Implementing High Performance Multicast in a Managed Environment

Krzysztof Ostrowski
Cornell University

Ken Birman
Cornell University

Abstract

Component integration environments such as Microsoft .NET and J2EE have become widely popular with application developers, who benefit from standardized memory management, system-wide type checking, debugging, and performance analysis tools that operate across component boundaries. This paper describes QuickSilver Scalable Multicast[1] (QSM), a new multicast platform designed to achieve high performance in managed environments. Memory-related overheads and phenomena related to scheduling are shown to dominate the behavior of the system. We discuss techniques that helped us to alleviate these problems, and argue that they reveal general principles applicable to other kinds of high-data-rate protocols and applications in managed settings.

1 Introduction

A component integration "revolution" is transforming the development of desktop applications. Platforms such as Windows .NET and J2EE promote an application development style in which components are implemented independently and heavily reused. By standardizing memory management and type checking, these platforms enable safe and efficient cross-component method invocations, avoiding overheads associated with protection boundaries, argument marshaling, copying and un-marshaling.

Broadly, our project is interested in leveraging these benefits to help developers implement robust and scalable computing services that will run on clusters or in datacenters. Early users of our platform are creating applications in areas such as parallelized data mining, event stream filtering software, and scalable web services.

Developers of clustered services need reliable multicast protocols for data replication; and in light of our broader goal of leveraging the power and component integration features of a managed framework, the multicast technology must run in a managed setting. But little is known about high-performance protocols in managed environments. It is interesting to realize that although Microsoft promotes end-user application development using C# in its .NET framework, the company's own products are still implemented primarily in unmanaged C++. By building XYX in the recommended manner, we found ourselves breaking new ground.

The multicast protocols employed by QSM were designed for performance and scalability, incorporating a mixture of new ideas and ideas drawn from prior systems. Nonetheless, the aspects on which we focus here reflect architectural responses to scheduling delays, overheads associated with threads, and costs arising in the memory management subsystem. Over the period during which QSM was developed (two years), these had pervasive consequences, forcing us to redesign and recode one layer of the system after another. For example, the original system was multithreaded, used synchronous I/O calls and was rather casual about buffering and caching; the current system is single-threaded, uses asynchronous I/O, and obsessively minimizes memory consumption.

Today, QSM (finally) performs well and is stable at high data rates, large scale and under stress. The finished system achieves extremely high performance with relatively modest CPU and memory loads. Although our paper is not "about" setting performance records the absolute numbers are good: QSM outperforms the multicast platforms we've worked with in the past – systems that run in unmanaged settings.

This paper won't tell the blow-by-blow story. Instead, we use QSM in a series of experiments that highlight fundamental factors. These reveal linkages between achievable performance and the costs and characteristics of the managed framework. Doing so sheds light on the challenges of working in a kind of environment that will be more and more prevalent in years to come. Our insights should be of value to developers of other high-performance communication and event-oriented systems. To summarize:

1. We propose a new "positioning" of multicast technology, as an extension of the component integration features of the Microsoft .NET managed runtime environment.
2. Although we started with a sophisticated multicast protocol, experiments reveal a series of problematic interactions between its high-speed event-processing logic and the properties of the managed framework, which we document.
3. We addressed these and achieved high performance by making some unusual architectural decisions, which we distill into general insights.

---

[1] This research was supported by AFRL/IF with additional support from AFOSR, NSF, I3P, and Intel. Address: Department of Computer Science, Cornell University, Ithaca, NY 14850, USA; Email: krzys@cs.cornell.edu, ken@cs.cornell.edu

The embedding of QSM into Windows yielded an unexpected benefit: it enables what we are calling "live distributed objects." As the term suggests, these are abstract data types in which content evolves over time. When an application binds to a live object, the current state of the object is imported and the object can send and receive updates at high data rates. An object could be a place in a game like Second Life, a media stream, a publish-subscribe topic, a shared file, etc. Live objects are a natural and powerful idea, and we plan to pursue the concept in future work. However, this use of QSM raises performance and scalability issues beyond the ones seen in our original target domain. For that reason, we leave detailed discussion of the idea for the future.

QSM has been available for free download since mid-2006, and it has a number of users, most working on clustered computing. For example, one large project is pairing QSM with high-speed event stream filtering and data mining system to obtain a scalable, cluster-hosted service capable of handling very high event rates.

2 Usage cases

Use of QSM in our target settings gives rise to potentially large numbers of overlapping communication groups. As we have seen, the primary goal is to support data replication in scalable, componentized services, in which sets of components are interconnected and cooperate to perform requests. To minimize latencies, components sets are normally co-located; when a service is replicated, each of its constituent components will need to replicate its portion of the service state. If QSM is used to disseminate updates, this results in a pattern of communication groups that are exactly overlapped: each replicated component will have one or more associated groups, delivering update streams to its replicas.

Of course, a datacenter will typically host many services, each with a disjoint set of components, and often deployed on disjoint sets of nodes. In cases where two services are co-located on the same node, we'll still see heavy overlap, but unless the degree of replication is identical, there may be two cases: nodes that host both services (and hence both sets of QSM groups), and nodes that just host one of them.

Cluster management systems use groups for purposes other than component replication, such as tracking node status and launching applications: these groups will span large numbers of nodes, perhaps the entire cluster. Such groups overlap with everything. The result is an environment in which there will be a hierarchy [6] of overlapping groups (Figure 1). QSM is highly effective in supporting this style of use.

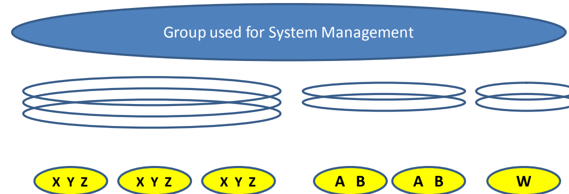

Figure 1: If sets of components are replicated, the associated multicast groups overlap hierarchically.

The foregoing is the *primary* use scenario for QSM, but may not be the *only* one. One could imagine an approach to laying out components on a cluster that would result in irregular layouts of groups. QSM can support such layouts, at least to a degree, but for reasons of brevity the discussion in the remainder of the paper focuses on regular, hierarchically structured communication groups with extensive and regular overlap. Initial users of our system haven't had any difficulty with this constraint: knowing QSM is particularly effective with regular layouts, they just design to favor regularity.

3 Architecture

Reliable multicast is a mature area, but a review of prior systems convinced us that no existing system would work well in the scenarios targeted by our project. This forced us to build a new system that combines features from a number of prior systems.

Our decision not to use some existing multicast system reflects a number of issues. Most prior multicast systems were designed to replicate state within just a single group at a time, for example a single distributed service. Some don't support multiple groups at all, while others have overheads linear in the number of groups to which a node belongs. For example, we looked at JGroups [2], a component of the JBoss platform which runs in a managed Java framework. JGroups wasn't designed to support large numbers of overlapping groups, and if configured to do so, overheads soar.

There has been a great deal of work on P2P pub-sub and content delivery platforms in recent years, often oriented towards content filtering in document streams. A good example is Siena, a system that has become popular in WAN settings [3]. However, systems in this class incur steep overheads associated with content filtering. Moreover, messages often follow circuitous routes from source to destination, incurring high latency. In high performance settings, these factors would degrade the performance of the replicated application.

The Spread multicast system implements "lightweight" groups [1, 4]. The groups seen by applications are an illusion; there is really only one

process group, consisting of a small set of servers to which client systems connect. Each application-level multicast is vectored through a server, which multicasts it to its peers. These filter the ordered multicast stream and relay messages back out to receivers. This approach can support huge numbers of groups with irregular overlap patterns, but the servers are a point of contention, and the indirect communication pathway introduces potentially high latencies.

These considerations convinced us that a new system was needed. QSM implements a approach similar to Spread's lightweight group abstraction, but without a separate server group. We define a *region of overlap* to be a set of nodes with approximately the same group membership (Figure 2). Under the assumptions of Section 2, our cluster should be nicely "tiled" by regions. QSM uses regions for multicast dissemination and for recovery of lost packets, employing different protocols for each purpose.

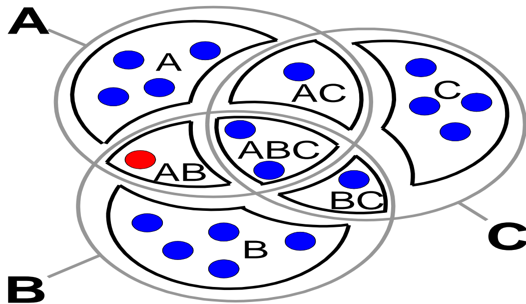

Figure 2. Groups overlap to form regions. Nodes belong to the same region if they have similar group membership.

For initial dissemination, QSM currently uses an unreliable IP multicast. Since a single group may span multiple regions, to send to group G, a node multicasts a message to each of the regions separately (Figure 3). Our approach makes it easy to aggregate messages across different groups, on a per-region basis. If a node has two messages to send to a pair of groups G1 and G2 which overlap in region R, then while transmitting to R, the node can batch these messages together.

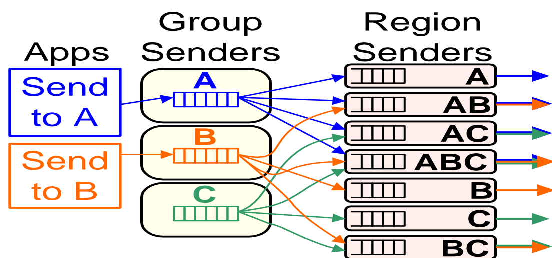

Figure 3: To multicast to a group, QSM sends a copy to each of the underlying regions.

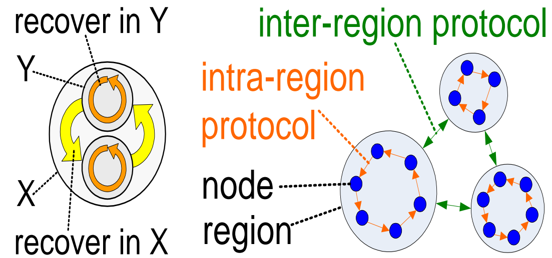

Figure 4: Hierarchical recovery in QSM. A group spans multiple regions. Each region has an associated structure of token rings (right).

To recover from packet loss, QSM uses a hierarchical structure of token rings (we considered using other structures, such as trees, but token rings produce a more predictable traffic pattern; the importance of this will become clear later). The basic structure is illustrated in Figure 4. At the highest level, QSM circulates tokens around sets of regions, aggregating information that can be used by a group sender to retransmit packets that were missed by entire regions (left). Within each region, a token circulates to provide loss recovery at the level of nodes belonging to the region (right).

If regions become large, QSM partitions them into smaller rings. This is illustrated in Figure 5. In the experiments reported in this paper, no token ring ever grows larger than about 25 nodes, and the system uses single and two-level hierarchies. In the future, we plan to experiment with larger configurations and will work with deeper hierarchies.

The QSM recovery protocol uses tokens to track message status (missing/received/cached) at each node. In effect, the token carries ACK and NAK information, aggregated over the nodes "below" each ring. Token rings avoid the kinds of ACK/NAK implosion problems with which reliable multicast protocols traditionally have struggled, but problems of their own: if a message is lost, the sender may not find out for quite a while. In QSM, this isn't a major issue because most message losses can be corrected locally, through cooperation among receivers.

The basic idea is to perform recovery "as locally as possible" (Figure 6). If a message is available within the same token ring, some process that has a

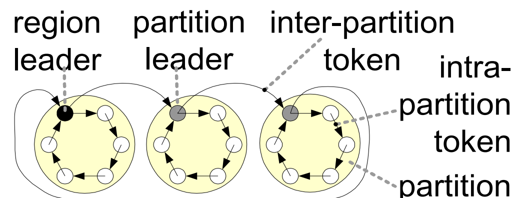

Figure 5. A hierarchy of token rings.

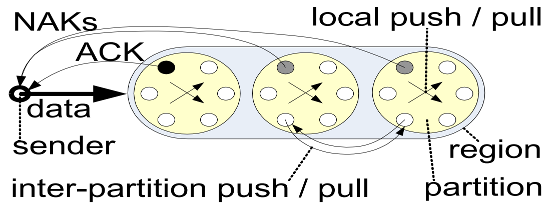

Figure 6. Recovery inside and across partitions.

copy will forward it to the process missing the message. To conserve memory, QSM implements a scheme originally proposed by Zhao [7]: even in a large ring, no more than five nodes cache any given message. QSM also uses this idea at the level of partitions: each message is cached in a single partition, round-robin fashion; if some partition is missing a message, the partition caching it steps in to resend it. Finally, if an entire region is missing a message, the sender becomes involved and re-multicasts it.

QSM tokens also carry other information, including data used to perform rate control and information used to trigger garbage collection.

The overall system configuration is managed by what we call the Configuration Management Service (CMS), which handles join and leave requests, detects node failures, and uses these to generate a sequence of membership views for each multicast. The CMS also determines and continuously updates region boundaries, maintains sequences of region views for each region, and tracks the mapping from group views to region views. In our prototype, the CMS runs on a single node, but we intend to replace this with a state-machine replicated version in the future to eliminate the risk of single-point failures. In the longer term we will move to a hierarchically structured CMS, similar to Moshe [8].

4 Implementation

When we set out to implement QSM, our intent was to leverage the component integration tools available on the Windows platform. We didn't expect that co-existence with the managed environment would require any special architectural features.

QSM is implemented much like any .NET component. The system is coded in C# (about 200,000 lines of code, of which 7500 are unmanaged C++ to interface to the native Windows asynchronous I/O library), and is accessible from any .NET application. Windows understands QSM to be the handler for operations on new kind of event stream. An application can obtain handles from these QSM-managed streams, and can then invoke methods on those handles to send events; incoming messages are delivered through upcalls. QSM is also registered as a "shell extension", making it possible to access the communication subsystem directly from the Windows GUI. For example, the user can store a shortcut to a QSM stream in the file system, and can point-and-click to attach a previewer or a viewer to an event stream.

The overall architecture is summarized in Figure 7. The system is single-threaded and event-driven. We use a Windows I/O completion port, henceforth referred to as an I/O queue, to collect all asynchronous I/O completion events, including notifications of any received messages, completed transmissions, and errors, for all sockets. A single "core thread" synchronously polls the I/O queue to retrieve incoming messages. The core thread also maintains an alarm queue, implemented as a splay tree, for timer-based events, and a request queue, implemented as a lock-free queue with CAS-style operations, for requests from the (possibly multithreaded) application. The core thread polls all queues in a round-robin fashion and processes the events sequentially.

Events of the same type are processed in batches, up to the limit determined by a quantum (currently 50ms for I/O, 5ms for alarms; there is no limit for

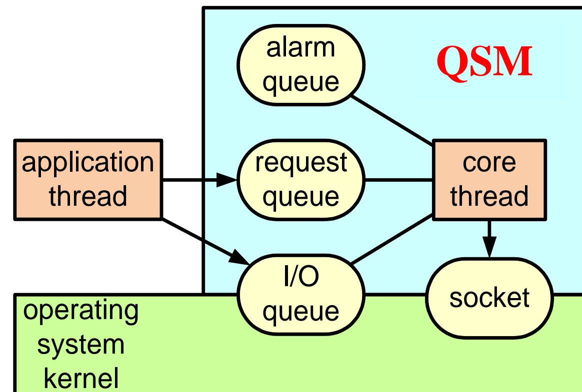

Figure 7. QSM uses a single-threaded architecture, with a "core" thread that controls three queues: for I/O requests, timer-based events, and requests from the possibly multithreaded application.

application requests). When an I/O event representing a received packet is retrieved for a given socket, the socket is drained to minimize the probability of loss.

Several aspects of the architecture are noteworthy because of their performance implications. First, QSM assigns priorities to different types of I/O events. The basic idea is that when an I/O event occurs, we retrieve all events from the I/O queue, determine the type of each, and then place it in an appropriate priority queue. Then, the system processes queued events in priority order (Figure 8). By prioritizing incoming I/O over sending-related events we

reduce packet loss, and by prioritizing control packets over data we reduce delays in reacting to packet loss or other control. In Section 5 we will see that this slashes system-wide memory overheads.

The pros and cons of using threads in event-oriented systems are hotly debated. In QSM itself, threads turned out to be a bad idea. Although we used threads rather casually in the first year of our effort, that version of the system was annoyingly

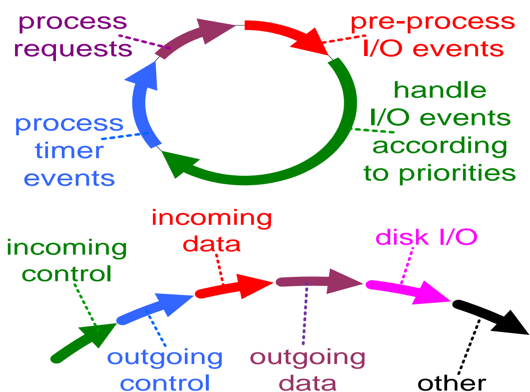

Figure 8. The QSM time-sharing and priority I/O processing policy.

unstable, and prone to oscillatory throughput when scaled up. When we decided to take control over event processing order, we also eliminated multithreading. Fine-grained scheduling eliminated convoy behavior and oscillatory throughput of the sort that can disrupt reliable multicast systems when they run at high data rates on a large scale[2].

The last aspect relates to the creation of new messages, particularly by QSM itself. Readers who have implemented multicast protocols will know that most existing systems are push-based: some layer initiates a new message at will, and lower layers then buffer that message until it can be sent. This makes sense under the assumption that senders often generate bursts of packets; by buffering them, the communication subsystem can smooth the traffic flow and keep the network interface busy. One consequence is that messages can linger for a while before they are sent. Not only does this increase memory consumption, but if a message contains "current state" information, that state may be stale by the time it's sent.

In contrast to this usual approach, QSM implements a "pull" architecture. Our original motivation

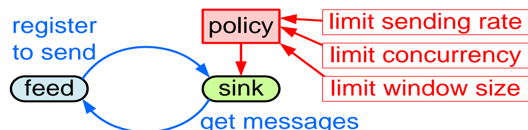

Figure 9. In a "pull" protocol a "feed" registers the intent to send with a "sink" that may be controlled by a policy limiting the send rate, concurrency etc. When the sink is "ready" to send, it issues an upcall.

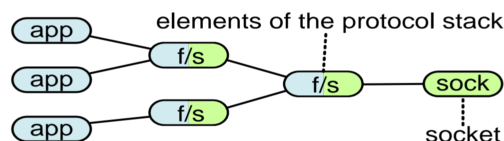

Figure 10. One can think of QSM as a collection of "protocol stacks" in which components act as both feeds and as sinks. The overall structure is of a forest of trees, rooted at sockets.

was to reduce staleness by postponing the creation of control messages until the time when transmission is actually about to take place. "Just-in-time" information is more accurate, and this makes QSM more stable. An unintended benefit is that the "pull" architecture slashes buffering and memory overheads, which, as we shall demonstrate, turns out to have an enormous impact on performance.

In QSM each element of a protocol stack acts as a feed that has data to send, or a sink that can send it (Figure 9), and many play both roles (Figure 10). Rather than creating a message and handing it down to the sink, a feed registers the *intent* to send a message with the sink. The message can be created at this time and buffered in the feed, but the creation may also be postponed until the time when the sink polls the feed for messages to transmit. The sink determines its readiness to send based on a control policy, such as rate, concurrency, windows size limitation, and so forth. When the socket at the root of the tree is ready for transmission, messages will be recursively pulled from the tree of protocol stack components, in a round-robin fashion. Feeds that no longer have data to send are automatically deregistered.

5 Evaluation

Evaluation of QSM could pursue many directions: costs of the domain crossing between the application and QSM, protocol design and scalability, and interactions between protocol properties and the managed framework. Here we focus on the latter. Our goal is to arrive at a deep understanding of the performance limits of QSM when operating at high data rates with large numbers of overlapping groups

[2] For reasons of brevity, we are unable to undertake a detailed analysis of oscillatory phenomena in this paper (also called *convoys* and *broadcast storms*; these plague many multicast and pub-sub products). Event prioritization eliminated such problems in the configurations tested by our experiments.

on varying numbers of nodes. We'll find that the experiments have a pattern: in scenario after scenario, the performance of QSM is ultimately limited by overheads associated with memory management in the managed environment. Basically, the more memory in use, the higher the overheads of the memory management subsystem and the more CPU time it consumes, leaving less time for QSM to run. These aren't just garbage collection costs: every aspect of memory management gets expensive, and the costs grow linearly in the amount of memory in use. When QSM runs flat-out, CPU cycles are a precious commodity. Thus, minimizing the memory footprint turns out to be the key to high performance.

All results reported here come from experiments on a 200-node[3] cluster of Pentium III 1.3GHz blades with 512MB memory, connected into a single broadcast domain using a switched 100Mbps network. Nodes run Windows Server 2003 with the .NET Framework, v2.0. Our benchmark is an nary .NET GUI application, linked to the QSM library, running in the same process. Unless otherwise specified, we send 1000-byte arrays, without preallocating them, at the maximum possible rate, and without batching. The majority of the figures include 95% confidence intervals, but these intervals are sometimes so small that they may not always be visible.

5.1 Memory Overheads on the Sender

We begin by showing that memory overhead at the sender is a central to throughput. Figure 11 shows throughput in messages/s in experiments with 1 or 2 senders multicasting to a varying number of receivers, all of which belong to a single group. With a single sender, no rate limit was used: the sender has more work to do than the receivers and on our clusters, isn't fast enough to saturate the network (Figure 12). With two senders, we report the highest combined send rate that the system could sustain without developing backlogs at the senders.

Why does performance decrease with the number of receivers? First, let's focus on a 1-sender scenario. Figure 12 shows that whereas receivers are not CPU-bound, and loss rates in this experiment (not shown here) are very small, the sender is saturated, and hence is the bottleneck. Running this test again in a profiler reveals that the percentage of time spent in QSM code is decreasing, whereas more and more time is spent in mscorwks.dll, the CLR (Figure 13). More detailed analysis (Figure 14) shows that the main culprit behind the increase of overhead is a growing cost of memory allocation (GCHeap::Alloc) and garbage collection (gc_heap_garbage_collect).

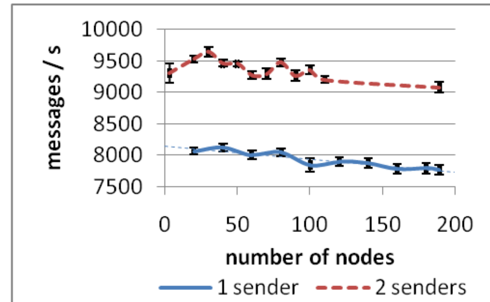

Figure 11. Throughput as a function of the number of nodes (1 group, 1KB messages).

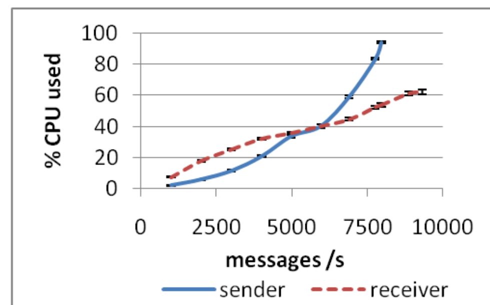

Figure 12. Processor utilization as a function of the multicast rate (100 receivers).

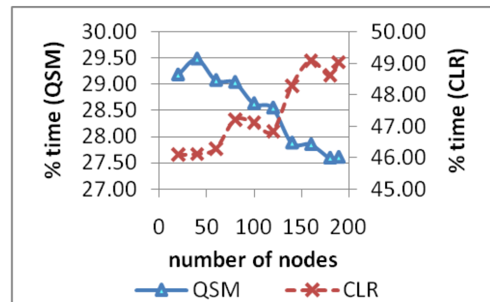

Figure 13. The percentages of the profiler samples taken from QSM and CLR DLLs.

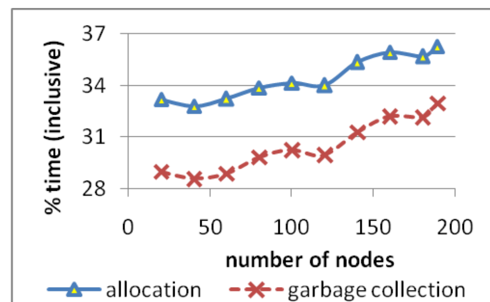

Figure 14. Memory allocation and garbage collection overheads on the

The former grows by 10% and the latter by 15%, as compared to 5% decrease of throughput. The bulk

[3] This configuration is typical of the host environment expected for our target applications.

of the overhead is the allocation of byte arrays to send in the application (“JIT_NewArr1”, Figure 15). Roughly 12-14% of time is spent exclusively on copying memory in the CLR (“memcopy”), even though we used our own scatter-gather serialization scheme that efficiently uses scatter-gather I/O.

The increase in the memory allocation overhead and the activity of the garbage collector are caused by the increasing memory usage. This, in turn, reflectsan increase of the average number of multicasts pending completion (Figure 16). For each, a copy is kept by the sender for possible loss recovery. Notice that memory consumption grows nearly 3 times faster than the number of messages pending acknowledgement. If we freeze the sender process and inspect the contents of the managed heap, we find that the number of objects in memory is more than twice the number of multicasts pending acknowledgement. Although some of these have already been acknowledged, they haven’t yet been garbage collected.

The growing amount of unacknowledged data is caused by the increase of the average time to acknowledge a message (Figure 17). This grows because of the increasing time to circulate a token around the region for purposes of state aggregation (“roundtrip time”). The time to acknowledge is only slightly higher than the expected 0.5s to wait until the next token round, plus the roundtrip time; as we scale up, however, roundtrip time becomes dominant. These experiments show that the critical factor determining performance is the time needed for the system to aggregate state over regions. Moreover, they shed light on a mechanism that links latency to throughput, via increased memory consumption and the resulting increase in allocation and garbage collection overheads.

An 500ms increase in latency, resulting in a 10MB increase in memory consumption, can inflate overheads by 10-15%, and degrade the throughput by 5%. One way to alleviate the problem we've identified could be to reduce the latency of state aggregation, so that it grows sub-linearly. In our system, this might be achieved by using a deeper hierarchy of rings, and by letting tokens in each of these rings circulate independently. This would create a more complex structure, but aggregation latency would grow logarithmically rather than linearly.

Is reducing state aggregation latency the only option? We evaluated two alternative approaches, but found that neither can substitute for lowering the latency of the recovery state aggregation.

Our first approach varies the rate of aggregation by increasing the rate at which tokens are released (Figure 18). This helps only up to a point. Beyond 1.25 tokens/s, more than one aggregation is underway at a time, and successive tokens perform redundant work. Worse, processing all these tokens is costly. Changing the default 1 token/s to 5 tokens/s decreases the amount of unacknowledged data by 30%, but increases throughput by less than 1%.

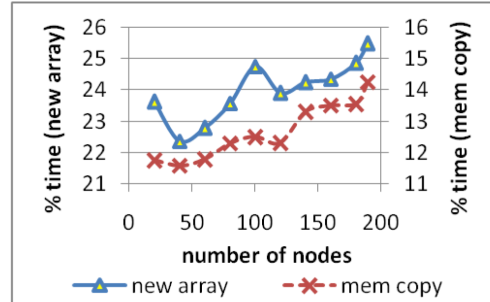

Figure 15. Time spent allocating byte arrays in the application, and copying.

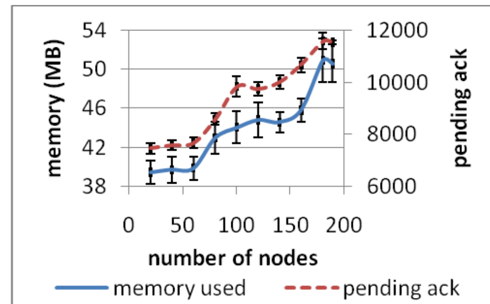

Figure 16. Memory used on sender and the number of multicast requests in progress.

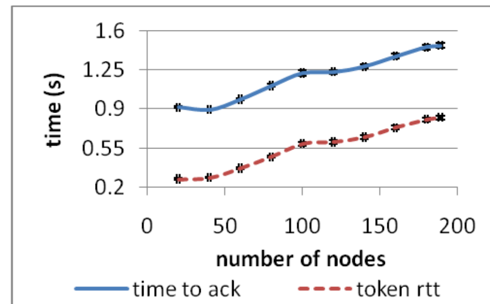

Figure 17. Token roundtrip time and an average time to acknowledge a message.

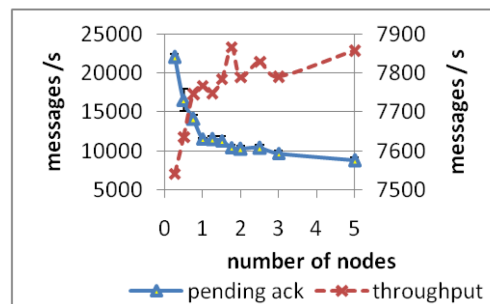

Figure 18. Varying token circulation rate.

Our second approach increased the amount of feedback to the sender. In our base implementation, each aggregate ACK contains a single value MaxContiguous, representing the maximum number such that messages with this and all lower numbers are stable in the region. To increase the amount of feedback, we permit ACK to contain up to k numeric ranges, (a1, b1), (a2, b2), …, (ak, bk). The system can now cleanup message sequences that have as gaps (messages that are still unstable).

In the experiment shown in Figures 19 and 20, we set k to the number of partitions. Unfortunately, while the amount of acknowledged data is reduced by 30%, it still grows, and the overall throughput is actually lower because token processing becomes more costly. Furthermore, the system becomes unstable (notice the large variances in Figure 21Figure), because our flow control scheme, based on limiting the amount of unacknowledged data, breaks down. While the *sender* can cleanup any portion of the message sequence, *receivers* have to deliver in FIFO order. The amount of data they cache is larger, and this reduces their ability to accept incoming traffic.

Notice the linkage to memory. In this case, the growth in memory occurs on the receivers, but the pattern is similar to what we saw earlier: merely having more cached data is enough to slow them down.

5.2 Memory Overheads on the Receiver

The reader may doubt that memory overhead on receivers is the real issue, considering that their CPUs are half-idle (Figure 12). Can increasing memory consumption affect a half-idle node? To find out, we performed an experiment with 1 sender multicasting to 192 receivers, in which we vary the number of receivers that cache a copy of each message ("replication factor" in Figure 22). Increasing this value results in a linear increase of memory usage on receivers. If memory overheads were not a significant issue on half-idle CPUs, we would expect performance to remain unchanged. Instead, we see a dramatic, super-linear increase of the token roundtrip time, a slow increase of the number of messages pending ACK on the sender, and a sharp decrease in throughput (Figure 23).

The underlying mechanism is as follows. The increased activity of the garbage collector and allocation overheads slow the system down and processing of the incoming packets and tokens takes more time. Although the effect is not significant when considering a single node in isolation, a token must visit all nodes in a region to aggregate the recovery state, and delays are cumulative. Normally, QSM is configured so that five nodes in each region cache each packet. If half the nodes in a 192-node region cache each packet, token roundtrip time increases 3-fold. This delays state aggregation, increases pending messages and reduces throughput (Figure 23).

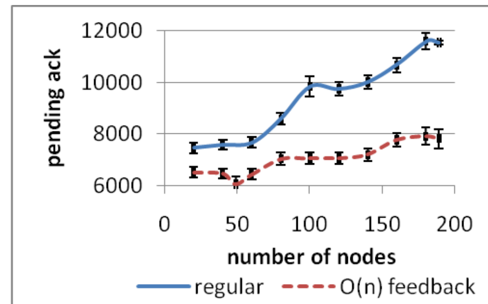

Figure 19. More aggressive cleanup with O(n) feedback in the token and in ACKs.

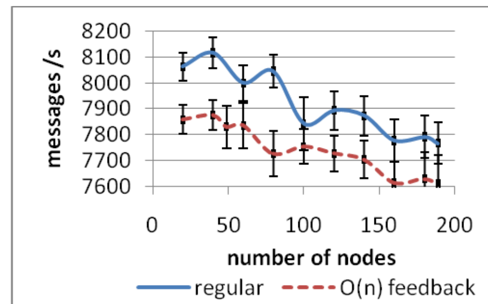

Figure 20. More work with O(n) feedback, and lower rates despite saving on memory.

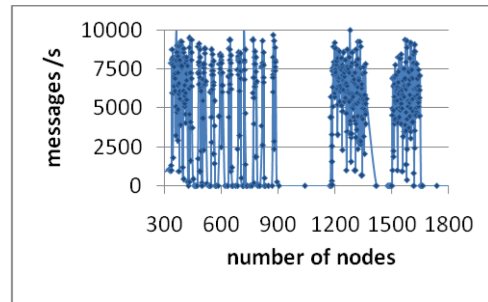

Figure 21. Instability with O(n) feedback.

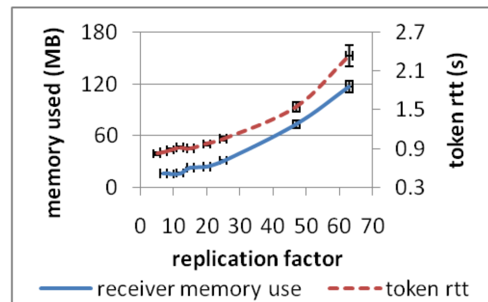

Figure 22. Varying the number of caching replicas per message in a 192-node region.

As the replication factor increasess, the sender's flow control policy kicks in, and the system goes into

a form of the oscillating state we encountered in Figure 21: the amount of memory in use at the sender ceases to be a good predictor of the amount of memory in use at receivers, violating what turns out to be an implicit requirement of our flow-control policy.

5.3 Overheads in a Perturbed System

The reader might wonder whether our results would be different if the system experienced high loss rates or was otherwise perturbed. To find out, we performed an experiment in which one of the receiver nodes experiences a periodic, programmed perturbation: every 5s the node sleeps for 0.5s. This simulates the effect of disruptive, overloaded applications. In the "loss" scenario, every 1s the node drops all incoming packets for 10ms, thus simulating 1% bursty packet loss. In practice, the loss rate is higher, around 2-5%, because recovery traffic interferes with regular multicast, causing further losses.

In both scenarios, CPU utilization at the receivers is in the 50-60% range and doesn't grow with system size, but throughput decreases (Figure 24Figure). In the sleep scenario, the decrease starts at about 80 nodes and proceeds steadily thereafter. It doesn't appear to be correlated to the amount of loss, which oscillates at the level of 2-3% (Figure 25). In the controlled loss scenario, throughput remains fairly constant, until it falls sharply beyond 160 nodes. Here again, performance does not appear to be directly correlated to the observed packet loss. Finally, throughput is uncorrelated with memory use both on the perturbed receiver (Figure 26) or other receivers (not shown). Indeed, at scales of up to 80 nodes, memory usage actually decreases, a consequence of the cooperative caching policy described in Section 3. The shape of the performance curve does, however, correlate closely with the number of unacknowledged requests (Figure 27).

We conclude that the drop in performance in these scenarios can't be explained by correlation with CPU activity, memory, or loss rates at the receivers, but that it does appear correlated to slower cleanup and the resulting memory-related overheads at the sender. The effect is much stronger than in the undisturbed experiments; the number of pending messages starts at a higher level, and grows 6-8 times faster. Token roundtrip time increases 2-fold, and if a failure occurs, it requires 2 token rounds before repair occurs, and then another round before cleanup takes place (Figures 28, 29). Combined, these account for the rapid increase in acknowledgement latency.

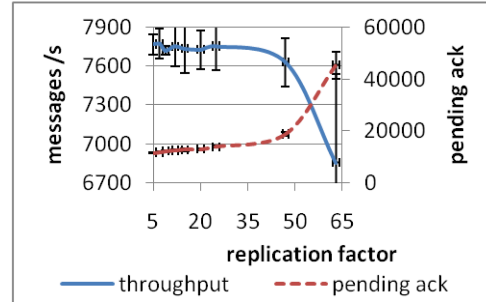

Figure 23. As the number of caching replicas increases, the throughput decreases.

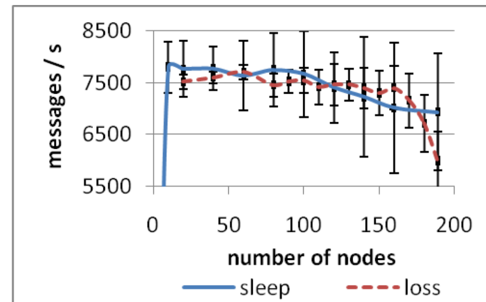

Figure 24. Throughput in the experiments with a perturbed node (1 sender, 1 group).

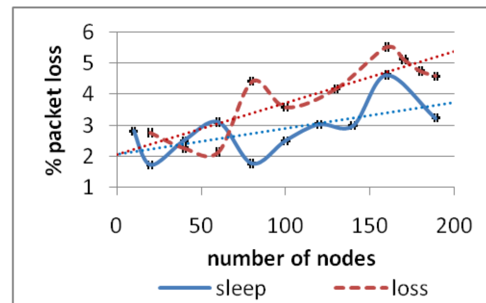

Figure 25. Average packet loss observed at the perturbed node.

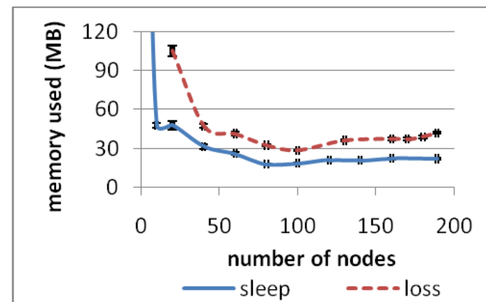

Figure 26. Memory usage at the perturbed node (at unperturbed nodes it is similar).

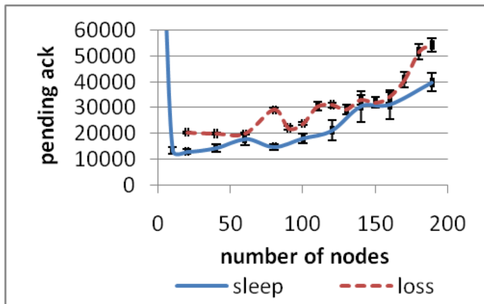

Figure 27. Number of messages awaiting acknowledgement in experiments with perturbances.

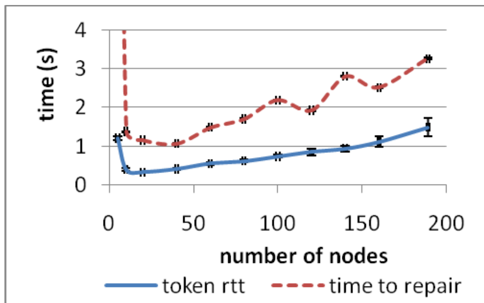

Figure 28. Token roundtrip time and the time to recover in the "sleep" scenario.

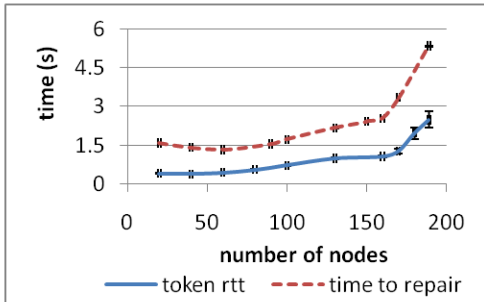

Figure 29. Token roundtrip time and the time to recover in the "loss" scenario.

It is worth noting that the doubled token roundtrip time, as compared to unperturbed experiments, can't be accounted for by the increase in memory overhead or CPU activity on the receivers, as was the case in experiments where we varied the replication factor. The problem can be traced to a priority inversion. Because of repeated losses, the system maintains a high volume of forwarding traffic. The forwarded messages tend to get ahead of the token, both on the send path, where in the sinks, we use a simple round-robin policy of multiplexing between data feeds, and on the receive path, where forwarded packets are treated as control traffic, and while they're prioritized over data, they are treated as equally important as tokens. They also increase the overall volume of I/O that the nodes process. As a result, tokens are processed with higher latency.

Although it would be hard to precisely measure these delays, measuring alarm delays sheds light on the magnitude of the problem. Recall that our time-sharing policy assigns quanta to different types of events. High volumes of I/O, such as caused by the increased forwarding traffic, will cause QSM to use a larger fraction of its I/O quantum to process I/O events, with the consequence that timers will fire late. This effect is magnified each time QSM is preempted by other processes or by its own garbage collector; such delays are typically shorter than the I/O quantum, yet longer than the alarm quantum, thus causing the alarm, but not the I/O quanta, to expire.

The maximum alarm firing delays taken from samples in 1s intervals are indeed much larger in the perturbed experiments, both on the sender and on the receiver side (Figures 30 and 31). Large delays are also more frequent (not shown). The maximum delay measured on receivers in the perturbed runs is 130-140ms, as compared in 12-14ms in the unperturbed experiments. On the sender, the value grows from 700ms to 1.3s. In all scenarios, the problem could be alleviated by making our priority scheduling more fine-grained, e.g. varying priorities for control packets, or by assigning priorities to feeds in the sending stack.

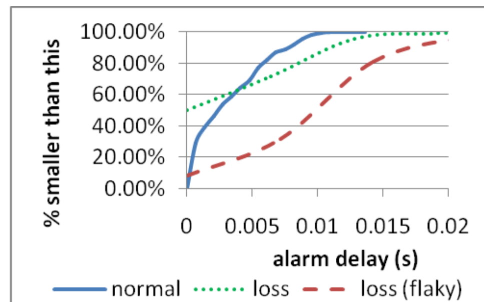

Figure 30. Histogram of maximum alarm delays in 1s intervals, on the receivers.

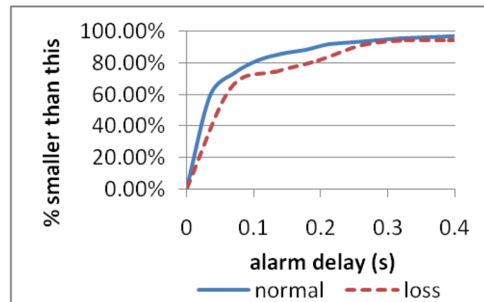

Figure 31. Histogram of maximum alarm delays in 1s intervals, on the sender.

5.4 Overheads in a Lightly-Loaded System

So far the evaluation has focused on scenarios where the system was heavily loaded, with unbounded multicast rates and occasional perturbations. In each case, we traced degraded performance or scheduling delays to memory-related overheads. But how does the system behave when lightly loaded? Do similar phenomena occur? Here we'll see that load has a super-linear impact on performance. In a nutshell, the growth in memory consumption causes slowdowns that amplify the increased latencies associated with the growth in traffic.

To show this we designed experiments that vary the multicast rate. Figure 12 showed that the load on receivers grows roughly linearly, as expected given the linearly increasing load, negligible loss rates and the nearly flat curve of memory consumption (Figure 33), the latter reflecting our cooperative caching policy. Load on the sender, however, grows super-linearly, because the linear growth of traffic, combined with our fixed rate of state aggregation, increases the amount of unacknowledged data (Figure 32), increasing memory usage. This triggers higher overheads: for example, the time spent in the garbage collector grows from 50% to 60% (not shown here). Combined with a linear growth of CPU usage due to the increasing volume of traffic, these overheads cause the super-linear growth of CPU overhead shown on Figure 12.

The increasing number of unacknowledged requests and the resulting overheads rise sharply at the highest rates because of the increasing token roundtrip time. The issue here is that the amount of I/O to be processed increases, much as in some of the earlier scenarios. This delays tokens as a function of the growing volume of multicast traffic. We confirm the hypothesis by looking at the end-to-end latency (Figure 34). Generally, we would expect latency to decrease as the sending rate increases because the system operates more smoothly, avoiding context switching overheads and the extra latencies caused by the small amount of buffering in our protocol stack.

With larger packets once the rate exceeds 6000 packets/s, the latency starts increasing again, due to the longer pipeline at the receive side and other phenomena just mentioned. This is not the case for small packets (also in Figure 34); here the load on the system is much smaller. Finally, the above observations are consistent with the sharp rise of the average delay for timer events (Figure 35). As the rate changes from 7000 to 8000, timer delays at the receiver increase from 1.5ms to 3ms, and on the sender, from 7ms to 45ms.

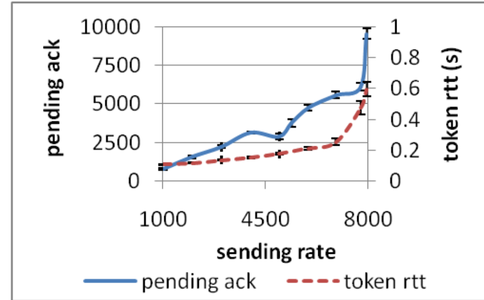

Figure 32. Number of unacknowledged messages and average token roundtrip time as a function of the sending rate.

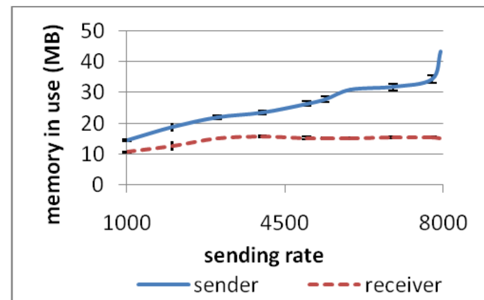

Figure 33. Linearly growing memory use on sender and the nearly flat usage on the receiver as a function of the sending rate.

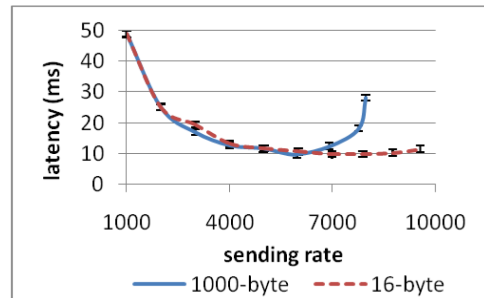

Figure 34. The send-to-receive latency for varying rate, with various message sizes.

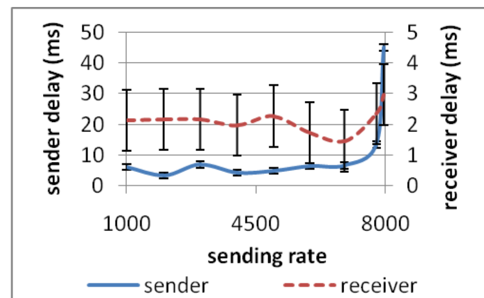

Figure 35. Alarm firing delays on sender and receiver as a function of sending rate.

5.5 Per-Group Memory Consumption

In a final set of experiments, we focus on scalability with the number of groups. A single sender multicasts to a varying number of groups in a round-robin fashion. All receivers join all groups, and since the groups are perfectly overlapped, the system contains a single region. QSM's regional recovery protocol is oblivious to the groups, hence the receivers behave identically no matter how many groups we use. On the other hand, the sender maintains a number of per-group data structures. This affects the sender's memory footprint, so changes to throughput or protocol behavior must be directly or indirectly linked to memory usage.

We do not expect the token roundtrip time or the amount of messages pending acknowledgement to vary with the number of groups, and until about 3500 groups this is the case (Figure 36). However, in this range memory consumption on the sender grows (Figure 37), and so does the time spent in the CLR (Figure 38), hurting throughput (Figure 39). Inspection of the managed heap in a debugger shows that the growth in memory used is caused not by messages, but by the per-group elements of the protocol stack. Each maintains a queue, dictionaries, strings, small structures for profiling etc. With thousands of groups, these add up to tens of megabytes.

We can confirm the theory by turning on additional tracing in the per-group components. This tracing is lightweight and has little effect on CPU consumption, but it increases the memory footprint by adding additional data structures that are updated once per second, which burdens the GC. As expected, throughput decreases (Figure 39, "heavyweight").

It is worth noting that the memory usages reported here are averages. Throughout the experiment, memory usage oscillates, and the peak values are typically 50-100% higher. The nodes on our cluster only have 512MB memory, hence a 100MB average (200MB peak) memory footprint is significant. With 8192 groups, the peak footprint approaches 360MB, and the system is close to swapping.

Even 3500-4000 groups are enough to trigger signs of instability. Token roundtrip times start to grow, thus delaying message cleanup (Figure 40) and increasing memory overhead (Figure 41). Although the process is fairly unpredictable (we see spikes and anomalies), we can easily recognize a super-linear trend starting at around 6000 groups. At around this point, we also start to see occasional bursts of packet losses (not shown), often roughly correlated across receivers. Such events trigger bursty recovery overloads, exacerbating the problem.

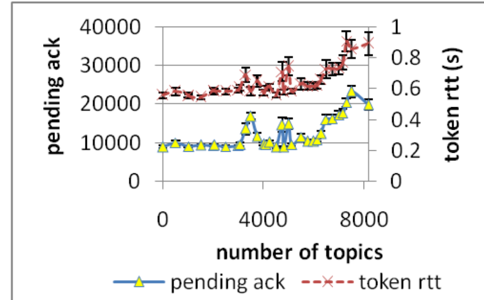

Figure 36. Number of messages pending ACK and token roundtrip time as a function of the number of groups.

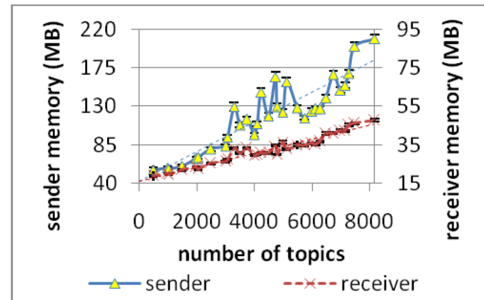

Figure 37. Memory usage grows with the number of groups. Beyond a certain threshold, the system is increasingly unstable.

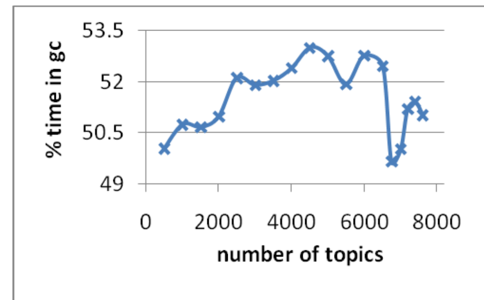

Figure 38. Time spent in the CLR code.

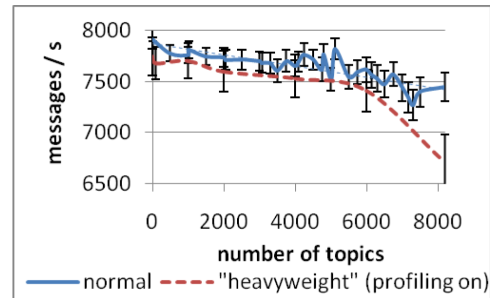

Figure 39. Throughput decreases with the number of groups (1 sender, 110 receivers, all groups have the same subscribers).

Stepping back, the key insight is that all these effects originate at the sender node, which is more loaded and less responsive. In fact, detailed analysis of the captured network traffic shows that the multicast stream in all cases looks basically identical, and hence we cannot attribute token latency or losses to the increased volume of traffic, throughput spikes or longer bursts of data. With more groups, the sender spends more time transmitting at lower rates, but doesn't produce any faster data bursts than those we observe with smaller numbers of groups (Figure 40). Receiver performance indicators such as delays in firing timer event or CPU utilization don't show any noticeable trend. Thus, all roads lead back to the sender, and the main thing "going on" in the sender is that it has a steadily growing memory footprint.

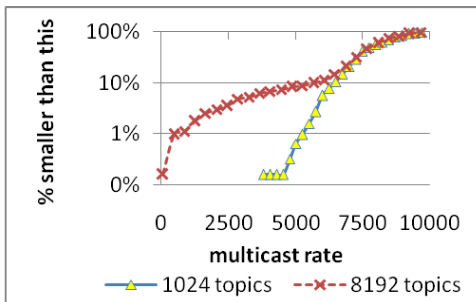

Figure 40. Cumulative distribution of the multicast rates for 1K and 8K groups.

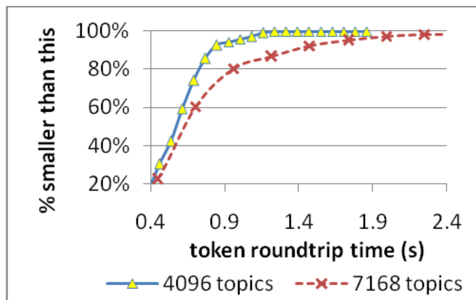

Figure 41. Token roundtrip times for 4K and 7K groups (cumulative distribution).

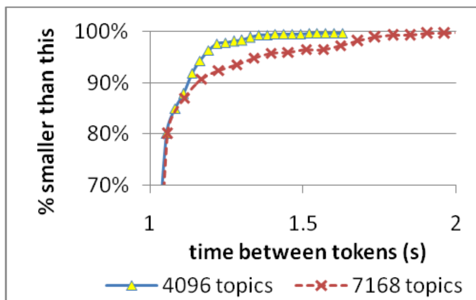

Figure 42. Intervals between the subsequent tokens (cumulative distribution).

We also looked at token round-trip times. The distribution of token roundtrip times for different numbers of groups shows an increase of the token roundtrip time, caused almost entirely by 50% of the tokens that are delayed the most (Figure 41), which points to disruptive events as the culprit, rather than a uniform increase of the token processing overhead. And, not surprisingly, we find that these tokens were most commonly delayed on the sender.

With many thousands of groups, the average time to travel by one hop from sender to receiver or receiver to sender can grow to nearly 50-90ms, as compared to an average 2ms per hop from receiver to receiver (not shown). Also, the overloaded sender occasionally releases the tokens with a delay, thus introducing irregularity. For 10% of the most-delayed tokens, the value of the delay grows with the number of groups (Figure 42). Our old culprit is back: memory-related costs at the sender! To summarize, increasing the number of groups slows the sender, and this cascades to create all sorts of downstream problems that can destabilize the system as a whole.

6 Discussion

The experiments just reported make it clear that the performance-limiting factor in the QSM system is latency, and that in addition to protocol factors such as the length of token rings, latency is strongly influenced by the memory footprint of the system. Of course, when we built the system it was obvious that minimizing latency would be important; this motivated several of the design decisions discussed in Section 3. But the repeated linkage of latency and oscillatory throughputs to memory was a surprise: we expected a much smaller impact. We can summarize our design insights as follows:

1. Minimize the memory footprint. We expected that the primary cost of managed memory would be associated with garbage collection. Instead, *all* costs associated with managed memory rise in the amount of allocated memory, at least in the Windows CLR. Implications include:

1.1 Pull data. Whereas traditional multicast systems accept messages whenever the application layer or the multicast protocols produce it, QSM uses an upcall-driven pull architecture. Often we can delay generating a message until the last minute, and we can also avoid situations in which data piles up on behalf of an aggressive sender.

1.2 Limit buffering and caching. Most existing multicast protocols buffer data at many layers and cache data rather casually for recovery purposes. This turns out to be extremely costly in a managed setting and must be avoided whenever possible.

1.3 Clear messages out of the system quickly. Data paths should have rapid data movement as a key goal.
2. Minimize delays. We've already mentioned that data paths should clear messages quickly, but there are other important forms of delay, too. Most situations in which QSM developed convoy-like behavior or oscillatory throughput can be traced to design decisions that caused scheduling jitter or allowed some form of priority inversion to occur, delaying a crucial message behind a less important one. Implications included the following:
2.1 Event handlers should be short, predictable and terminating. In building QSM, we struggled to make the overall behavior of the system as predictable as possible – not a trivial task in configurations where hundreds of processes might be multicasting in thousands of overlapping groups. By keeping event handlers short and predictable and eliminating the need for locking, we obtained a more predictable system and were able to eliminate multithreading, with the associated context switching and locking overheads.
2.2 Drain input queues. Here we encounter a tension between two goals. From a memory footprint perspective, one might prefer not to pull in a message until QSM can process it. But in a datacenter or cluster, most message loss occurs in the operating system, not on the network, hence message loss rates soar if we leave messages on input sockets for long.
2.3 Control the event processing order. In QSM, this involved single-threading, batched asynchronous I/O, and the imposition of an internal event processing prioritization. Small delays add up in large systems: tight control over event processing largely eliminated convoy effects and oscillatory throughput problems.
2.4 Act on Fresh State. Many inefficiencies can be traced to situations in which one node takes action on the basis of stale state information from some other node, triggering redundant retransmissions or other overheads. The pull architecture has the secondary benefit of letting us delay the preparation of status packets until they are about to be transmitted.

7 Conclusions

The premise of our work is that developers of services intended to run on clustered platforms desire the productivity and robustness benefits of managed environments, and need replication tools integrated with those environments. Building such tools so posed challenges to us as protocol and system designers, which were the primary focus of our paper. A central insight is that high-performance protocols running in managed settings need to maintain the smallest possible memory footprint. By repeated application of this principle, QSM achieves scalability and stability even at very high loads.

An unexpected side effect of building QSM in Windows was that by integrating our system tightly with the platform, we created a new kind of live distributed objects: abstract data types that form groups, share state, and that are updated using QSM multicasts. These look natural to the Windows user: such an object changes faster than the average Windows object, but the same basic mechanisms can support them, and the component integration environment (type checking, debugging, etc) extends seamlessly to encompass them. Although a great deal of additional work is needed, QSM should eventually enable casual use of live objects not just in datacenters but also on desktops in WAN settings, opening the door to a new style of distributed programming.

The current version of QSM is stable in cluster settings and, as noted earlier, has a growing community of users. Looking to the future, we plan to scale QSM into WAN settings, to support a wider range of multicast reliability properties, and to introduce a gossip infrastructure that would support configuration discovery and other self-* mechanisms. Live objects pose a protocol design challenge: they give rise to irregular patterns of overlapping multicast groups; hence our region-oriented state aggregation mechanisms will need to be redesigned. We have an idea for solving this (basically, recovery would be performed by selecting a subset of nodes that form a clean overlay structure, rather than just treating every single receiver as a member of a recovery region). Whether this can really scale remains to be seen.

8 References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, J. Stanton. The Spread Toolkit: Architecture and Performance. 2004.
[2] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. (1998).
[3] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3):332-383, Aug 2001.
[4] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. Distributed Systems Engineering. Mar 1994. 1:29-36.
[5] S. Maffeis, D. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine feature topic issue on Distributed Object Computing, Vol. 14, No. 2, February 1997.

[6] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System. PDCS, 2005.
[7] Zhen Xiao, Robbert van Renesse, Kenneth Birman. Optimizing Buffer Management for Reliable Multicast. Proceedings of the International Conference on Dependable Systems and Networks (DSN '02), June 2002.
[8] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. ACM Transactions on Computer Systems, Vol. 20, No. 3, August 2002, p. 191-238.