with an awareness of these kinds of pitfalls and tried to develop an architecture that can scale in multiple dimensions simultaneously, without the kinds of limitations just discussed.

A first decision was to exploit IP multicast channels to promote buffering, reduce resource usage and the need for the OS to filter undesired packets, and allow sockets to be better provisioned. But this raises an obvious question: **Is there a way for groups to define and share IP multicast channels that completely avoids filtering?**

In QSM, we partition the system into *regions*. Consider *G(n)*, the set of groups to which node *n* belongs. We will say that nodes *n* and *m* belong to the same *region* if *G(n)=G(m)*. Each region is assigned a separate IP multicast address. Every group maps precisely to a set of regions[3] (see Figure 1, right), hence no filtering is necessary: at the sender, packets for a given group are transmitted via separate IP multicasts to each of the regions spanned by the group.

This regional mapping could give rise to a situation where each node belongs to a unique set of groups, and ends up isolated in a region to which no other node belongs. In practice, degenerate regions are rare (the intuition is that cloned applications exhibit high degrees of regularity). In settings where this issue is a concern, one work-around would be to redefine the notion of region by assigning nodes with "similar" group memberships to the same region, and then having each node check for and discard unwanted messages. This reverts towards lightweight groups. Instead, QSM employs a "hybrid" mechanism, described later.

A regional mapping has system-wide implications. Batching, serialization, and rate control must done per-region, rather than on a per-group basis. This introduces complexity, but also opportunities for sharing workload, buffering and rate control. The actual work of determining the region boundaries is handled by the Global Membership Service (GMS).

Shifting attention to the receiver side, notice that under our definitions, a given node will be a member of a single region[4], and can use a single socket for all incoming multicasts (a second socket is also needed, for unicast messages). This allows us to assign a generous amount of buffering space (in experiments we use 4 MB of kernel space, and post 100 simultaneous asynchronous receives with a 64K buffer each). Doing so helps avoid dropped packets when QSM is disturbed by the garbage collector or other processes. Buffering turned out to be critical for performance, and generalizes into a design principle: *data loss avoidance* should always come first, followed by *local recovery*; a *sender retransmission* should be viewed as the last resort. When combined with a scalable loss recovery mechanism described later, this technique allowed us to scale almost independently of the number of groups in the experiment on Figure 3.

What about irregularly overlapping groups? Does this technique have limitations? As mentioned earlier, mapping to regions can backfire: a message sent to a group spanning over **k** regions will be physically transmitted **k** times, in the **k** different multicast groups assigned to the regions. Thus if an application lacks regularity, **k** might become large and a single multicast would require many separate send operations.

Consider the scenario presented in Figure 5. A single sender is multicasting in a group of 110 nodes artificially divided into a varying number of regions, to quantify the performance impact. Since the network is limited to 100 Mbps, QSM becomes network-bound. Moreover, notice that this can happen long before we reach the extreme scenario alluded to earlier, where one could imagine having each process belong to a single region of which it is the only member.

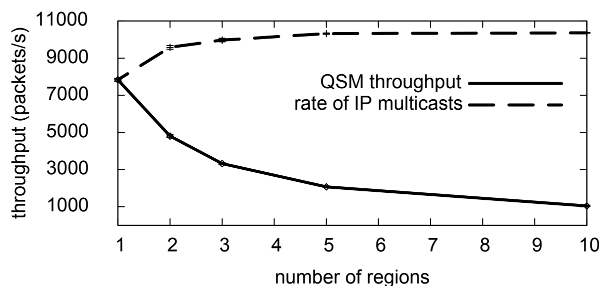

Figure 5. Throughput and IP multicast rate in QSM for the default setting (per-region multicast), in a 110-node group on a 100 Mbps network.

[3] Specifically, each *group view* maps precisely to a set of *region views* (we use a 2-level membership scheme).

[4] More precisely, of only one *region view*, except for brief periods when membership is changing. A node may then temporarily be a member of multiple old views and of the new region view.

To address this class of concerns, QSM offers a special hybrid mode of multicasting. The application can designate that for certain groups, a message should be multicast to a per-group IP-multicast address, but accompanied by a list of per-region control headers for all regions spanned by the group. In effect the single per-group multicast substitutes for a series of regional ones. Recovery/retransmissions are then handled on a per-region basis. This multicast eliminates redundant sends and avoids the bottleneck visible in Figure 5. On the other hand, it can only be used for a limited number of groups, since a node might otherwise need to join many IP-multicast addresses, triggering driver-level filtering, and would also lose the aggregation opportunities associated with regions.

At present, we believe that large applications will fit with the regional mapping approach. In typical large deployments, a process will belong to a single region and at most a few additional multicast channels, hence most messages will be transported using IP multicast with a reasonably large fanout. Of course we could be proved wrong. But until we encounter important applications for which the present mixture of options is inadequate, it seems wiser to opt for simplicity.

2.2. Reliability

What other use can be made of regions? Notice that the nodes belonging to a region exhibit what we shall refer to as *interest sharing*. By definition, all nodes in a region are receivers of exactly the same messages. This allowed us to use a single IP multicast address and perform batching, rate control and other dissemination-related tasks per region. But note that interest sharing is also a good opportunity to perform local recovery. *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. Thus it makes sense to plan for cases in which some members of a region have received a packet that others missed, and to implement local loss recovery mechanisms at this level.

Similarly, suppose that *all* members of a region miss a packet. Rather than having all nodes individually seek a retransmission from the sender, it makes sense to report their aggregated loss in a manner similar to recovery in RMTP, and have the sender retransmit with per-region IP multicast. It also makes sense to coordinate rate control protocol on a regional basis.

This leads to the structure we showed earlier, in Figure 4 (right), where each region runs its own local recovery protocol. ACK/NAK information for regions, aggregated by their local protocols, is then used by a higher-level protocol that runs across regions and includes the sender. At that higher level, regions can be viewed as black boxes: information concerning individual region members is hidden from the protocol running across regions.

Can regional recovery benefit scalability? Note that since all members of a region are members of the same groups, a single protocol can perform loss recovery inside a region, for all groups simultaneously. In fact, in QSM senders number all packets they send to regions on a per-region rather on a per-group basis. Information about the group to which a packet was addressed is required for delivery, but for the purpose of identifying the individual messages, region members can view all packets multicast by a given sender to their region as a single, contiguous sequence. Likewise, their aggregate ACK/NAK information can be calculated, and communicated to the sender, in terms of this sequence, rather than on a per-group basis. This way, a regional recovery protocol, by design, performs independently of the number of groups.

Another benefit is that since, as we noted, every packet sent to a region is destined for all its nodes, all can cooperate in repair of every packet, irrespective of its source. It is therefore reasonable to run a single recovery protocol covering all senders. Note that this could not be the case had region members received different sets of packets, for in such case some of the nodes would be asked to participate in repairs for packets that they were never meant to see, an inefficiency of the sort that led to our concerns about lightweight groups. Accordingly, the QSM loss recovery control packets carry a list of recovery records, one for each sender actively multicasting into the given region. If the number **S** of senders is small, the *size* of control packets grows with **S**, but the *number* of packets does not.

Could we run other protocols inside a region? In fact, every level in the hierarchy could use a different recovery protocol, an idea explored in [1]. Recognizing that what looks good in theory may not work well in practice; we implemented this mechanism and experimented with two different recovery protocols for regional recovery: one based on a token ring, and one using a range of tree-like structures, including balanced and DHT-like trees with different fan-outs. In both cases, we performed local repair among neighbors, and aggregated the regional ACK/NAK information at a

single *region leader*. Our ring protocol was based on a circulating token. The tree protocol resembled RMTP. We represented ACK/NAK information exchanged by our nodes in a compressed manner, as a list of intervals. We experimented with *push*, *pull*, and hybrid schemes.

To our surprise, the main problems we encountered had little to do with a specific protocol; they turned out to be manifestations of the entropy inherent in the system. The subsections that follow amplify on this.

Relative asynchrony. We use the term *relative asynchrony*[5] to refer to the fact that nodes run at different speeds, might receive the same packets at different times or, if using asynchronous I/O, in different orders. This forces receivers to be conservative when classifying packets as missing and forwarding (*push*) or requesting (*pull*) them; if not, unnecessary forwarding destabilizes the system. Relative asynchrony can be an important phenomenon: QSM is preempted or disrupted by garbage collection and other events, and these events may persist for hundreds of milliseconds. Nodes must wait at least this long before deciding that a packet seen at one node but not another was lost.

Bounding state representations. Receiving packets at different speeds and in different order means not just that representing the ACK/NAK information for a single node will be difficult (our encoding as a list of intervals is expensive if packets do not arrive as an uninterrupted sequence), but also implies that aggregations of information over an entire region require more space. As we struggled with this issue in QSM, our control packets grew; with massive losses, packets tens of kilobytes in size arose. The insight is that any complete local state representation will be expensive during periods of instability. Scalable protocols must limit the amount of state exchanged per message. Consequently, QSM limits the information that may be stored in a single control packet by bounding the number of ACK/NAK intervals that a single message may carry. Beyond the limit, recovery is postponed. In situations where only a partial state can be efficiently exchanged, we may have to cope with delays in exchanging state related to loss recovery.

Inherent need for locality. Detailed information about a given node, such as ACK/NAKs, cannot be dragged around the entire region, for control traffic would grow linearly with region size. We found this to be especially disruptive to token ring protocols. Also, carrying state information more than a few hops makes it less valuable, for it becomes stale, e.g. upon receiving a stale NAK, it may not make sense to perform a *push*, since the node that issued this NAK could have already retrieved the missing packet from elsewhere.

To summarize, a consequence of relative asynchrony is that in larger regions, state is more complex, and more costly to exchange, especially over more than a few hops. Furthermore, in larger regions packet losses occur proportionally more frequently and hence it takes more time to acknowledge packets. As a result, buffering overhead grows at receivers in larger regions.

Dealing with the overhead of increasing buffering. To address this issue, QSM splits a region of size $\mathbf{m}$ into $\mathbf{k}$ partitions, $\mathbf{k} \approx \mathbf{m} / \mathbf{r}$, where $\mathbf{r}$ (typically 5) is a *replication coefficient*. Nodes in each partition cache, for the purpose of loss recovery, $\mathbf{1/k}$ of the incoming messages. This reduces buffering overhead linearly with the region size. The smaller the value of $\mathbf{r}$, the higher the risk that a packet is missed by all caching nodes, and that the recovery involves the sender. Larger values of $\mathbf{r}$ require more buffering overhead.

Since exchanging state over multiple hops is expensive, nodes in the same partition should be clustered close to one another, and since they cache the same packets, it makes sense for them to cooperate on recovery. The same thinking suggests that state exchanged by nodes in *different* partitions should be aggregated. Since partitions are small, the natural choice for a protocol to run among nodes in the same partition is token ring. We favor simplicity; hence we also use a token ring protocol across partitions. The resulting design is shown in Figure 6 and Figure 7. In essence, QSM's hierarchy of groups and regions has gained a level.

In QSM, a *region leader* generates the token at a fixed rate. The token goes around the region in loops, returning to *partition leader* before moving to the next partition. While circulating around a partition, the token contains ACK/NAK information only for the messages cached in this partition. Partition members use this information for local repair via both *push* and *pull*. If node failures or recoveries occur, the entire region and partition membership is updated (a separate service provides consistent membership updates: QSM Global Membership Service (GMS)).

The token can also accumulate aggregated ACK/NAK information for this partition (for the sub-sequence of packets cached in it). When the token returns to the

[5] Section 3 will revisit *relative asynchrony* from a different perspective.

partition leader, this aggregate ACK/NAK information is used to issue NAKs, sent by the partition leaders directly to the sender. While circulating, the token also accumulates information such as the maximum sequence number seen in the region, maximum contiguous packet number cached by all nodes in each partition etc.

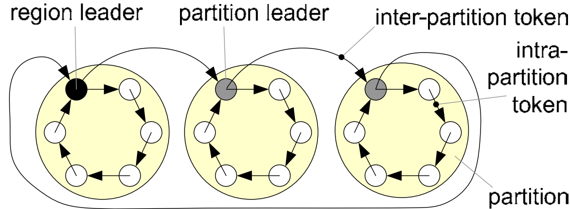

Figure 6. QSM's partitioned token ring protocol.

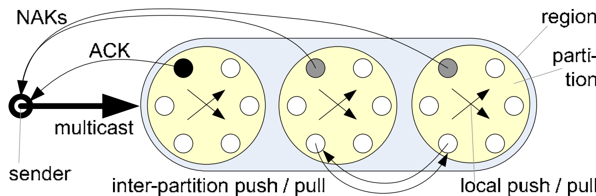

Figure 7. Loss recovery in QSM. If possible, lost data is recovered locally (within a partition); otherwise, a NAK solicits retransmission by the sender.

After the token returns to the region leader, it creates an ACK and sends it directly to the sender. The sender reacts to ACKs by cleaning up messages and to NAKs by retransmitting. The token is also used for garbage collection. Finally, if a node is missing packets cached in partitions different than its own, it sends *pull* requests to nodes in those partitions. Nodes respond to pulls with forwarding when the requested packets become available.

In the interest of brevity, the above summary omits many details of the actual protocol. The mile-high summary, however, is that the token ring protocol forms the core of the QSM recovery and cleanup protocol. As tokens circulate, a variety of status structures associated with each partition and each region are first created, and then subsequently updated.

Figure 8 is a timeline illustrating some of these events at a typical receiver node in the middle of a multicast experiment. We see data arrival times (*multicast*), discovery of losses (*nak*), recovery of lost packets (*forwarded*), a point up to which messages may be considered as missing (*cutoff*), an estimate of the oldest live packet in the system, and a garbage collection frontier.

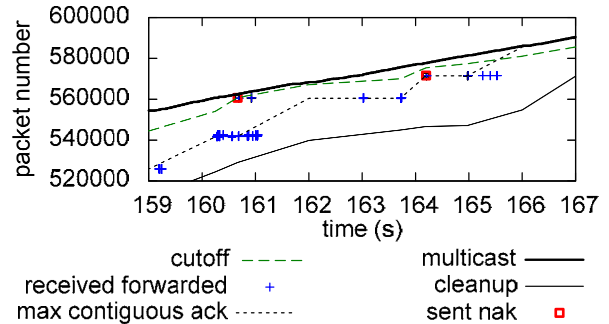

Figure 8: QSM timeline at a receiver.

The temporal dynamics of the scheme are further explored in Figure 9 and Figure 10. Figure 9 illustrates the latencies measured when the system is sending multicasts at varying data rates, and the delay until cleanup occurs. At higher data rates QSM works more smoothly and avoids scheduling delays and blocking system calls due to pipelining effects, thus latencies are lower. With larger packets, however, the CPU load becomes a bottleneck and latencies rise again. The reader will recognize this effect in the latency distributions on Figure 10.

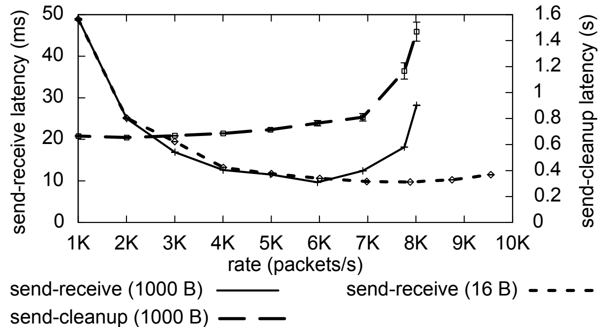

Figure 9. Send-receive and send-cleanup latencies as a function of the sending rate (1 sender, 110 nodes).

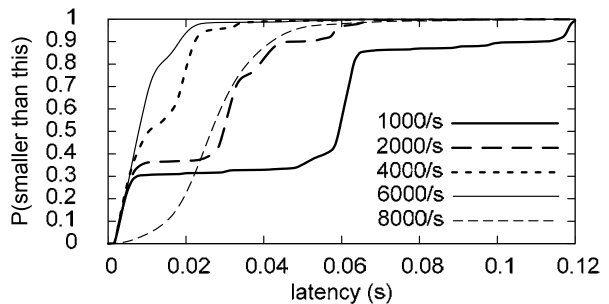

Figure 10. Cumulative distribution of send-to-receive latency for different sending rates.

3. Performance

We now shift attention to the question of peak performance. Returning to Figure 2 and Figure 3, recall that

JGroups actually didn't perform well even at small scales. Neither, in fact, did earlier versions of QSM.

What factors limit multicast performance, and how can they be eliminated? With large numbers of groups, it turns out that both JGroups and the earliest versions of QSM were CPU-bound, caused in large part by frequent context switching. In QSM, the early versions of the system were multi-threaded, and the regional mapping protocol required a fairly complex synchronization when the associated data structures were accessed. Fine grained locking resulted in additional overheads, yet reducing lock granularity only triggered high levels of lock contention. Furthermore, our system suffered from stalls and various subtle deadlock conditions, resulting from a combination of locking and flow control. Our observations about relative asynchrony make it clear that a high level of concurrency is necessary for high performance, yet pre-emptive scheduling is disruptive, particularly because with the exception of garbage collection, most tasks performed by QSM are short, predictable and terminating.

Is multithreading really the best way to achieve a high level of concurrency? Is preemptive scheduling needed for an event-driven system such as our reliable multicast protocol stack? QSM's performance was so poor in our original multi-threaded implementation that we decided to re-implement it using the single-threaded model shown in Figure 11. We bind all send and receive sockets to a single I/O completion port, a queue-like structure that the Windows kernel employs to report the success or error for asynchronous I/O operations, such as transmission, or a received packet. An I/O completion port can be polled in either a blocking or a timed non-blocking manner. An *alarm queue*, implemented in C# as a splay tree, holds timer events.

A single *core thread* switches between processing events in the alarm and I/O queues, in a round-robin fashion, in batches, up to the time limit (*quantum*) assigned to the given queue (in the experiments, we use 50ms quantum for I/O, and 5ms for timer events). Application threads issue their requests to the core thread either by placing completion events on the I/O queue, or via a non-blocking queue of *downcalls*. In particular, applications can register with the core thread their intent to send data using a construct we term a *feed* (a form of *pull* interface). A *feed* can produce packets to send on demand up to the specified limits. Our core thread polls the registered feeds when it is ready to send a new portion of data. When a feed reports that it has no more packets to send, it is *deactivated*, and it must be explicitly *reactivated* by the application. This reduces polling overhead. The decision of when the core thread can transmit data is driven by resource limits and simple concurrency and rate control schemes, built into that layer for efficiency.

The decision to use the *pull* model with feeds was a matter of efficiency, permitting us to send multiple packets at once and minimize buffering between QSM and the kernel. A further advantage, discussed below, is that pulled data is always fresh.

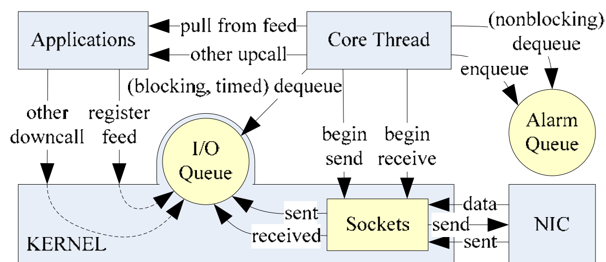

Figure 11. The architecture of QSM. A single thread processing all events using our own scheduling policy.

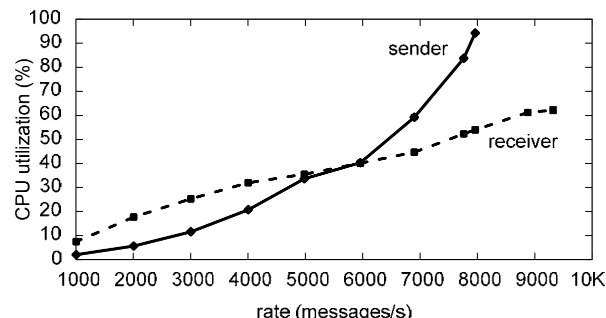

Figure 12. Processor utilization in QSM. The highest rates (beyond 8000 packets/s) use 2 senders.

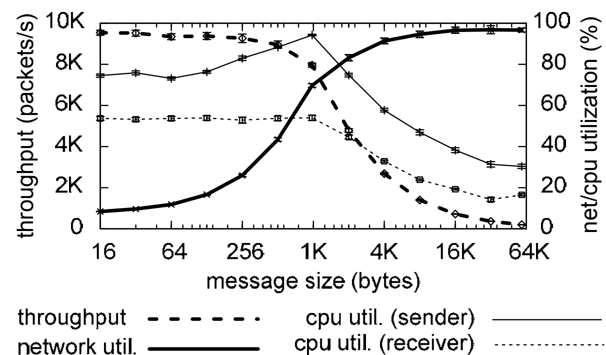

Figure 13. Throughput and CPU loads as a function of message size.

Moving to a single-threaded model made a big difference. As shown on Figure 12, our current system can send and receive thousands of kilobyte-sized packets/s while utilizing a fraction of a 1.3 GHz CPU. In con-

trast, the multi-threaded version exhausted the CPU at 1/3rd of this data rate.

Figure 13 and Figure 14 explore the impact of message size on various metrics. As messages grow, throughput can be maintained until the interconnect reaches its full capacity. Processor utilization at the sender grows up to some point as messages become larger, partly due to memory allocation and zeroing overheads, and partly reflecting the costs of a managed framework. These costs are also visible if we focus on the latency from when a message is sent to when an average receiver hands it off to the application in a test running at the peak sustainable throughput rates. Of course, the figure illustrates the worst case: latencies are much lower for a single multicast sent when the system is idle.

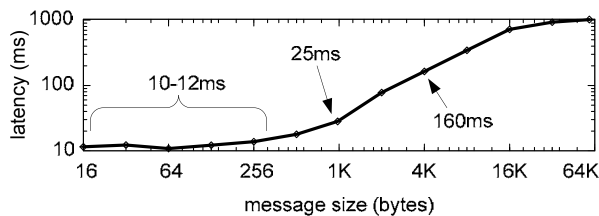

Figure 14. Average send-to-receive latency in a 110-node group as a function of message size.

Where does the single-threaded system spend CPU time? Profiling reveals that in a typical scenario, on the platform we tested, QSM spends about 30% of CPU time in our code, 60% in CLR and in Win32 DLLs (the remaining 10% is unclear). Functionally, the sender spends 80% of its time sending (including allocating messages in the application, going down the protocols stack, concurrency and rate control, and finally, socket operations), and 10% processing incoming messages: tokens, ACKs, NAKs, performing cleanup etc. A more detailed analysis of major overhead components reveals that 20% of the CPU time is spent on allocating messages in the application, about 10-15% initiating the transmissions, roughly the same for software loopback and updating sender structures, only 8-9% on serialization (we use scatter-gather), and the same on processing ACK/NAKs by the sender. Scheduling code, including interacting with the I/O completion port, and using our alarm queue etc., amount to less than 10%. Most of the overhead appears to come from data structures, such as the .NET dictionaries, which we use quite extensively. It is likely that an intensive optimization effort could reduce many of these costs.

Is a round-robin, FCFS policy the ideal way to process events within QSM? How does processing order affect performance? The single-threaded architecture forces the system to decide which events to process first and how much time to assign for each category of event. Our intuition was to prioritize I/O, under the assumption that each dropped packet represents a significant cost, and most of the tens to hundreds of thousands of timer events that typically reside on the alarm queue are either not time-critical, or will eventually be cancelled. With this in mind, we experimentally gave I/O unconditional priority over timer events, but performance plummeted: our data structure cleanup code - was suffering from starvation. The solution we settled upon was discovered almost accidentally, during debugging.

Recall that QSM's recovery and cleanup protocols are based on per-region tokens. We discovered that QSM relies far more heavily on the *regularity* of token circulation than intuition suggested. In particular, in large systems, particularly with multiple senders, the time to circulate a token around the region would occasionally grow high. This triggered a costly form of priority inversion. Nodes transmitting or receiving lots of data would have large numbers of I/O operations queued, with incoming or outgoing tokens or other control packets being just a few in a long sequence. Processing those operations sequentially in each node led to an increased latency for control packets, to which tokens are particularly sensitive, as the latencies on nodes in this case would add up. Increased latency for control packets, in turn, would occasionally trigger retransmissions and other actions that would further increase the load, thus closing the positive feedback loop, until the moment when sliding windows on senders are full, and the entire system quiesces for a period of time required to repair and issue ACKs. At high data rates, our system was prone to unstable, explosive behavior.

The upshot of this rather complex picture is that we observed runs with long bursts at a rate of 8000/s followed by equally long periods where no new packets were sent by any of the senders and the entire system focused on recovering from losses (we'll see similar behavior, although triggered by a different phenomenon, in Figure 22). This led us to the realization that in a reliable multicast system, there are two distinct layers, the control layer and data layer, which must be treated differently. Specifically, it is essential to provide the control layer with a high quality of service.

Why not just prioritize the processing of control packets? We tried a very simple technique, where all incoming I/O is pre-processed just to recognize the type of operation and assigned to one of several priority queues. The actual processing of events in the queues

is deferred to the time where no new I/O completion events are available. Processing then continues in the order of decreasing priority. This idea was inspired by the manner of handling interrupts in operating systems. We assigned priorities using a few simple rules: first we process all the unicast traffic, which represents the control layer, followed by the multicast, and we process received data before initiating any new transmissions to minimize packet loss.

Starvation and priority inversion can occur on the "sender" side too. We gained a further insight when experiments revealed that QSM would occasionally enter a loop of endless and mostly unnecessary forwarding, where the forwarding itself would destabilize the platform even further. The problem turned out to be triggered when a long sequence of forwarded data packets for local repair was placed in an outgoing buffer ahead of a token or other control packet, often unnecessarily, in a situation where (due to some delay, or packets arriving out of sequence) a node would decide that it was "missing" packets that were in fact already in transit. The resulting duplicate messages delayed the token, yet swift exchange of the token is the key to bringing QSM's nodes back into synchronization. Furthermore, control packets delayed this way would occasionally contain stale, old information, resulting in wasteful overheads and occasionally leading to inconsistencies if packets are processed out of order.

Besides reinforcing the view that control traffic needs priority handling and QoS, this made us realize the importance of *freshness* and what we might call a *relative synchrony*. It is critical for nodes to maintain an up to date view of the state of their peers (*relative synchrony*), in order to avoid costly mistakes such as unnecessary forwarding. Notice that the goal of relative synchrony is in tension with the reality of relative asynchrony. Striking the right balance is key to achieving high throughput rates.

Ensuring that information about peer states is fresh. In the situation just summarized, the problem arises from buffering delays. Accordingly, we eliminated most buffering from our protocol stack, extending the *pull* interface exposed by our socket layer to the rest of the system. In the resulting structure, at every send socket, rooted is a tree of *data feeds*. Each feed represents an element of a protocol stack; child nodes in the tree register their intent to send with parent nodes and parent nodes poll child nodes to pull data. At the root, polling is driven by socket concurrency and rate control policies. At intermediate nodes, multiplexing among many attached feeds is done in a round-robin fashion. When an element of our protocol stack responsible for peer-to-peer forwarding decides to forward data to a node, it simply registers its intent to send with the lower layers. Only when its turn to send comes, would a forwarding packet actually be created, registered and serialized; the same applies to most messages. In the resulting scheme, all packets we send are fresh, because they are created *just in time* for transmission. We still use buffering, but to a limited extent, to reduce the call stack overheads.

A simple scheme, analogous to our scheduling quanta, ensures fairness and prevents starvation. The registered feeds, polled in a round-robin fashion, are each given an opportunity to send up to the limits permitted by the concurrency/rate controllers, and subsequently moved to the end of the queue.

Can we generalize from this experience? We've discussed a series of issues more or less in the order we encountered them. However, our experience fits into a general pattern. Whether the cause was priority inversion, scheduling, or buffering, poor performance in QSM was typically manifest by *convoys*: long sequences of packets or requests waiting in a buffer or queue for delayed, sequential processing. We can summarize the code changes discussed earlier by recognizing that on the send side, we eliminated convoys by creating requests lazily just-in-time, and on the receiver by moving I/O completion events them to a non-sequential data structure, and deferred processing.

Seen this way, the process of maximizing throughput in QSM evokes a phenomenon familiar to any reader: that of driving in heavy traffic on a crowded highway. Highway speeds are highest when vehicles are least dense. But as the numbers of vehicles rises, drivers have more difficulty maintaining safe distances from one-another and this eventually triggers oscillations in the sustainable speeds. Small obstructions, like a pothole in the pavement, can be enough to slow a car down, and this will ripple through the traffic stream and may trigger a jam. QSM faces a similarly delicate balance when moving high volumes of data at high speeds.

QSM, at its highest throughput rates, functions like a highway down which packets race with some small separation. When this stream is delayed by garbage collection, scheduling, data loss, etc., bursts of packets tend to pile up, and once such a problem occurs, the original smooth separations may be hard to restore. Our changes were of several kinds. Some fill in the potholes by eliminating obvious sources of slow-down. A second group lets QSM's emergency vehicles (tokens

and control packets) get through faster and more smoothly, correcting problems so that congestion eases and the backlogged packets can clear. With luck, this prevents an insurmountable slowdown from arising.

With these changes, QSM is ultimately CPU limited, by the sender (which does more work than the receivers). We can see this in Figure 15, which shows that as the message rate reaches the maximum, QSM alarms begin to trigger late in the sender, and the token round-trip times climb sharply. Recall from Figure 12 that this is precisely when CPU loads on the sender approach 100% at very high data rates. QSM uses several mechanisms to tolerate late alarms. For example, the rate controller overshoots target rates to "catch up".

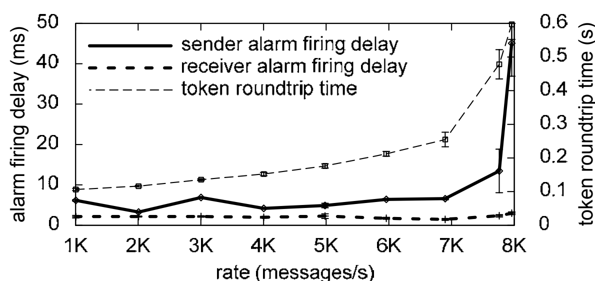

Figure 15: Late alarms and slow tokens in a sender.

4. Stability

So far we've focused on optimal conditions, where senders are undisturbed by messages from other senders, maximum achievable sending rates were found by trial and error, no crashes occur etc. But real systems aren't always so lucky. **What if QSM is misconfigured or disturbed? Will performance degrade smoothly? Or will the system collapse?**

The most common disturbance is data loss. In real systems, losses occur in batches. How does QSM respond to a batch of losses? To find out, we let a single node send at the maximum speed to a single group of varying sizes. After the first half of the experiment, we start to periodically disturb a single receiver. The choice of receiver matters, but brevity limits us to discussion of a single experiment, in which receiver is in a partition "close" to the sender. Every 10s, the disturbed receiver drops all incoming packets (data and control) for a period of 1s, and then resumes normal operation. We measure throughput, *send-to-receive* and *loss-to-repair* latencies (Figure 16). It should be noted that this pattern of bursty loss is easily provoked on our experimental platform and that episodes of bursty loss are also common in routed LANs, because routers drop packets to signal congestion to the TCP windowing algorithm.

The disturbance decreases throughput by 10-15%, close to the amount of time "wasted" by dropping packets, peaking in the largest group where throughput decreased by 25%. There is an increasing trend: in QSM, larger groups are clearly more sensitive to disturbance. Send-to-receive latency, normally ~25ms (not shown), is now between 660ms in small and 1.7s in the largest groups, and the increasing trend is evident. Notice that although one might have assumed that having smaller numbers of nodes would usually be fastest, the loss-to-repair (recovery) latency for the smallest groups (of fewer than 30 nodes) is larger than for mid-sized groups, reflecting the advantages of parallel recovery. As shown on Figure 17, shorter sequences of loss have a proportionally smaller impact on QSM.

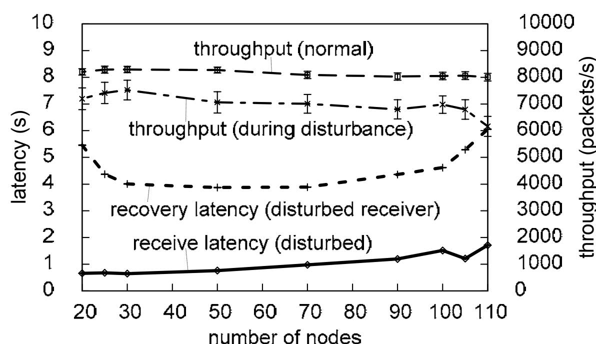

Figure 16. A single sender multicasts at the maximum rate. Every 10s, a selected node simulates a burst of losses by dropping all incoming packets for 1s.

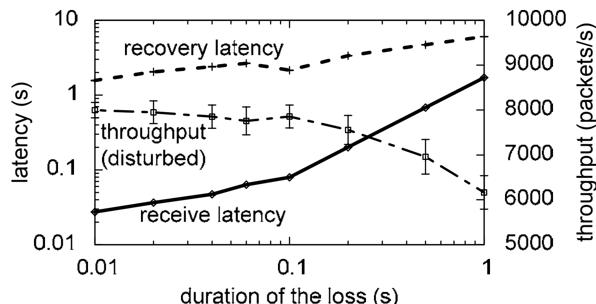

Figure 17. Like above, but the duration of the sequence of losses varies. Group size 110, maximum send rate.

These findings suggest to us that while QSM is much more scalable than JGroups (and, we believe, than anything else reported in the literature), the costs associated with loss recovery grow moderately with the group size, and hence QSM would also encounter scalability limits in large real deployments where disturbances and "flakey" hardware are not uncommon.

Although node failures are relatively rare, it is natural to ask: **How does QSM react to a node crash?** To

find out we modified one receiver to terminate abruptly in the middle of the experiment. On Figure 18, we illustrate the QSM behavior by drawing the total number of packets: *sent*, *received* at a selected correct node, and *completed* (acknowledged by all nodes and cleaned up at the sender).

The failure breaks the token ring, and consequently, the sender cannot receive aggregate ACKs and cleanup the successfully delivered requests, hence the *cleanup* line stops progressing immediately following the crash, while new multicasts continue uninterrupted until the send-side limit on the number of pending multicasts is reached. In this scenario, we setup our failure detector to consider nodes as faulty after 10 seconds of unresponsiveness. With a more aggressive failure detector setting, QSM can reconfigure itself faster.

After the node is recognized as faulty, a membership change occurs. The sender briefly pauses to allow receivers to recreate their structures, and resumes. The old token ring structure will now be bypassed, and a new structure established. While the new token starts circulating, the old token resumes for a few rounds until all nodes are in sync, then quiesces. Within a few seconds after detecting the failure, the sender is back at full speed, and all old multicasts have been acknowledged. Send-receive latency stays undisturbed in this scenario. Obviously, the degree of disruption depends on how promptly the failure detector can recognize the failure and how much the sender slows down. With sufficient buffering space, or if the failure is detected rapidly, multicast can proceed almost undisturbed.

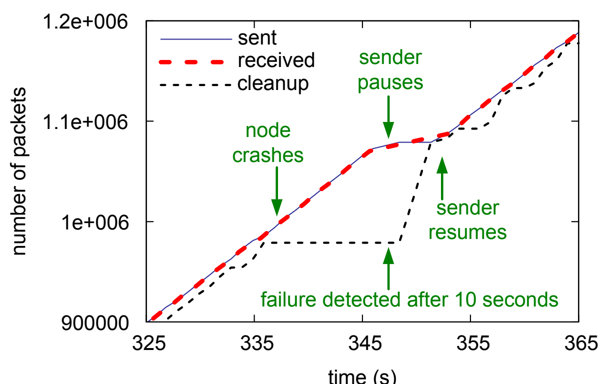

Figure 18. Node crash in a 110-node group, recognized by the failure detector after 10s. Multicasting at 8000/s.

This behavior turns out to be independent of group size, which may seem surprising. **Shouldn't crashes be inherently more disruptive to larger groups?** Indeed they are, although for a less obvious reason. In QSM, distributing a membership change takes a few seconds. While the mechanism we used is simple and slow and might be improved, in a busy system it will never be instantaneous. Nodes process the membership updates and adjust their structures at different times. It often happens that the sender processes the change early and starts multicasting in the new view before receivers have a chance to learn about the change, and create the appropriate structures, hence they would drop the incoming packets as unrecognized. To remedy this, we let the sender wait for 3s before any transmission to a newly created IP multicast address (the reader can see this in Figure 18), and have the receivers buffer a limited number of unrecognized packets and retry processing them after a short timeout. This was sufficient for the scenarios we tested, but in a larger group, either the timeouts or the limit on buffering would certainly need to be higher. And conversely, in small groups neither the buffering, nor suspending multicast is necessary.

What about nodes joining groups? A single node join in the middle of the experiment, in the same scenario as on Figure 18, is even cheaper, for there is no disruption to the circulation of the token. Our GMS processes the membership changes in batches, typically every 10-20s, hence churn is not a big issue. We omit a detailed discussion of joins and churn for brevity.

While a crash is an important scenario, in reality nodes are much more likely to be slow, or simply overloaded, e.g. as a result of contention with other applications, paging, system upgrades, or even flakey hardware. If an unresponsive node is assumed by the failure detector to be faulty, QSM behaves as in the case of crash. **What if a node is slow and unresponsive, but isn't recognized as "faulty" by the failure detector?** To find out, we disable the failure detector, and we make a single receiving node freeze for 10 seconds in the middle of the experiment, then resume (Figure 19). Immediately after resuming, the process retrieves a few requests from the network buffers, but most data is lost and is forwarded by other receivers. Many requests are unacknowledged, thus causing the sender to fill its upper threshold of 100,000 pending multicasts and pause. This allows the disturbed receiver to catch up, the number of pending requests falls below the low watermark, and the sender resumes multicasting. The entire period of disturbance is more than twice as long as for crashes.

Notice that in contrast with an outright crash, a freeze ultimately triggers an extended data recovery episode. **Is a freeze more disturbing to larger groups?**

To quantify the degree of disturbance, we use a metric that we shall call *cumulative delay*, and that represents the additional time it takes to multicast a fixed number of messages in a disturbed run, as compared to an undisturbed run (see Figure 20). Surprisingly, in very small groups the disturbance is up to three times more pronounced, perhaps because of the opportunity for parallelism in the recovery code. This figure recalls Figure 16, where the recovery latency for a small disturbed group was larger than for one of moderate size.

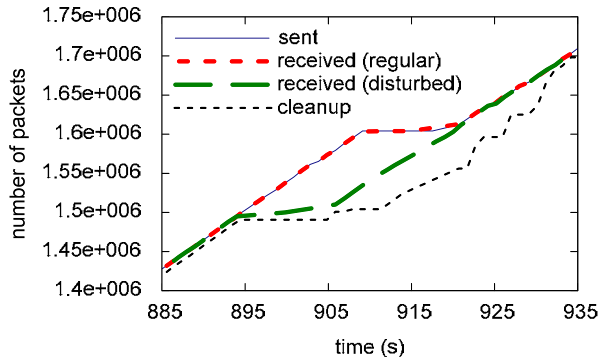

Figure 19. One receiver in a 100-node group becomes unresponsive for 10 seconds without triggering a membership change. Multicasting at 7500/s.

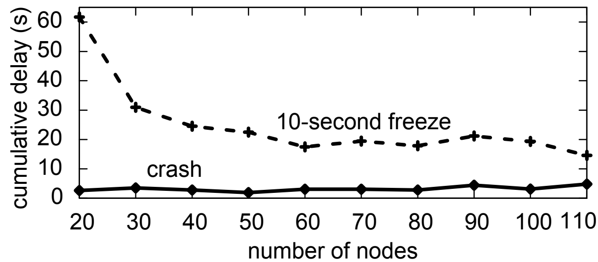

Figure 20. Cumulative delay, extra time needed to send after various disturbances as a function of group size.

Figure 21 examines traffic at the disturbed node. As the episode lengthens, protocol overhead (everything except original delivery of data by IP multicast) sent and received by the node grows steadily. The curve is similar whether we measure packets or bytes. Within the overhead, the fraction attributable to data recovered from unperturbed peers also grows steadily. (We also instrumented unperturbed nodes but didn't include this data on the graph: even during disturbances, if the region size is moderately large, 99% or more packets are received by IP multicast and overheads of all forms are below 1 %.)

Recall that thanks to partitioning, the larger the regions in QSM, the smaller fraction of packets cached at every single node. Conversely, if we consider a sequence of lost packets, in larger regions packets in this sequence will be cached by a larger number of partitions. Consequently, with larger regions QSM runs a more parallel recovery. A worst-case recovery scenario would thus involve a very small region in which one node has lost many messages and there are very few other local nodes from which they can be recovered.

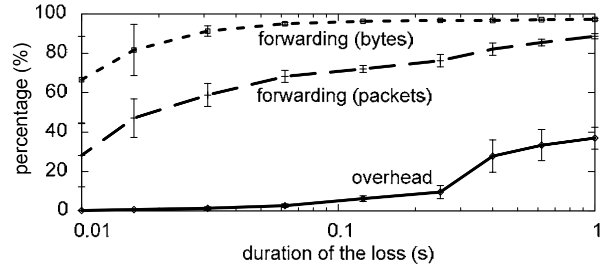

Figure 21. Breakdown of network traffic at a recovering node after a 10s "freeze". The fraction of overhead in the traffic present on the link, and the contribution of forwarding to the overheads are shown. The remaining overhead comes mostly from tokens and ACK/NAKs.

What happens if we overload QSM? As was discussed earlier, QSM fetches data by doing up-calls to the application, and hence there are many configurations in which the system simply cannot exceed certain data rates. However, with two senders that run at the maximum rate, QSM can exhibit load surges exceeding the capacity of our network interfaces.

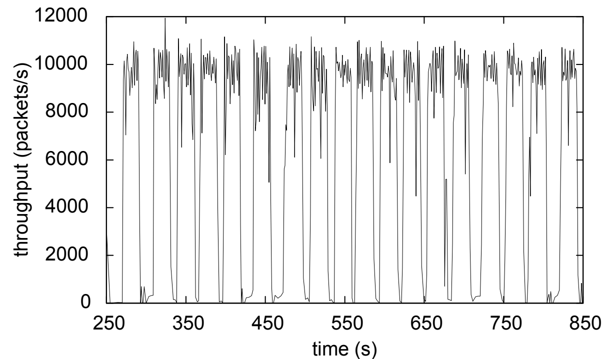

Figure 22. Oscillating combined throughput with two senders exceeding the maximum sustainable combined rate of 9300 packets/s by a small margin (110 nodes).

This is illustrated in Figure 22, where we see a fluctuation in receive rates as the system sends a burst, overwhelms receiver interfaces and experiences a brief episode of very high data loss. The problem seems to be that our network interfaces are unable to handle extended sequences of back-to-back IP multicast packets, which this pattern of bursty sending can trigger (a form

of what are called “multicast storms” in the literature). Thus a high but random loss rate occurs, system-wide. Eventually, the sender runs out of buffering space and pauses. Meanwhile, nodes detect loss and recover the data locally. When the sender finally manages to clean its buffers, it resumes and the pattern repeats.

If we pull back the covers, most recovery is occurring through point-to-point local recovery, because most packets were received by at least one node in each partition of each region. Had the entire partition missed the data, an IP multicast from the sender would have recovered it in a single action, and the impact on throughput would have been far less pronounced.

5. Conclusion

We believe that QSM is the first multicast platform designed specifically to maximize steady-state throughput while scaling in both numbers of overlapping multicast groups and numbers of members. The system is intended for use in very large data centers, and we have little doubt that surprises await as real applications begin to use the software. However, early results are very encouraging.

As mentioned early in the paper, work on QSM is ongoing. Although the current version of the system is available for download, we are now extending the platform with a tool to assist applications in responding, RPC style, to incoming multicasts (a single reply is easy, but all-to-one reply patterns require some form of aggregation), a virtual synchrony fault-tolerance layer (optionally selectable on a per-group basis), integration with the .NET type system, and support for a publish-subscribe API as well as other end-user presentation options. We also hope to support a group-key security architecture. Results will be reported elsewhere.

Even at this “early” stage, the development of QSM has not been a simple linear process. As described in this paper, we repeatedly confronted situations that surprised us, confounded intuition, or forced us to balance between competing tensions. Much of the code has been re-implemented at least once. Early versions didn’t perform very differently than prior platforms, and as we worked around the bottlenecks, suffered from fragility and were prone to convoy phenomena that triggered oscillatory behaviors and throughput collapse. Indeed, as was seen in Figure 22, the current system can still be pushed into degraded behavior, although we believe that this figure represents a worst-case scenario that is especially hard to provoke.

QSM scales quite well, although we have also seen that it has its own limits. But while improved scalability is important, the more interesting contributions of our effort may actually be the systematic study of scalability, and the lessons learned by repeatedly hitting limits, tracing them to their sources, and then finding ways to work around them. We hope that this methodology may prove useful even to developers facing scalability challenges in domains remote from reliable multicast.

The initial version of QSM is available to the public from our web site [7].

6. Acknowledgements

Our work reflects a great many suggestions, comments and ideas from colleagues here at Cornell. We are particularly grateful to Mahesh Balakrishnan, Danny Dolev, Maya Haridasan, Tudor Marian, Robbert van Renesse, and Einar Vollset.

7. References

[1] K. Ostrowski and K. Birman. Extensible Web Services Architecture for Notification in Large-Scale Systems. To appear in IEEE ICWS 2006. http://www.cs.cornell.edu/projects/quicksilver/pubs.html

[2] K. Birman. A review of experiences with reliable multicast. Software Practice and Experience, 1999.

[3] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. IEEE/ACM Transactions on Networking, 1997.

[4] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. INFOCOM, 1996.

[5] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread Toolkit: Architecture and Performance. 2004.

[6] B. Levine, D. Lavo, and J. Garcia-Luna-Aceves. The Case for Reliable Concurrent Multicasting Using Shared Ack Trees. ACM Multimedia, 1996.

[7] www.cs.cornell.edu/projects/quicksilver/

[8] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. (1998).

[9] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System (1993).