Quicksilver Scalable Multicast (QSM)

Krzysztof Ostrowski[†], Ken Birman[†], and Danny Dolev[§]
[†]*Cornell University and* [§]*The Hebrew University of Jerusalem*
{krzys,ken}@cs.cornell.edu, dolev@cs.huji.ac.il

Abstract

QSM is a multicast engine designed to support a style of distributed programming in which application objects are replicated among clients and updated via multicast. The model requires platforms that scale in dimensions previously unexplored; in particular, to large numbers of multicast groups. Prior systems weren't optimized for such scenarios and can't take advantage of regular group overlap patterns, a key feature of our application domain. Furthermore, little is known about performance and scalability of such systems in modern managed environments. We shed light on these issues and offer architectural insights based on our experience building QSM.

1. Introduction

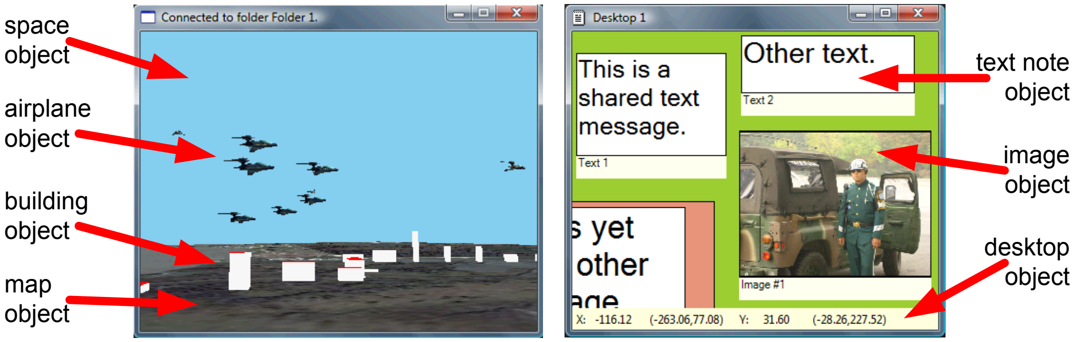
Web applications are becoming increasingly dynamic: users expect live content powered by real time data that can be modified, and demand a high-quality multimedia experience. To a degree these are opposing goals: peer-to-peer streaming systems are optimized to move data one-way; interactive client-server systems don't scale as well. Reliable multicast is a useful third option: instead of storing content on servers, we replicate it across clients and multicast updates between them in a peer-to-peer manner. Not every application can be built this way, but the model is broadly applicable; in particular, it is a good fit for interactive "mashups" of the sort shown on Figure 1. Each application object, such as a shared desktop or a text on the desktop, has its data replicated among all of its clients (the processes displaying it), which form a multicast group.

In [22], we described a general programming model based on this approach. Here, we focus on how to implement such systems. The key element of the platform is a multicast engine: most objects either have replicated state, or are driven by a stream of updates. Reliability is important, but weak guarantees are often sufficient: one rarely needs atomic transactions or consensus. Total ordering can be avoided: often updates originate at one or a few sources, and portions of objects can be locked prior to updating via separate locking facilities. More importantly, one needs high streaming bandwidth, low CPU footprint, and the ability to support large numbers of users, and large numbers of multicast instances. The last factor is important: each object could be accessed by a different group of clients, and so it needs its "own" separate instance of a multicast protocol.

Multicast groups corresponding to different objects often overlap in regular ways. To see this, consider airplane A, on a spatial desktop X of the sort shown on Figure 1. When the user opens X, the user can also see A. The group of X's clients is thus a subset of A's clients. Overlap occurs also if objects related to common topics are used by people with common interest. Furthermore, in [28] we show that even if objects are used at random, we can often partition them into a small number of subsets such that groups in each set overlap hierarchically. Regularities in group overlap occur naturally in our application domain. As we shall demonstrate, they could be leveraged to amortize overhead across protocol instances.

In the work reported here, we focus on enterprise computing environments: centers with thousands of commodity PCs running Microsoft Windows and communicating on a shared high-speed LAN. While our broader vision can be applied in WAN scenarios, the path to broad adoption of the paradigm leads through successful use in corporate settings and on campus networks, in context of applications such as collaborative editing, interactive gaming, online courses and videoconferencing, or distributed event processing. Accordingly, we assume an environment with system-wide support for IP multicast, in which packet loss is relatively uncommon and bursty (mostly triggered by overloaded recipients).

QSM is a system designed with precisely the objectives articulated above: it offers a simple, but useful reliability guarantee despite bursty loss. It streams at close to network speeds, at rates up to 9500 packets/s in a cluster of 1.3GHz/512MB nodes on a 100Mbps



This is a shared text message.

Other text.

Figure 1. Example mashups enabled by our live objects platform and targeted by QSM. Each object shown here is replicated among its clients and backed by a multicast protocol. Other examples and videos of the platform in use can be found at http://liveobjects.cs.cornell.edu.

LAN. It has a low CPU footprint: 50% CPU usage on receivers at the highest data rates. Throughput falls by 5% as group size increases to 200 nodes and degrades gracefully with loss. QSM exploits regularities in group overlap, and amortizes protocol overheads across sets of nodes with a similar group membership.

In summary, this paper makes the following contributions:

- We propose a novel approach to scaling reliable multicast with the number of groups by leveraging regularity in the group overlap patterns to amortize overhead across protocol instances.
- We discover a previously unnoticed connection between memory usage and local scheduling policy in managed runtime environment (.NET), and multicast performance in a large system.
- We describe several techniques, such as priority I/O scheduling or pull protocol stack architecture, developed based upon these observations, which increase performance via reducing instabilities causing broadcast storms, unnecessary recovery and other disruptive phenomena.

2. Prior Work

Reliable multicast is a mature area [13] [16] [19] [23] [24], but existing systems were not optimized for our scenario. Most systems were designed for use with one group at a time. Some don't support multiple groups at all, others run separate protocol instances per group and incur overhead linear in the number of groups. Popular toolkits such as JGroups [2] perform best at small scale [3] and aren't optimized for network speeds. Systems that use IP multicast and run a protocol per group also suffer from *state explosion*: large numbers of IP multicast addresses in use require hardware to maintain a lot of state; this caused many data centers to abandon IP multicast based products. Also, the ability of NICs at client nodes to filter IP multicast is limited. With 1000s of addresses in active use local CPU is involved even on nodes that didn't subscribe to any of these addresses.

Systems like Isis and Spread can support lots of groups [1] [12] [26], but these groups are an application-level illusion; physically, there is just one group. In Isis, it consists of the members of all application groups. Spread uses a smaller set of servers to which client systems connect: each application-level message is sent to a server, multicast among servers and filtered depending on whether a server has clients in the destination group or not, and finally, relayed by servers to their clients. Support for large numbers of groups in such systems comes at a high cost: unwanted traffic and filtering in software, extra hops and higher latency, and bottlenecks created by the servers.

Application-level multicast systems such as OverCast, NARADA, NICE or SplitStream have been proposed [4] [6] [8] [14] that do not use IP multicast. These are very scalable, but messages follow circuitous routes, incurring high latency. In enterprise LANs, where IP multicast is available, such solutions simply don't fully utilize existing hardware support. Moreover, nodes are asked to forward messages that don't interest them at very high rates. This imposes additional overheads.

SpiderCast [7] and the techniques proposed in [27] have addressed the issue of scaling with the number of groups, but only in the context of unreliable multicast. Their overall approaches differ significantly from ours.

3. Protocol

QSM's approach to scalability is based on the concept of a *region*. A region is a set of nodes that are members of the same multicast groups. The system is partitioned into regions (Figure 2) by a *global*

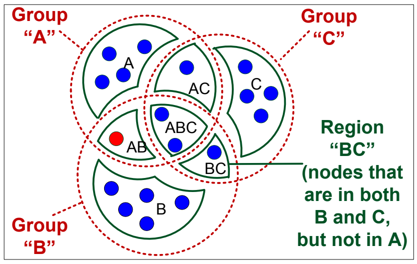


Figure 2. Groups overlap to form regions. Nodes belong to the same region if they joined the same groups.

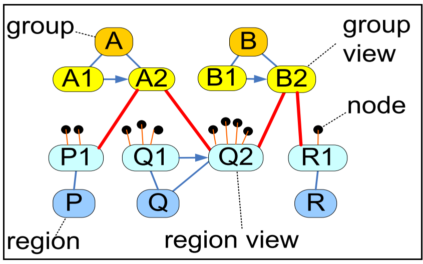


Figure 3. GMS keeps a mapping from groups to regions. Nodes use it to reliably construct distributed structures.

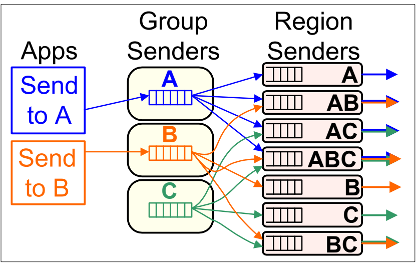


Figure 4. To multicast to a group, QSM sends the message to each of the regions that the group spans over.

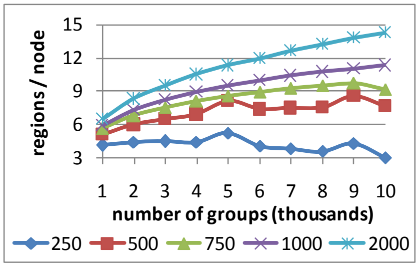


Figure 5. A set of irregularly overlapped groups is partitioned into a small number of regularly overlapped subsets.

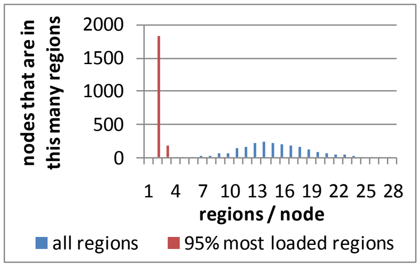


Figure 6. Most of the traffic a node sees is concentrated in just 2 of the regions of overlap to which it was assigned.

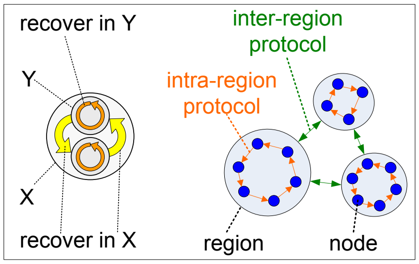


Figure 7. Recovery is done in a hierarchical manner, first locally, and then globally, via a hierarchy of token rings.

membership service (GMS [15]). Nodes contact the GMS to join groups, and it monitors their health. As the system changes, the GMS maintains a sequence of group and region *views* (sets of nodes that were members of given groups and regions at given points in time) and a mapping from one to the other (Figure 3). Each group view in this mapping is mapped to all region views that contain members of this group. The relevant parts of it are distributed to nodes, and are used to construct distributed structures, such as token rings, in a consistent manner. The GMS also assigns an individual IP address to each region. Multicasting to a group is then done by transmitting the message to each region the group spans over, using a single per-region IP multicast (Figure 4).

This scheme is less bandwidth-efficient than multicasting to a per-group IP multicast address, but it avoids the address explosion problem mentioned earlier: there are fewer regions than groups, and each node only needs to subscribe to a single IP multicast address at a time. The technique is most efficient when overlap is regular enough, so that regions consist of at least a few nodes, and each group maps to at most a few regions. As mentioned earlier, by using a technique described in [28], this can be achieved even in scenarios with irregular overlap, by partitioning groups into subsets. In a nutshell, we start by choosing the largest group, and then look for another group that is either contained in it, or overlaps with it in a way that doesn't produce small regions. We keep adding groups to the set as long as maintaining regular overlap is possible, and then simply start a new set of groups: pick the largest, and proceed as before. In the end, this greedy scheme yields a partitioning of groups such that in each partition, groups overlap regularly. The GMS can then simply run multiple instances of itself, each instance maintaining a mapping of the sort described before. Each node may now be a member of a few regions: at most one region for each subset of groups.

Only practice can tell how well this scheme works in real usage scenarios, but simulation results are promising. Figure 5 shows an experiment, in which a varying number of nodes (250 to 2000) subscribed to some 10% of a varying number of groups (1000 to 10000), using Zipf popularity with parameter α=1.5. Several studies suggest that this scenario is realistic [11] [17] [25]. After partitioning, an average node belongs to between 4 and 14 regions. Additionally, 95% of the packets a node will receive is concentrated in only 2 of these (Figure 6; here 2000 processes each join some

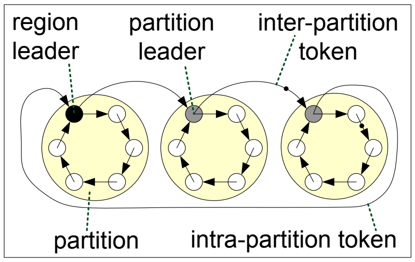


Figure 8. Token rings at the higher levels in the hierarchy are run by designated partition and regions leaders.

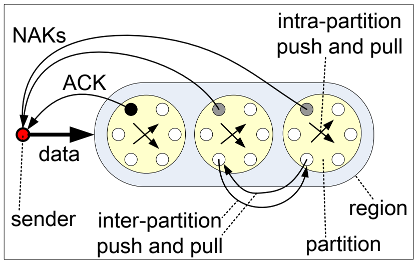


Figure 9. Per-partition rings enable recovery within partitions. Regional rings enable recovery across partitions.

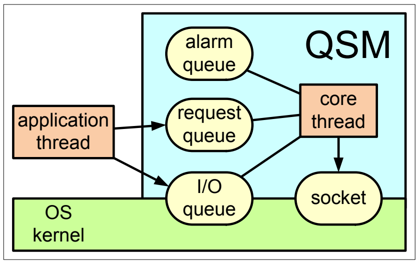


Figure 10. A single thread in QSM processes I/O, timer and application events based on its internal scheduling policy.

10% of a set of 10000 groups). In these experiments, no group is ever fragmented into more than ~5 regions. Regions generally contained 5-10 members, and the most heavily loaded ones often reflected the intersection of ~10 or more groups. This result suggests that in our design and analysis we can henceforth focus on just a single subset of regularly overlapping groups.

Recovery in QSM is hierarchical; the basic idea is to recover as locally as possible (Figure 7). To achieve this, groups are subdivided into smaller and smaller entities. First, a group is divided into regions it spans across. Each region is then subdivided into *partitions* of a fixed size. Each partition runs a local recovery protocol to ensure that its members received the same messages. Each region runs a higher-level protocol to ensure that all of its partitions received the same messages. Finally, a protocol run at an inter-regional level ensures that if the entire region lost a message, it is recovered from the source.

At each level, recovery is performed by a token ring (Figure 8, Figure 9). At the lowest level the ring is run by nodes in a partition. The token is carrying ACKs and NAKs, which neighbors on the ring compare, and use to initiate push or pull recovery from each other. The token is also used to calculate collective ACK and NAK data, describing the status of the entire partition (messages lost or received by all). The aggregate is intercepted by a designated *partition leader*, which participates in a higher-level ring. Again, neighbors compare ACKs and NAKs collected from partitions they represent, and use these to initiate recovery across partitions and calculate aggregate ACKs/NAKs for the entire region. These are collected by the *region leader*.

In QSM, this recursive scheme is only 3 levels deep, but it could be generalized [20] [21]. The overhead is extremely low: in our experiments, tokens circulate only 1 to 5 times/s. This is a key factor enabling high performance. Despite the low overhead, recovery in QSM is efficient, in part thanks to QSM's cooperative caching similar to [5], and massively parallel recovery. In each region only one partition, selected in a round-robin fashion, keeps a copy of each message. When a long burst of messages is lost by a node, often every partition is involved in recovery, and since nodes to recover from are picked at random among those in a partition, often the entire region is helping to repair the loss. The efficiency of this technique is especially high in very large regions.

The key to QSM's scalability edge is the observation that, since nodes in the region are all members of the same groups, they receive exactly the same messages. Thanks to this, QSM can run a *single* token ring in each region and partition, and use it to perform recovery simultaneously for *all* groups.

The benefits are two-fold. First, each node in a region has at most four neighbors, and receives a fixed small number of control packets/s. This is in contrast to systems that run a separate protocol per group, where nodes may have unbounded in-degrees and experience arbitrary rates of control traffic. Furthermore, recovery overhead in each region depends *only* on the number of currently active senders, *not* on the number of groups or the total number of senders in the system. This is because when sending to a region, senders can index messages on a per-region basis, across groups.

The token carries a separate recovery record on a per-sender basis. When a sender ceases to actively multicast, in 2-3 token rounds QSM removes its record from the token, to reintroduce it again next time a message from this sender is seen in the region.

4. Architecture

We implemented QSM as a .NET library (mostly in C#). In the course of doing so, we found landmark features of managed environments, such as garbage collection and multithreading, to have surprisingly strong performance implications.

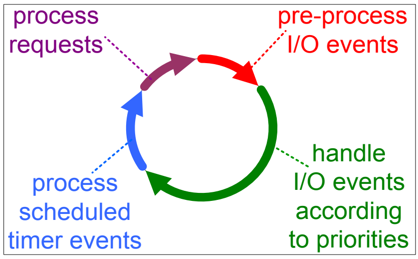

process requests
pre-process I/O events
handle I/O events according to priorities
process scheduled timer events

Figure 11. QSM uses custom time-sharing scheme, with a quantum per event type. I/O is handled like interrupts.

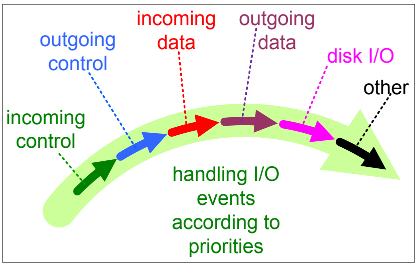


Figure 12. QSM prioritizes I/O events: control packets and inbound I/O are handled ahead of everything else.

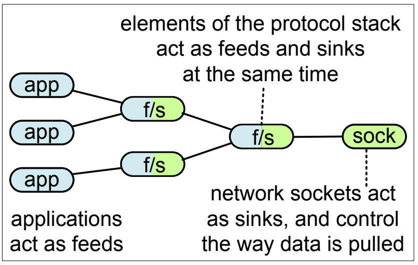

elements of the protocol stack act as feeds and sinks at the same time
app
app
app
f/s
f/s
f/s
sock
applications act as feeds
network sockets act as sinks, and control the way data is pulled

Figure 13. Elements of QSM protocol stack form trees rooted at sockets. Sockets pull data from "their" trees.

QSM is single-threaded and event-driven: a dedicated core thread processes events from three queues. An *I/O queue*, based on a Windows I/O completion port, collects asynchronous I/O completion notifications for all network sockets or files used by QSM. An internal *alarm queue* based on a splay tree stores timer events. Finally, a lock-free *request queue* implemented on CAS-style operations allows the core thread to interact with other threads (Figure 10). The core thread polls its queues in a round-robin fashion and processes events of the same type in batches (Figure 11), up to the limit determined by its quantum (50ms for I/O, 5ms for timer events; no limit for application requests), except that if an incoming packet is found on a socket, the socket is drained of I/O to reduce the risk of packet loss. For I/O events, QSM further prioritizes their processing, in a manner reminiscent of interrupt handling. First, events are read off the I/O queue, and scattered across 6 priority queues. Then, they are handled in priority order. Inbound I/O is prioritized over outbound I/O to reduce packet loss and avoid contention. Control or recovery packets are prioritized over regular multicast, to reduce delays in reacting to packet loss (Figure 12). The pros and cons of using threads in event-oriented systems are hotly debated. In our case, multithreading was not only a source of overhead due to context switches, but more importantly, a cause of instabilities, oscillatory behaviors, and priority inversions due to the random processing order. Eliminating threads and introducing custom scheduling let us take control of this order, which greatly improved performance. In Section 5 we will see that the latency of control traffic is the key to minimizing memory overheads, and as a result, it has a serious impact on the overall system performance.

Control latencies and memory overheads motivate another design feature: a pull protocol stack architecture (Figure 13). QSM avoids buffering data, control, or recovery messages, and delays their creation until the moment they're about to be transmitted. The protocol stack is organized into a set of trees rooted at individual sockets, and consisting of *feeds* that can produce data and *sinks* that can accept it. Feeds register with sinks. Sinks pull data from registered feeds according to their local rate, concurrency, windows size, or other control policy. Using this scheme yields two advantages. First, bulky data doesn't linger in memory and stress a garbage collector. Second, information created just in time for transmission is fresher. ACKs and NAKs become stale rather quickly: if sent after a delay, they often trigger unnecessary recovery or fail to report that data was received. Likewise, recovery packets created upon the receipt of a NAK and stored in buffers are often redundant after a short period: meanwhile, the data may be recovered via other channels. Postponing their creation prevents QSM from doing useless work.

5. Evaluation

In our evaluation of QSM, we focus on scalability and on the interactions of the protocol with the runtime environment that have driven our architectural decisions. The experiments we report reveal a pattern: in each scenario, performance is limited by overheads related to memory management in .NET, which grow linearly with the amount of memory in use, causing the .NET CLR to steal CPU cycles from QSM. Managing the use of memory within QSM turned out to be the key to achieving stable, high performance.

In Section 5.1 and Section 5.2 we show that memory overhead at senders and receivers is linked to latency, and that latency is affected by the overhead it causes. In Sections 5.3 and Section 5.4, we show that this is also true also if the system is perturbed or if it is not saturated. In Section 5.5 we show how the number of groups can cause such effects.

Our results are abbreviated for lack of space (details can be found in our technical report). All results were obtained on a 200-node Pentium III 1.3 GHz, 512

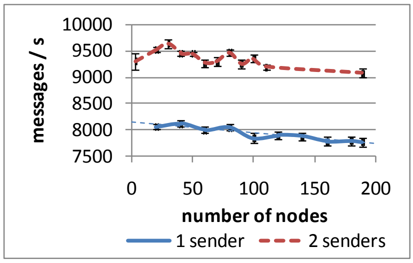


Figure 14. Max throughput in messages/s as a function of the number of receivers with 1 group and 1KB messages.

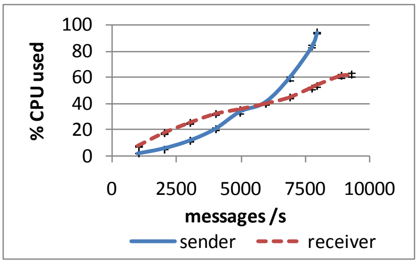


Figure 15. % CPU utilized as a function of multicast rate (single group, 100 receivers).

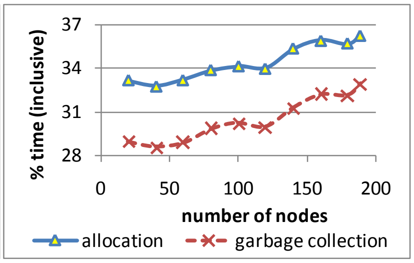


Figure 16. % of time spent on memory-related tasks on the sender: allocation and garbage collection in CLR code.

MB cluster, on a 100 Mbps LAN, running Windows Server 2003, .NET 2.0. Our benchmark is an ordinary .NET executable, using QSM as a library. We send 1000-byte arrays without pre-allocating them, with no batching, at the maximum rate. Our 95% confidence intervals were not always large enough to be visible.

5.1 Memory Overheads on the Sender

Figure 14 shows throughput in messages/s as a function of the number of receivers (all in a single group). Why does performance decrease with the number of receivers? Figure 15 shows that receivers are not CPU-bound, but the sender is. Profiling reveals that the CLR at the sender is taking over the CPU; specifically, memory allocation and garbage collection costs are growing by as much as 10-15% (Figure 16). Inspecting the managed heap shows that memory is used mostly by the multicast messages pending ACK on the sender: these have to be buffered for the purpose of loss recovery. Memory usage is actually 3 times larger than what the number of pending messages would suggest: at these high data rates, the CLR can't garbage collect fast enough, hence old data is still lagging in memory. Acknowledgement latency is caused by the increase in the time to circulate a token around the region for the purpose of state aggregation (token "roundtrip time"). Hence, our throughput degradation is ultimately caused by the latency to collect control information by the protocol. Just a 500ms increase in token RTT resulted in 10MB extra memory usage, inflated overhead by 10-15%, and degraded throughput by 5%. The need to reduce this latency to ensure a smooth token flow was among the key reasons for the architectural decisions outlined in the preceding section. Using a deeper hierarchy of token rings would also help to alleviate this problem (and indeed, this idea led us to the design proposed in [20]). On the other hand, simply increasing the token rates helps only up to a point (Figure 17), and causing tokens to carry larger amounts of feedback per round by making this amount proportional to the region size increases processing complexity, and despite memory saving, it is counterproductive (Figure 18).

5.2 Memory Overheads on the Receiver

The growth in cached data at the receivers repeats the pattern of performance linked to memory. The increase in the amount of such data slows us down, despite the fact that receiver CPUs are half-idle (Figure 15). How can memory overhead affect a half-idle node? Figure 19 shows results of an experiment where we varied *replication factor*, the number of receivers caching a copy of each message, causing a linear increase of memory usage. We see a super-linear increase of the token roundtrip time and a slow increase of the number of messages pending ACK on the sender, causing a sharp decrease in throughput (Figure 20). The underlying mechanism is as follows. The increased garbage collector activity and allocation overheads slow nodes down, and processing of the incoming packets and tokens takes more time. Although this is not significant on just a single node, it accumulates, since a token must visit all nodes to aggregate state. Increasing the number of caching replicas from 5 to all 200 nodes in the region, increases token RTT 3-fold!

5.3 Overheads in a Perturbed System

Another question to ask is whether our results would be different if the system experienced high loss rates or was otherwise perturbed. To find out, we performed two experiments. In the "sleep" scenario, one of the receivers experiences a periodic, programmed perturbation: every 5s, QSM instance on the receiver suspends all activity for 0.5s. This simulates the effect of an OS overloaded by disruptive applications. In the "loss" scenario, every 1s the node drops all incoming packets for 10ms, thus simulating 1% bursty packet loss. In practice, the resulting loss rate is up to 2-5%,

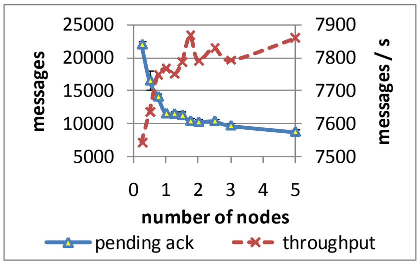


Figure 17. Throughput and the # of messages pending ACK as a function of token circulation rates.

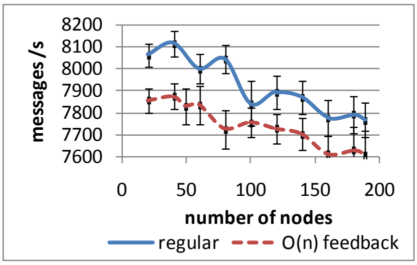


Figure 18. With O(n) feedback performance is worse due to higher overhead, despite the savings on memory usage.

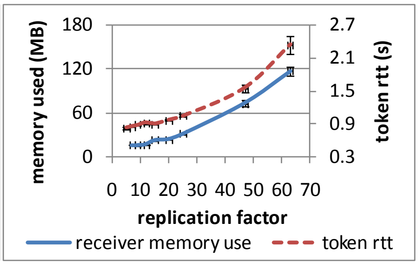


Figure 19. Results of varying the number of caching replicas per message in a 192-node region.

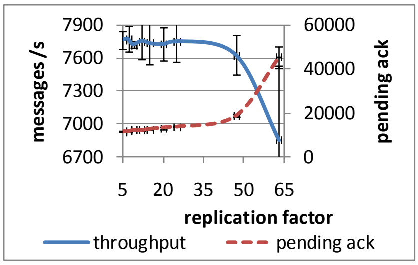


Figure 20. As a # of caching replicas increases, throughput decreases despite CPUs on receivers being 50% idle.

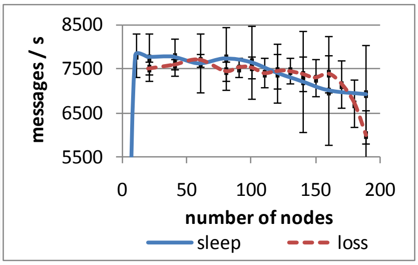


Figure 21. Throughput in the experiments with a perturbed node (1 sender, 1 group).

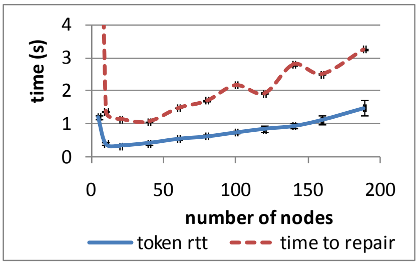


Figure 22. Token roundtrip time and the time to recover in the "sleep" scenario.

because recovery traffic interferes with regular multicast, causing further losses. In both scenarios, CPU usage at the receivers is in the 50-60% range and doesn't grow with system size, but the throughput decreases (Figure 21). In neither case does this decrease in throughput seem to be correlated to the loss rate. It does, however, correlate perfectly to the token RTT and memory utilization on the sender (Figure 22, Figure 23), repeating again the by now familiar pattern.

A closer look at this experiment reveals that while the increased ACK latency and resulting memory usage can be explained by the extra token rounds needed to perform recovery, the 2-fold overall increase in token RTT in these scenarios, as compared to undisturbed experiments, can't be as easily explained. The problem can be traced to a priority inversion. Because of repeated losses, the system maintains a high volume of forwarding traffic. Forwarded data tends to get ahead of tokens both on a sending and on a receiving path. As a result, tokens are slowed down.

5.4 Overheads in a Lightly-Loaded System

We've just discussed a perturbed system, now what if it's lightly loaded? We'll see that load has a super-linear impact on overheads. As we increase the multicast rate, the linear growth of traffic, combined with our fixed rate of state aggregation, linearly increases the amount of unacknowledged data and memory usage on the sender (Figure 24). This triggers higher overheads. For example, the time spent in the GC grows from 50% to 60%. Combined with the linearly growing demand for CPU due to the increasing volume of traffic, these effects together cause the super-linear growth of CPU overhead on the sender (Figure 15). The overhead skyrockets at the highest rates because the increasing amount of I/O slows down processing of control messages; much as in Section 5.2. We can confirm this by looking at the end-to-end latency (Figure 25), or at the delay in firing timer events (Figure 26), which at the highest rates get starved by the I/O.

5.5 Per-Group Memory Consumption

In this set of experiments, we explore scalability in the number of groups. One sender multicasts to a varying number of groups, in a round-robin fashion. Each receiver joins the same groups; the system contains just one region. QSM's regional recovery protocol is oblivious to the groups, but the system maintains a number of per-group data structures, which affects the memory footprint (Figure 27). Memory being involved, we expect the familiar pattern, where an increased memory usage triggers GC and decreases the throughput, and

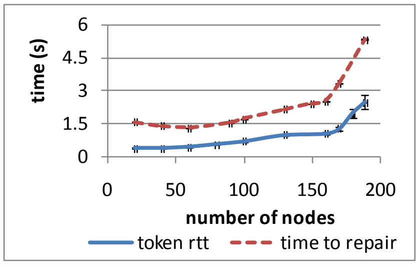


Figure 23. Token roundtrip time and the time to recover in the "loss" scenario.

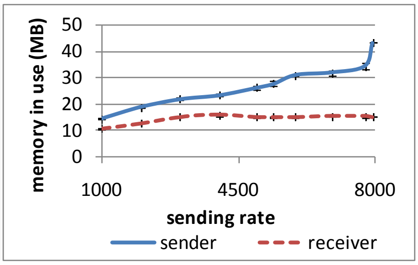


Figure 24. Linearly growing memory use on a sender and flat usage on receivers as a function of the sending rate.

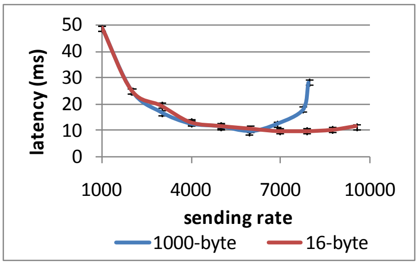


Figure 25. Latency measured from sending to receiving for varying sending rate and with various message sizes.

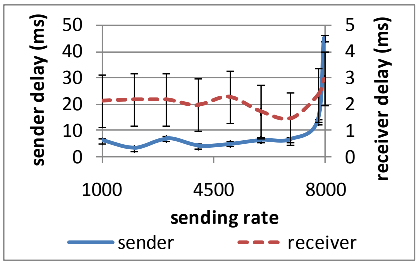


Figure 26. Delays in firing of timer events as a function of the sending rate, demonstrating "starvation through I/O".

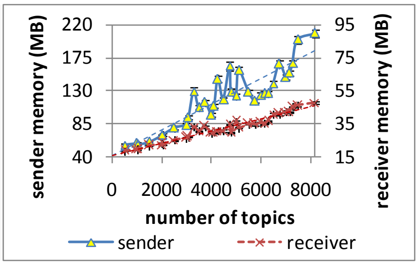


Figure 27. Memory use grows with the # of groups. Beyond a threshold, the system becomes increasingly unstable.

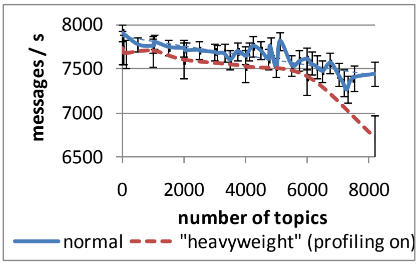


Figure 28. Throughput decreases with the # of groups (1 sender, 110 receivers, all groups perfectly overlap).

this is indeed the case (Figure 28). The effect becomes even clearer if we turn on extra tracing in per-group protocol stack components. This tracing is lightweight and has no effect on CPU, but increases memory usage, which burdens the GC. As expected, now throughput degrades even more (Figure 28, "profiling on").

A closer look at this scenario provides an even deeper insight. Note how at 6000 groups throughput degrades sharply (Figure 28) due to the increased token RTT and control latency (Figure 29). The growth of overhead suddenly becomes super-linear, and event at 4000 groups we are actually starting to see spikes of occasional packet loss, clear signs of slight instability. Detailed analysis again points to sender overhead as the culprit. Most delays come from 40% of tokens (Figure 30), since it is caused by disruption in their flow, not system-wide increase of overhead. This disruption is caused by the sender, which is busy and delays about 10% of the tokens (Figure 31), causing irregularities in their flow. The magnitude of this delay increases with the number of groups.

6. Discussion

Our experiments clearly show that memory is a performance-limiting factor in QSM, and that its cost is tried to latency by a positive feedback loop. Our results aren't specific to QSM and .NET; while managed environments do have overheads, we believe the phenomena we're observing are universal. Application with large amounts of buffered data may incur high context switching and paging delays, and even minor tasks get costly as data structures get large. Memory-related overheads can be amplified in distributed protocols, manifesting as high latency when nodes interact. Traditional protocol suites buffer messages aggressively, so existing multicast systems certainly exhibit such problems no matter what language they're coded in or what platform they use. The mechanisms QSM uses to reduce memory use, such as event prioritization, pull protocol stack or cooperative caching, should therefore be broadly useful. Below, we list our design insights.

1. ***Exploit structural regularity.*** We've recognized that even in irregular overlap scenarios one can restructure the problem to arrange for regularities, which can then be exploited by the protocol. This justified focus on optimizing performance in the scenario with a single heavily loaded set of regularly overlapping groups.
2. ***Minimize memory footprint.*** This applies especially to messages cached for recovery purposes.
   a. ***Pull data.*** Most protocols accept data whenever the application or a protocol layer pro-

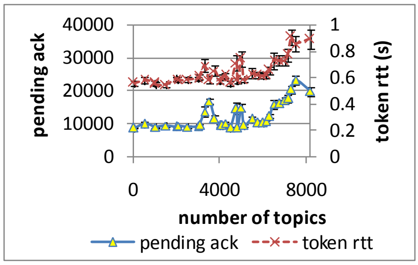


Figure 29. # messages pending ACK and a token RTT as a function of the # of perfectly overlapping groups.

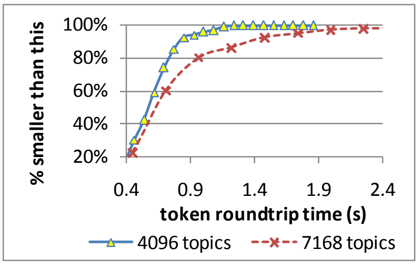


Figure 30. Cumulative distribution of the token RTT with 4096 and 7168 groups.

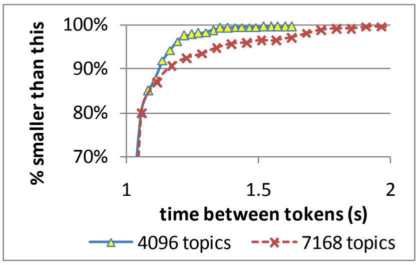


Figure 31. Cumulative distribution of the intervals between the subsequent tokens with 4096 and 7192 groups.

duces it. In contrast, by using an upcall driven "pull" architecture, QSM can delay generating messages until the very last moment and thus prevents data from piling up in the buffers.

b. ***Limit buffering and caching***. Most protocols buffer and cache data rather casually for recovery purposes. QSM avoids buffering and uses distributed, cooperative caching. Paradoxically, by reducing memory overheads, the reduction in cached data allows for a much higher performance.

c. ***Clear messages out of the system quickly***. Data paths should have rapid data movement as a key goal, to limit the amount of time packets spend in the send or receive buffers.

d. ***Message flow isn't the whole story***. Most protocols are optimized for steady low-latency data flow. To minimize memory usage, QSM sometimes tolerates an increased end-to-end latency for data, so as to allow for a faster flow of the control traffic; this allows faster cleanup and recovery.

3. ***Minimize delays***. Most situations in which we observed convoys and oscillatory throughputs can be traced to design decisions that permitted scheduling jitter or some form of priority inversion, delaying crucial messages behind less important ones. Implications included the following.

a. ***Event handlers should be short, predictable and terminating.*** Using the event-driven model consistently allowed us to eliminate the need for locking or preemption; we obtained a more predictable system, and got rid of multithreading, with its associated context switching overheads.

b. ***Drain input queues.*** From a memory footprint perspective, one might prefer not to pull in a message until QSM can process it. In data centers and clusters, though, most losses occur in the OS, not in the network, and loss rates soar if packets are left in the system buffers for too long.

c. ***Control the event processing order.*** In QSM, this involved single-threading, batched asynchronous I/O, and internal event prioritization. Small delays add up in large systems: tight control over event processing largely eliminated convoy effects and the oscillatory throughput problems.

d. ***Act upon fresh state.*** Our pull architecture has the added benefit of letting us delay the preparation of status packets until they are about to be transmitted, thus minimizing the risk that nodes act on stale information and trigger retransmissions that aren't longer needed, or other overheads.

4. ***Handle disruptions gracefully***. Broadcast storms are triggered when recovery itself becomes disruptive, causing convoy effects or triggering bursts of even more loss. In addition to the above, QSM employs the following techniques to keep balance.

a. ***Limit resources used for recovery***. QSM limits the maximum rate of the recovery traffic and delays the creation of recovery packets to prevent such traffic from overwhelming the system.

b. ***Act proactively on reconfiguration***. Reconfiguration after joins or failures can destabilize the system: changes reach different nodes at different times and structures such as trees and rings can take time to form. To address this, senders in QSM briefly suspend multicast on reconfiguration and receivers buffer unknown packets for a while in case a join is underway.

c. ***Balance recovery overhead***. In some protocols, bursty loss triggers a form of thrashing. QSM delays recovery until a message is stable on its caching replicas; then it coordinates a parallel recovery in which separate point-to-point retransmissions can be sent concurrently by tens of nodes.

7. Conclusions

The premise of our work is that new options are needed for performing multicast in modern platforms, specifically in support of a new drag-and-drop style of distributed programming inspired by web mash-ups, and for use in enterprise desktop computing environments, or in datacenters where multi-component applications may be heavily replicated. Using multicast in such settings requires a new flavor of scalability - to large numbers of multicast groups - largely ignored in previous work. QSM achieves this by exploiting regularities and commonality of interest.

Our performance evaluations led to a recognition that memory can be surprisingly costly. The techniques that QSM uses to reduce such costs and maintain high stable throughput despite perturbations should be useful even in systems that do not run in managed runtime environments.

8. Acknowledgements

Our work was funded by AFRL/IF, with additional funds from AFOSR, NSF, I3P, and Intel. We want to thank Jong Hoon Ahnn, Mahesh Balakrishnan, Lars Brenna, Lakshmi Ganesh, Maya Haridasan, Chi Ho, Ingrid Jansch-Porto, Tudor Marian, Amar Phanishayee, Stefan Pleisch, Robbert van Renesse, Yee Jiun Song, Einar Vollset, and Hakim Weatherspoon for feedback.

9. References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and 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] B. Ban. Performance Tests JGroups 2.5. http:// jgroups.org/javagroupsnew/perfnew/Report.html

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. SIGCOMM'02.

[5] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. TOCS 17(2), 1999.

[6] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in a Cooperative Environment. SOSP'03.

[7] G. Chockler, E. Melamed, Y. Tock, and R. Vitenberg. SpiderCast: a Scalable Interest-Aware Overlay for Topic-Based Pub/Sub Communication. ACM DEBS 2007.

[8] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. IEEE JSAC 20(8), 2002.

[9] E. Decker. http://researchweb.watson.ibm.com/compsci/project_spotlight/distributed/dsc/.

[10] D Dolev, and D Malki. The Transis Approach to High Availability Cluster Communication. CACM 39(4), 1996.

[11] X. Gabaix, P. Gopikrishnan, V. Plerou, H. E. Stanley. A Theory of Power-Law Distributions in Financial Market Fluctuations. Nature 423, p. 267-270, 2003.

[12] 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.

[13] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. The Reliable Multicast Design Space for Bulk Data Transfer, RFC 2887, August 2000.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. OSDI'00.

[15] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A Group Membership Service for WANs. ACM TOCS 20(3), p. 191-238, August 2002.

[16] B. N. Levine, and J. J. Garcia-Luna-Aceves. A Comparison of Reliable Multicast Protocols. Multimedia Systems 6: 334-348, 1998.

[17] H. Liu, V. Ramasubramanian, and E.G. Sirer. Client Behavior and Feed Characteristics of RSS, A Publish-Subscribe System for Web Micronews. IMC 2005.

[18] S. Maffeis, and D. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communication Magazine, Vol. 14, No. 2, Feb. 1997.

[19] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, and T. P. Archambault. The Totem System. FTCS 25 (1995).

[20] K. Ostrowski, K. Birman, and D. Dolev. Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. JWSR 4(4), 2007.

[21] K. Ostrowski, K. Birman, and D. Dolev. Declarative Reliable Multi-Party Protocols. Cornell University Technical Report, TR2007-2088. March, 2007.

[22] K. Ostrowski, K. Birman, D. Dolev, and J. Ahnn. Programming with Live Distributed Objects. ECOOP'08.

[23] C. Papadopoulos, and G. Parulkar. Implosion Control For Multipoint Applications. 10th Annual IEEE Workshop on Computer Communications, Sept. 1995.

[24] S. Pingali, D. Towsley, and J. F. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. SIGMETRICS'94, pp. 221-230.

[25] B. M. Roehner. Patterns of Speculation: A Study in Observational Econophysics. Cambridge University Press (ISBN 0521802636). May 2002.

[26] L. Rodrigues, K. Guo, P. Verissimo, and K. Birman. A Dynamic Light-Weight Group Service. Journal of Parallel and Distributed Computing 60:12. 2000.

[27] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System. IASTED PDCS 2005.

[28] Y. Vifgusson, K. Ostrowski, K. Birman, and D. Dolev. Tiling a Distributed System for Efficient Multicast. Unpublished manuscript.