Scalable Publish-Subscribe in a Managed Framework

Krzysztof Ostrowski
Cornell University

Ken Birman
Cornell University

Abstract

Reliable multicast, publish-subscribe and group communication are highly effective in support of replication and event notification, and could serve as the enabling technologies for new types of applications that are both interactive and decentralized. To fully realize this vision, we need a high-performance, scalable, and reliable multicast engine as an integral part of the runtime environment. Since the majority of development today is done in managed, strongly-typed environments such as Java or .NET, integration with such environments is of particular importance. What factors limit performance and scalability of a reliable multicast engine in a managed environment? What support from the runtime could improve performance, avoid instabilities, or make such systems easier to build? This paper sheds light on these questions by analyzing the performance of QuickSilver Scalable Multicast (QSM), a new multicast protocol and system built entirely in .NET. Memory-related overheads and scheduling-related phenomena are shown to dominate the behavior of our system. We discuss techniques that helped us alleviate some of these problems, and point to areas where better support from the runtime would be desirable.

1. Introduction

The work[1] reported on in this paper represents a step towards a flexible general-purpose development platform based on a scalable, reliable, high-performance variant of the *publish-subscribe* paradigm.

In this section, we explain why such platform is necessary, and why it is hard to build with existing publish-subscribe technologies. We argue that to realize its full potential, such platform has to meet two important requirements: (a) deliver high performance, reliability, and scalability in several important dimensions, and (b) deeply integrate with the development environment, programming language and type system. Because most of today's development is done in managed environments, such as Java or .NET, we believe that the second requirement can only be satisfied by building a platform that is an integral part of a managed environment, or indeed one that is implemented in a managed language such as .NET. This paper is dedicated to answering some of the questions related to how to build a high performance, reliable and scalable multicast platform in a managed runtime environment.

[1] Our work was supported by grants from AFRL, AFOSR, DARPA, Intel, and NSF. Contacts: krzys@cs.cornell.edu, ken@cs.cornell.edu. QSM is free, available for download at http://www.cs.cornell.edu/projects/quicksilver/QSM/.

Several technologies that are publish-subscribe in flavor exist, and are known to simplify the construction of distributed systems. Commercial publish-subscribe, focused on event notification or message queueing, is a popular middleware technology. It has been applied by companies such as Amazon.com as a core mechanism for component integration in their data centers. Virtually synchronous group communication, in which groups of processes are the equivalent of publish-subscribe topics, has been used for building high-performance replicated services in the New York and Swiss Stock Exchange, the French Air Traffic Control System, and the US Navy AEGIS warship [5]. Other forms of reliable multicast, such as SRM [6] or RMTP [7], have been successfully used in a variety of high-performance streaming scenarios.

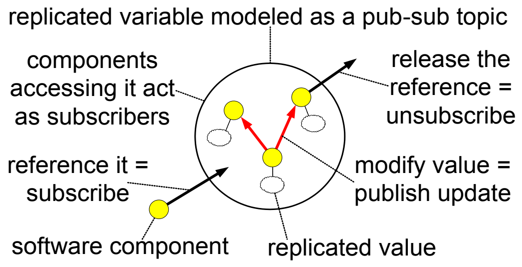
replicated variable modeled as a pub-sub topic
components accessing it act as subscribers
release the reference = unsubscribe
reference it = subscribe
modify value = publish update
software component
replicated value

Figure 1. Publish-subscribe services are generalizations of a replicated, writable variable.

One of the reasons that the paradigm has been popular is because it offers natural support for many applications that cannot be efficiently implemented using other approaches. For example, it can support services that are simultaneously *decentralized* (no dedicated central server is needed to host the service) and *interactive* (multiple clients can concurrently, consistently, and reliably modify the state of the service). These properties are hard to achieve using other popular distributed programming models, such as *client-server* and *peer-to-peer*. Client-server systems are interactive in a sense defined above and can provide reliability or QoS guarantees, but are centralized and hard to scale without costly hardware, infrastructure support and large maintenance overheads. Peer-to-peer systems such as BitTorrent, DHTs or content-distribution networks are decentralized and scale well, they are cheap and easy to deploy, but the flow of data is typically one-way, from the server to clients, latency can be very poor, and the end-to-end guarantees are weak. The sets of features offered by these paradigms are disjoint, and neither matches the need.

Reliable publish-subscribe services can fill this gap. To see this, think of a publish-subscribe topic as if it represented a replicated variable. Components accessing the variable subscribe to the topic; the "value" is replicated among all such components. To support persistence, a service can include one or more replicas that maintain historical logs or checkpoints, when a new client joins, it uses the history to catch up. The value is updated by disseminating changes in a reliable, ordered, and consistent way to the set of all subscribers (Figure 1). Publish subscribe can thus enable a style of programming in which shared variables are used casually and pervasively.

Publish-subscribe can also be used in other ways. One common configuration treats each topic as an event stream. This has emerged as a good fit with service-oriented data centers, in which large numbers of small services process requests collaboratively. A topic could also represent a stock in a trading system. The technology could even be used in embedded systems. For example, in an office building, a topic may represent a security policy governing a set of door scanners. The service "provided" by the topic here is a decentralized enforcement of the policy it represents, delegated to the door scanners by a central database. The policy definition, parameters, and policy-related data are replicated among the scanners, and any relevant events, e.g. policy updates, alerts, granting or revoking of access rights etc., generated by either the central database or the scanners, are reliably published to the topic members, directly by the devices that produced the events.

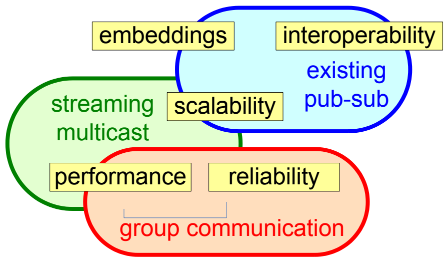

Figure 2. Existing publish-subscribe technologies are insufficient.

In each of these scenarios, communication passes directly between the components of the service (without indirection through a helper service). This is important, because it avoids the bottleneck, latency, single point failure concerns and overheads of indirect communication mediated by centralized services. Direct communication is already mandatory in large data centers, and will become even more so as the World Wide Web as a whole embraces dynamic and interactive content. For example, suppose that we move from today's Web, where users may interact with web pages, but hardly with each other, towards a dynamic, interactive virtual world composed of millions of virtual places. Instead of creating web pages users might create *virtual rooms*, design their interior, post multimedia content inside and link rooms with virtual corridors. Unlike web pages, these rooms could be interactive: users could walk between them, talk to each other, and see each other, as in the massively-multiplayer online games such as Second Life or World of Warcraft. We might think of each room as a service, its interior, content placed in it or user's positions as the service state, and the user's actions as the operations performed against the service.

Today, one would probably build such a system using a client-server approach, where all such services are hosted on a server farm, but this model is hard to scale to millions of users. Peer-to-peer approaches would host each virtual room on the machine of its creator, but doing so could easily overload that host, for example when someone with an elaborate avatar enters a room hosted by a slow machine. Modeling each room as a "replicated variable", and implementing it as a publish-subscribe service in the manner outlined above, removes bottlenecks and makes each content generator responsible for its own contribution to the data stream: a natural approach.

Although relatively successful, today's publish-subscribe technologies are inadequate for the kinds of uses we've suggested (Figure 2). The most popular commercial platforms lack end-to-end reliability, leaving it up to the application to ensure that the replicated state is updated consistently. Group communication toolkits offer strong flavors of reliability; but suffer from scalability issues and are perceived as hard to use by developers. Streaming multicast systems offer good throughput, but latency is often high, and such products only provide simple forms of reliability. We know of no system that simultaneously offers reliability, high performance, and scalability in the important dimensions mentioned above.

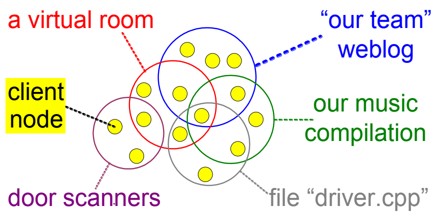

Figure 3. In large systems, there may be large numbers of topics. Different topics might represent different types of services that would require a truly large-scale platform to offer a variety of reliability and security guarantees.

In [1], [3] we argue[2] that existing approaches to reliable data dissemination fall into two classes that each scale poorly, although in different senses: (a) systems that run separate protocol instances per topic, like Isis [5], and (b) *lightweight group* approaches [11] such as Spread [8]. The Isis-like systems can't support large numbers of topics due to the linear per-topic overhead component. The lightweight-group systems vector all data through a small set of servers and then filter it prior to delivery; this works well in small scenarios, but can be inefficient in larger systems, and the indirection through servers introduces a bottleneck and latency. Moreover, raw performance of multicast systems that send data directly from sender to receivers is a problem. For example, we found ([1], [3]) that JGroups [10], a widely popular group communication component of JBoss, can't run at more than a fraction of the bandwidth of a 100 Mbps network, slows down significantly with 100 nodes, and collapses with 512 groups. Yet all of the examples given earlier require far greater scalability, and only make sense if the full performance of the hardware can be exploited. Moreover, existing systems are inadequately customizable. Different topics could represent different classes of replicated entities and require different reliability, security, QoS, fault-tolerance etc. guarantees (Figure 3).

But even if we had a high-performance, reliable group multicast or publish-subscribe platform that scaled in all important dimensions, we would need to address a second serious issue: many users find the paradigm poorly integrated with modern platform and development tools, making them hard to understand and deploy. As argued in [4], existing systems lack a standardized, flexible, general-purpose, easy to use API that decouples the application from the multicast platform used at the backend. Existing web services publish-subscribe standards are too simplistic and limited to be usable outside a narrow class of applications ([2]), while group communication systems employ proprietary APIs, e.g. requiring the developer to learn a new and domain-specific vocabulary.

[2] A note to the reviewer: We want to emphasize that no technical paper on QuickSilver has been published to date, in any venue. The citations are technical reports and short white papers that lack details or evaluation.

Much research has been dedicated into making the multicast more developer-friendly. One well known approach is the fault-tolerant CORBA [12] standard, which takes a CORBA service and transparently replicates it. However, transparency is costly, and the approach can only be used with unthreaded, deterministic applications. To leverage the full potential of publish-subscribe services, we need the flexibility to match the protocol to the application, and a deployment model better matched to modern platforms and architectures. Our premise is that the key to success will come not from transparency, but rather from a deep integration of publish-subscribe with the programming language and type system.

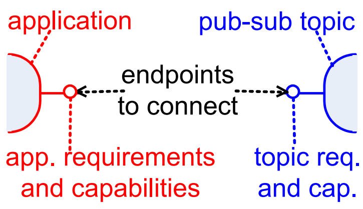

Figure 4. Matching typed topic and application endpoints.

Although we are well advanced on implementing a version of QuickSilver (QS/2), which offers this sort of deep embedding into .NET, the work is still in progress and discussion of the associated issues would be beyond the scope of this paper. Instead, we'll just summarize some of the implications. QS/2 is designed to be tightly integrated with the runtime environment and type system; doing so adds multicast groups to .NET much in the way that typed objects are supported in .that system. In this approach, topics become first-class language entities, with <u>types</u> that represent their reliability or security properties, and the operations possible on a topic correspond to its type and are implemented by type-specific code. Type matching can be enforced at runtime (Figure 4). The platform also automates the generation of stubs for accessing existing, deployed topics, much as is done today for web services. Such embeddings of the paradigm into the language and the type system can greatly simplify programming, similarly to Language Integrated Queries (LINQ) or Windows Communication Foundation (WCF). Furthermore, they bring the advantages of strong typing into the realm of distributed computing, thus resulting in more robust, predictable, and better-behaved code. In the spirit of other component architectures, the choice of a protocol or its parameters can be postponed until runtime, and determined when the topic is created based on the type of the software component requesting access to the topic. Communication can be integrated with the eventing architecture, allowing the platform to interact with the application to retrieve buffered messages, perform message delivery, etc. Developers can specify types of reliability or security guarantees for their topics using a declarative language provided by the platform, and can therefore customize the platform, design their own protocols, and share code in the spirit of collaborative development. Although we've worked in the context of .NET, the system should port elegantly into J2EE or CORBA.

But the full QS/2 system is still under development. The remainder of this paper is dedicated to the questions of: <u>how the decision to run in a managed runtime environment affects performance and scalability of a high-</u>

performance, scalable, reliable multicast engine, what are the dominant phenomena and issues that arise, what mechanisms can be used to alleviate these problems, and what support from the managed runtime could facilitate building such systems. We base the discussion on our experiences building and evaluating Quicksilver Scalable Multicast (QSM). Unlike QS2, QSM offers only a high-performance multicast substrate with an ACK-based reliability property similar to [6] or [7], and the Windows embedding doesn't take full advantage of the type mechanisms available in the .NET framework. Nonetheless, it scales in several major dimensions, tolerates several different types of perturbances; and the extension to the full platform isn't expected to change these characteristics. Moreover, we have early users and believe that QSM is a useful and powerful system in its own right. QSM establishes that, with proper care, a high-performance communications platform can operate within a managed setting. It sustains multicast rates as high as 9500 message/s for 1000-byte messages on a 200-node cluster of 1.3GHz Pentium III workstations connected with a 100Mbps switched LAN, a value close to the maximum capacity of our hardware. Throughput degrades by only a few percent as the system scales to 200 members or to 8000 groups. QSM was written entirely in .NET, mostly in C#. Only about 2.5% of our code is in C++, and only to provide direct access to Windows I/O completion ports, which are not adequately supported in C#.

Although most results we report on here are specific to our system and protocol, we believe that our findings are generally applicable. When we set out to build QSM, we assumed that operation in a managed setting would impose insurmountable overheads relative to unmanaged code in a language like C++, and that we would simply need to tolerate these overheads to gain the benefits of closer platform integration. Today, we've come to appreciate that managed environments are not necessarily incompatible with even the most performance-demanding uses. Indeed, although detailed comparisons with other platforms are outside the scope of the work reported, QSM is faster and more scalable than any other multicast or publish-subscribe platform with which our group has worked during twenty-five years of interest in the technology. The system may be the fastest, most stable, and most scalable multicast platform in existence.

For reasons of brevity the discussion of the protocol and architecture included in this paper is limited; additional details, including more discussion of the rationale behind our design choices and a more comprehensive comparison with related work can be found in our technical reports [1], [3]. Moreover, as noted, QSM is just a first step, and should be viewed not as a goal in itself, but as a prototype demonstrating feasibility and as a testbed. Generalizations and the continuation of this work are described in [2], [4].

2. Architecture

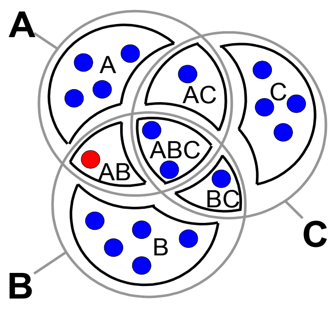

Figure 5. Topics overlap over regions. Topic A spans over regions A, AB, AC, and ABC.

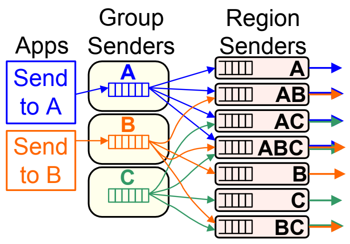

Figure 6. A message targeted at a topic is transmitted to all regions that topic spans over.

In the preceding section, we noted that existing systems that support reliable multicasting to multiple topics fall roughly into two classes: systems that run separate protocols per topic and those that treat the entire system as a single broadcast domain and filter on receive. Both techniques are intrinsically limited ([1], [3]). In QSM we employ a hybrid approach. The network is divided into a set of *regions*. The manner in which regions are defined may vary. In our prototype, regions are defined based on the similarity of interest: nodes **x** and **y** are in the same region iff **T**(**x**) = **T**(**y**), where **T**(**x**) is the set of topics that **x** is subscribed to (Figure 5). Our future work will explore other possible arrangements: regions can be defined as containing nodes with approximately similar interest (e.g. **x** and **y** can be in the same region iff **T**(**x**) and **T**(**y**) have 90% topics in common), constrained to be of size at least 20, or be defined with respect to administrative domain boundaries. Also, they might partially overlap. The problem is nontrivial, and is outside the scope of this paper.

Each region is a separate multicast domain with its own IP multicast address and a local recovery protocol. The set of subscribers to any given topic T spans over a number of regions, say R_1 through R_K. When sending to topic T, a sender node multicasts a message to each of the regions separately (Figure 6). Nodes in regions R_1 through R_K that did not subscribe to T or that receive duplicates can simply discard the message, without delivering it to the application. Note that by assigning separate regions to each topic, or by treating the entire system as a single region, we can mimic the two classes of existing approaches mentioned earlier. The technique just described thus generalizes both. We are not the first to make this observation: a similar approach, limited to unreliable multicast, is proposed in [13].

Our approach allows the sending overhead to be amortized between topics. If a node has two messages to send to a pair of topics T_1, T_2 that overlap on a region R, then while transmitting to R, the node can batch these messages together, as in the lightweight group approaches. At the same time, if regions are defined in a manner that takes into account similarities of interest, batching overheads will be small: a message sent to a region will be, with high probability, of interest to most region members. Finally, by moving flow and rate control to the regional level, we avoid the situation where thousands of per-group protocols compete for bandwidth. If regions are disjoint, as they are in our prototype, our protocol is a generalization of application multicast over TCP, with regions as the “destinations”.

Partitioning into regions is also applied to loss recovery. The key ideas behind our design, outlined below, are (a) a hierarchical composition of protocols, and (b) merging of protocols inside regions. To understand the former note that if **X** represents a set of nodes that can be partitioned into subsets, e.g. $\mathbf{X} = \mathbf{Y_1} \cup \mathbf{Y_2} \cup \ldots \cup \mathbf{Y_K}$, then the task of performing recovery in **X** may be implemented by performing local loss recovery in every $\mathbf{Y_i}$ (i.e. ensuring that for any pair of nodes in $\mathbf{Y_i}$ if one has a message **m**, then so eventually does the other), independently, and additionally, performing recovery across different $\mathbf{Y_i}$ (i.e. ensuring that for each pair of subsets $\mathbf{Y_i}$ and $\mathbf{Y_j}$, if nodes in one of them have a message, then so eventually do some nodes in the other). Note that each of these (**K+1**) subtasks, local recovery in each $\mathbf{Y_i}$ and across different $\mathbf{Y_i}$, can be thought of, and indeed, even implemented, as a separate, independently running "sub-protocol". Loss recovery in the entire **X** is achieved by running all those sub-protocols simultaneously (Figure 7, left). Recall that in QSM, each topic spans across a set of regions. For reasons that shall be explained later, each region is further sub-divided into *partitions*, and the partitions contain individual nodes. Recovery is implemented by running (a) a local recovery protocol among nodes in each partition, (b) a "higher-level" recovery protocol across all partitions in each region, and (c) a recovery protocol that runs across regions in each topic. Now, the protocol that runs in a given region performs recovery, simultaneously, for all topics that span over the region (Figure 7, right). The latter can be thought of as "merging" protocols running for different topics when they overlap on regions. By doing so we amortize overheads across topics, reduce the number of control packets, and control node "fan-out".

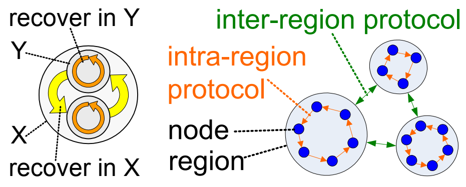

Figure 7. Hierarchical recovery in QSM (left) applied to topics vs. regions (right).

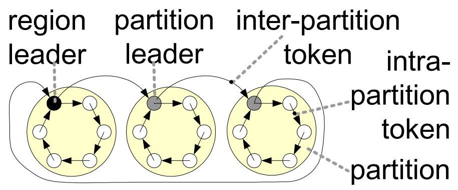

Figure 8. Our hierarchy of token rings.

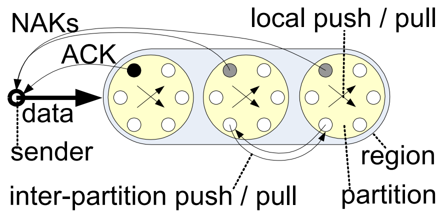

Figure 9. Recovery inside of and across partitions, and aggregate ACKs / NAKs.

The scheme just outlined can be implemented in a variety of ways, generalized, and used to implement protocols with stronger reliability properties ([2]). In QSM, we based the implementation on a token ring protocol, due to its simplicity, and on a simple ACK-based reliability property (in the future, we hope to offer stronger guarantees, implemented with essentially the same techniques ([2], [4]). At this stage, our primary goal was to demonstrate the feasibility of the approach and capture the basic phenomena related to scalability, high performance and the consequences of running in a managed environment, rather than the artifacts of enforcing a particular type of reliability.

Accordingly, nodes in each partition form a token ring (Figure 8). The token is used by neighboring nodes to compare the sets of messages they received, and to perform recovery by local push or pull forwarding (Figure 9). It is also used to calculate aggregate information about the partition, such as which messages have been received by all of the nodes in the partition. Finally, the token is used to distribute information about messages that have been received by any of the nodes, messages unlikely to still be in transit, ready to be cleaned etc. The token is released and collected by a selected *leader* node in the partition. Now, the partition leaders run another, "higher-level" token ring protocol; they compare aggregate partition information for the partitions they represent (collected by the "lower-level" tokens), and use it to calculate aggregate information about the entire region. The "higher-level" token is released, and collected, by a *region leader*, which uses the regional aggregate collected by this token e.g. to control cleanup in the region, or to generate ACKs for the sender. In our experiments, the region leader releases tokens about once per second, and accordingly, each region sends an aggregate ACK to every sender about once a second. Despite the minimal amount of feedback, the system is stable; indeed, reducing the amount of recovery-related burden the senders have to deal with was one of the crucial factors in achieving high performance and good scalability.

There are many ways in which different tokens (intra-partition tokens and inter-partition tokens) might be synchronized with each other; for example, they might not be synchronized at all. In QSM we adopted a simple scheme, where an intra-partition token is triggered by the arrival of the inter-partition token at the partition leader, and the inter-partition token is not passed over to the next partition leader till the intra-partition token completes a full round across the partition. Indeed we might think of there being only a single token in the entire region that "impersonates" inter-partition or intra-partition tokens as it jumps between or zooms around the individual partitions. As we shall see in the following section, state aggregation latency is by far the most critical factor that determines performance. Accordingly, the details of the hierarchical token protocol are an ongoing preoccupation in our effort; we will have more to say about this in the evaluation section, below.

Finally, to reduce memory overheads, we employ cooperative caching. Rather than cache each message on each of the receivers in the region for the purposes of local recovery, we designate approximately **k** nodes in the region as our *caching replicas* (an idea first proposed by Zhao [14]). Specifically, we subdivide a region of size **r** into $\lfloor \mathbf{r}/\mathbf{k} \rfloor$ partitions, each of size at least **k**, and we cache each message in a single partition, the responsibility distributed across the partitions in a round-robin fashion. In the resulting scheme, the overhead of caching of the received messages by any individual receiver decreases linearly with system size.

Responsibility for announcing the overall system configuration rests with a Global Membership Service (GMS), which processes subscribe and *unsubscribe* requests, detects node failures, and uses the latter to generate a sequence of membership views for each topic, much like a conventional GMS in a group communication system. Additionally, our GMS determines and continuously updates region boundaries, maintains sequences of region views for each region, and a mapping from group views to region views (Figure 10). The relevant parts of this structure and any changes to it are communicated in a reliable manner to the affected nodes, which rely on this information as a common knowledge, and use it e.g. to create and update the structure of token rings, elect leaders etc. In our prototype, the GMS is implemented by a single node. In future versions, the GMS will be hierarchical and fault-tolerant (along the lines of the Moshe scalable GMS [15]).

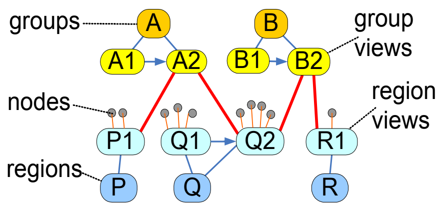

Figure 10. Tracking subscriptions and region membership by GMS in QSM.

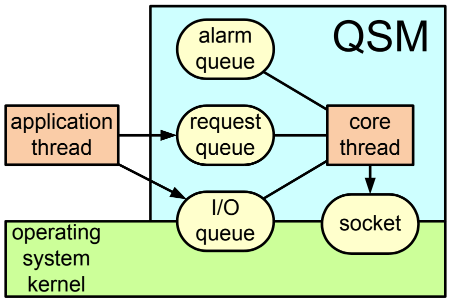

Figure 11. QSM uses a single-threaded architecture, with a "core" thread that controls three queues: for I/O requests, timer-based events, and requests from the possibly multithreaded application.

3. Implementation

Although modern languages like Java and C# encourage the use of threads, in QSM preemptive scheduling is disruptive, and adds unnecessary overhead: all tasks in the system are short, predictable and terminating [3]. Consequently, we implemented[3] QSM in a purely single-threaded, event-driven manner (Figure 11). 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 either in a blocking, or in a non-blocking manner, to retrieve I/O events. 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. For efficiency, events of the same type are processed in batches, up to the limit determined by a quantum (typically 50ms for I/O, 5ms for alarms, and no limits for the application requests). When an I/O event representing a received pack-

[3] An initial implementation was multi-threaded, but we soon realized that this was a mistake.

et is retrieved for a given socket, the socket may be synchronously polled and drained of packets to minimize the probability of loss. If there are no events to process, the core thread waits in a blocking system call.

This time-sharing policy is further refined by assigning priorities to different types of I/O events, and processing I/O similarly to the way Windows treats interrupts, by first retrieving all events from the I/O queue and draining sockets, pre-processing them only as much as is necessary to determine their type, placing events in priority queues, and only then processing them in the order of decreasing priorities (Figure 12). By prioritizing the processing of incoming I/O over sending-related events we reduce packet loss, and by prioritizing control packets over data we make the system more stable [3], for it helps to reduce control traffic latency, a factor critical for good performance.

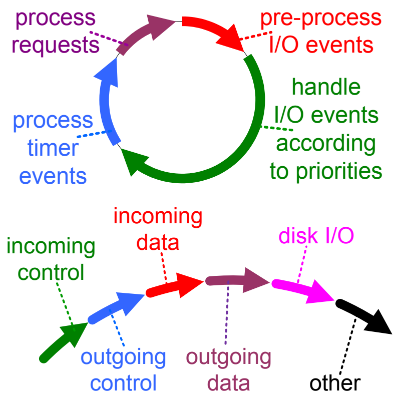

Figure 12. Our time-sharing and priority I/O processing policy.

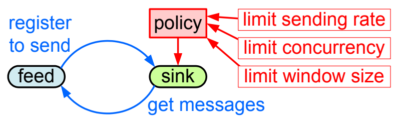

Figure 13. In our "pull" protocol stack a "feed" registers the intent to send with a "sink" that may be controlled by a policy limiting the send rate, concurrency etc. When the sink is "ready" to send, it calls the registered feeds for messages.

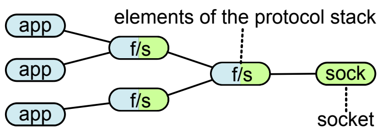

Figure 14. Elements of the protocol stack act as both feeds and as sinks, thus forming trees of such compoments, rooted at sockets.

The last aspect of the architecture relevant to this discussion is the "pull" architecture of our protocol stack. Much as the priority-based processing of I/O events allows us to reduce the control latency on the receiver side, a "pull" protocol stack reduces latency on the sender side by postponing the creation of messages until the time when transmission is actually about to take place. Control messages created "just-in-time" for transmission are more "fresh"; they contain more up-to-date information based on recent state of the sending node, which makes the system more stable ([3]). Additionally, the "pull" architecture allows us to almost eliminate buffering and reduce memory overheads, which, as we shall demonstrate, strongly affect performance.

In QSM, each element of the protocol stack acts as a *feed* that has data to send, or a *sink* that can send it (Figure 13), and most elements act as both (Figure 14). When a feed wants to send a message, rather than creating it and handing it down to the sink, as it is normally done in the "push" scenario, the feed only 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 of the message may also be postponed, as it is usually done in QSM, until the time when the sink polls the feed for messages to transmit. The sink determines when to send based on its control policy, such as rate, concurrency, or windows size limitation, and, unless the sink represents a physical socket, also based on the ability of the downstream sink to send.

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 unregistered.

4. 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. In this section, we focus on the latter.

All results reported here come from experiments on a 200-node 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 ordinary .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 figures include 95% confidence intervals, but they are often very small.

4.1. Memory Overheads on the Sender

On Figure 15 we show throughput in messages/s in experiments with 1 or 2 senders multicasting to a varying number of receivers, all of which subscribed to a single topic. With 1 sender, we let it transmit at an unlimited rate. This is possible because at very high rates the sender requires more CPU than the receivers and its CPU is not fast enough to saturate the network (Figure 16). With 2 senders, we report the highest combined send rate that the system could sustain.

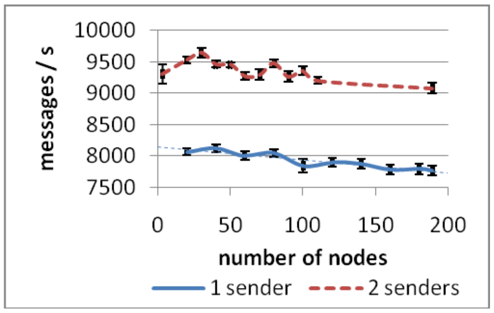

Figure 15. Throughput as a function of the number of nodes (1 topic, 1KB messages).

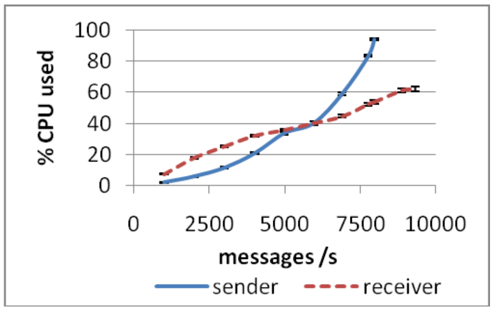

Figure 16. Processor utilization as a function of the multicast rate (100 receivers).

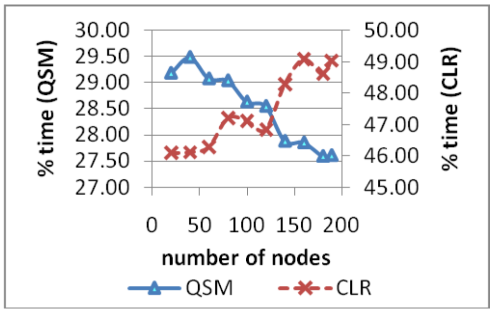

Figure 17. The percentages of the profiler samples taken from QSM and CLR DLLs.

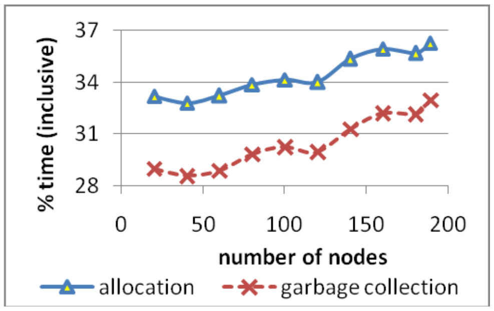

Figure 18. Memory allocation and garbage collection overheads on the sender node.

Why does performance decrease with the number of receivers? First, let's focus on a 1-sender scenario. Figure 16 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 17). More detailed analysis (Figure 18) 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). The former grows by 10% and the latter by 15%, as compared to 5% decrease of throughput. The bulk of the overhead is the allocation of byte arrays to send in the application ("JIT_NewArr1", inclusive, Figure 19). 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.

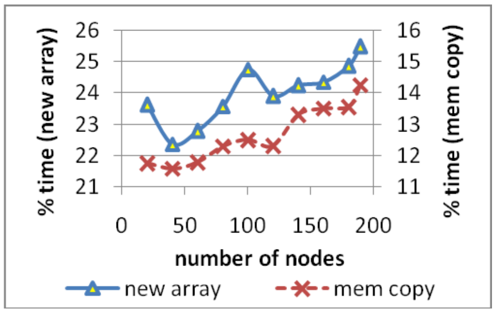

Figure 19. Time spent allocating the byte arrays in the application (inclusive), and time spent copying memory (exclusive).

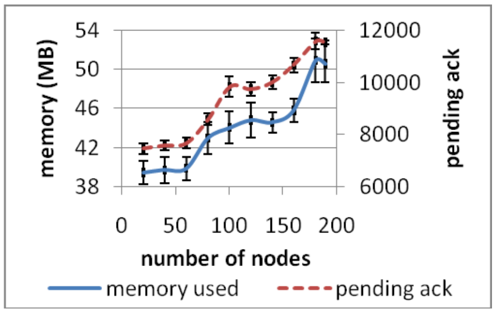

Figure 20. Memory used on sender and the number of multicast requests in progress.

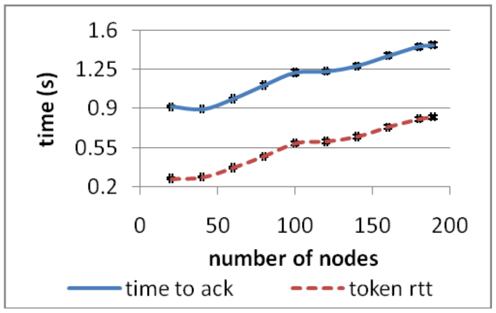

Figure 21. Token roundtrip time and an average time to acknowledge a message.

The increase in the memory allocation overhead and the activity of the garbage collector are caused by the increasing memory usage. This is caused by the increase of the average number of multicasts pending completion (Figure 20). For each of these multicasts, a copy of the message data is kept by the sender for the purpose of loss recovery. The reader might notice that memory consumption grows nearly 3 times faster than the number of messages pending acknowledgement multiplied by 1000-bytes (the size of the message data). Indeed, if we freeze the sender process in the debugger 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 are not immediately garbage collected, thus resulting in increased memory consumption.

The growing amount of unacknowledged data is caused by the increase of the average time to acknowledge a message (Figure 21). This grows because of the increasing time to circulate a token around the region for the 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. In larger experiments, the roundtrip time dominates.

These experiments show that the latency of state aggregation can be a critical factor determining performance. We just revealed a mechanism that directly links this latency to throughput, via increased memory consumption and the resulting increase in allocation and garbage collection overheads. An increase in latency by 500ms, resulting in a 10MB increase in memory consumption, can inflate these 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 from each other. This way, we could limit each level of the hierarchy, and hence the size of each ring, to be of size at most K (a fixed value), at the cost of using a more complex structure with $\log_K N$ levels of protocols, and the state aggregation latency would grow logarithmically.

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

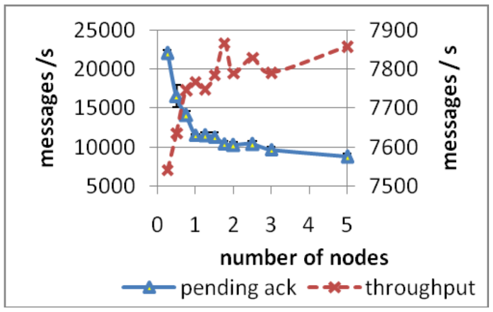

Figure 22. Varying token circulation rate.

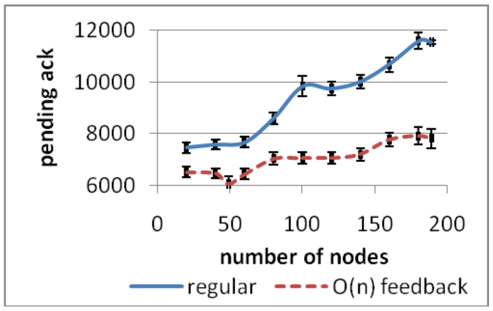

Figure 23. More aggressive cleanup with O(n) feedback in the token and in ACKs.

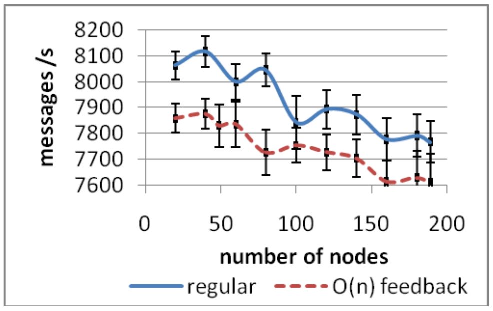

Figure 24. More work with O(n) feedback, and lower rates despite saving on memory.

In the first approach, we vary the intensity of aggregation by varying the rate at which tokens are released (Figure 22). Increasing it helps only up to a point. Beyond 1.25 tokens/s, more than one aggregation is in progress at a time and the work performed by tokens starts to be redundant. Furthermore, token processing itself and the ACKs it triggers represent overhead. Changing the default 1 token/s to 5 tokens/s decreases the number of unacknowledged data by 30%, but it increases throughput by less than 1%.

In the second approach, we increase the amount of state aggregated by the token and the amount of feedback provided to the sender. By default, 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, ($\mathbf{a_1}$, $\mathbf{b_1}$), ($\mathbf{a_2}$, $\mathbf{b_2}$), …, ($\mathbf{a_k}$, $\mathbf{b_k}$). The way this is to be interpreted by the sender is that messages with numbers from $\mathbf{a_1}$ to $\mathbf{b_1}$, from $\mathbf{a_2}$ to $\mathbf{b_2}$, and so on, up to messages from $\mathbf{a_k}$ to $\mathbf{b_k}$, can be cleaned up. Varying the value of **k** varies the intensity of cleanup by varying the ability of the system to "skip over" gaps in the sequence of stable messages. In the experiment shown on Figure 23 and Figure 24, we set **k** to the number of partitions, thus increasing it proportionally to the region size. Unfortunately, while the amount of acknowledged data is reduced by 30%, it still grows, and the throughput in this scenario is actually lower than in the default scenario, because the overall process is now more complex and consumes more CPU. Furthermore, the system now tends to behave in an unstable manner (notice the large variances in Figure 25), because our flow control scheme, based on limiting the amount of unacknowledged data, fails. While the sender can cleanup any portion of the message sequence, receivers have to deliver in the FIFO order, and the amount of data they cache is larger, and this reduces their ability to accept incoming traffic.

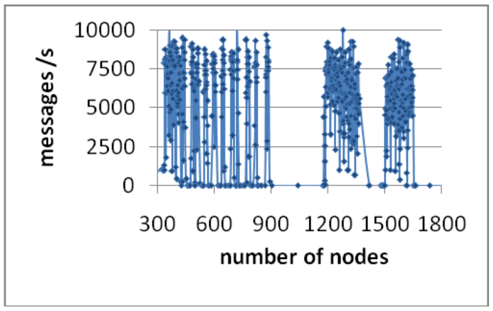

Figure 25. Instability with O(n) feedback.

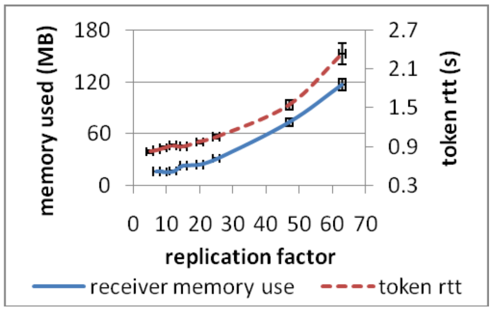

Figure 26. Varying the number of caching replicas per message in a 192-node region.

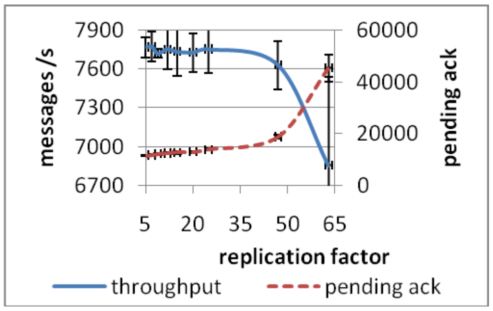

Figure 27. As the number of caching replicas increases, the throughput decreases.

4.2. Memory Overheads on the Receiver

The reader may doubt that memory overhead on receivers is an issue, considering that their CPUs are half-idle (Figure 16). Can increase in 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 caching replicas per message (replication factor). 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 (Figure 26), a slow increase of the number of messages pending ACK on the sender, and a sharp decrease in throughput (Figure 27).

The mechanism behind what we have observed 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 considered in isolation on a single node, it adds up: the token must visit all nodes to aggregate the recovery state. If we configure each node in a 192-node region to cache even 50% of the packets, as opposed to the default 5 replicas per packet, the token roundtrip time increases 3-fold. This delays state aggregation, increases pending messages and reduces throughput (Figure 27). With the highest replication factors, the sender's flow control policy kicks in, and the system goes into an oscillating state, similar to that on Figure 25, but milder. If each receiver caches *all* messages, the system can no longer run without rate control.

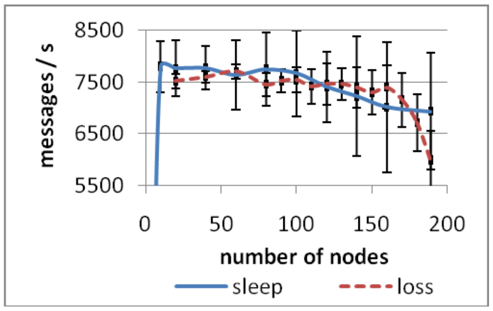

Figure 28. Throughput in the experiments with a perturbed node (1 sender, 1 topic).

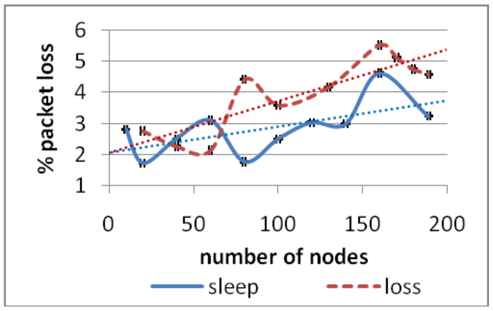

Figure 29. Average packet loss observed at the perturbed node.

4.3. 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 perform an experiment in which one of the receiver nodes is "flaky", it experiences a periodic, programmed perturbation. In the "sleep" scenario, every 5s the node spins for 0.5s. Because QSM is single-threaded, this essentially stops the flow of time. This simulates the effect of disruptive, very busy applications. In the "loss" scenario, every 1s the node drops all incoming packets for 10ms, thus simulating 1% of bursty packet. In practice, the observed loss rate is higher, around 2-5%, because recovery traffic interferes with regular multicast, thus causing extra losses.

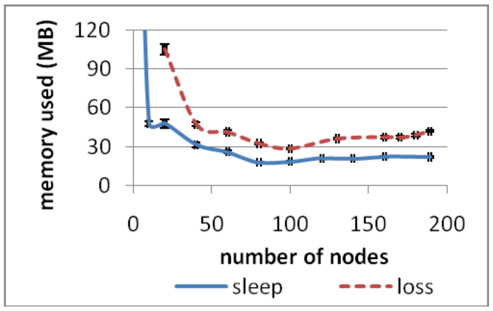

Figure 30. Memory usage at the perturbed node (at unperturbed nodes it is similar).

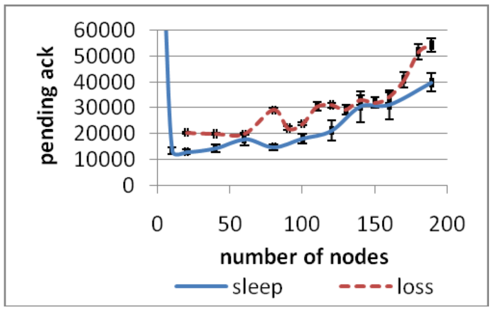

Figure 31. Number of messages pending ACK in experiments with perturbances.

In both scenarios, CPU utilization at the receivers in the 50-60% range and doesn't grow with system size, but throughput decreases (Figure 28). In case of sleep, 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 29). In case of controlled loss, 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 to memory used on neither the perturbed receiver (Figure 30), nor other receivers (not shown here). Indeed, at scales of up to 80 nodes, memory usage actually decreases thanks to our cooperative caching policy. The shape of the performance curve does, however, correlate quite well to the number of unacknowledged requests (Figure 31).

We conclude that the drop in performance in these scenarios can't be explained by correlating it to CPU activity, memory, or loss rates at the receivers, and 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 messages pending ACK starts at a higher level, and grows 6-8 times faster. This is due to the fact that the token roundtrip time is about 2 times longer, and if a failure occurs, it requires typically 2 token rounds on average to get repaired, plus another round to perform cleanup (Figure 32, Figure 33). These effects, combined together, account for the faster increase in acknowledgement latency.

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 to prove this, measuring the time alarms are delayed gives us a good insight into the magnitude of the problem. Recall that our time-sharing policy assigns

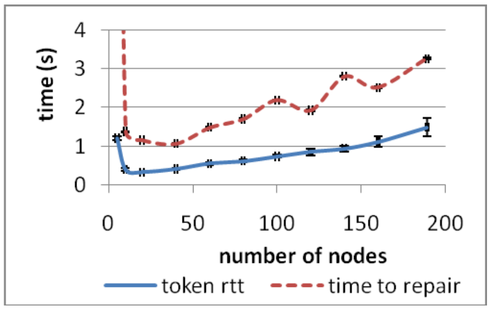

Figure 32. Token roundtrip time and the time to recover in the "sleep" scenario.

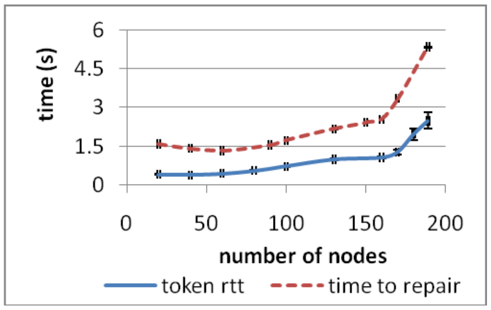

Figure 33. Token roundtrip time and the time to recover in the "loss" scenario.

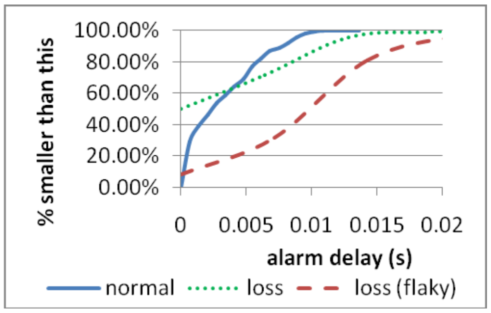

Figure 34. Histogram of maximum alarm delays in 1s intervals, on the receivers.

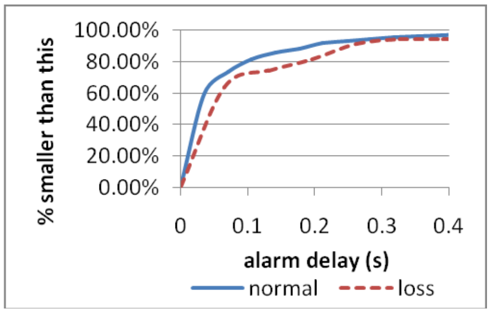

Figure 35. Histogram of maximum alarm delays in 1s intervals, on the sender.

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, at the cost of bigger delays in processing timer events. 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.

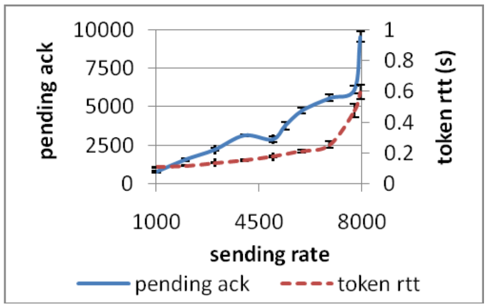

Figure 36. Number of unacknowledged messages and average token roundtrip time as a function of the sending rate.

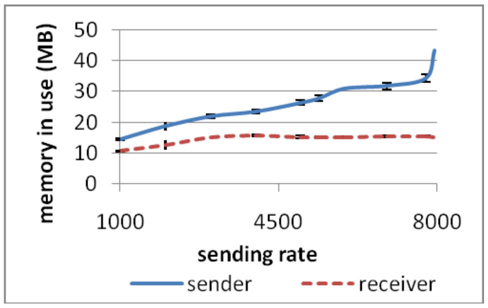

Figure 37. Linearly growing memory use on sender and the nearly flat usage on the receiver as a function of the sending rate.

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 (Figure 34, Figure 35). 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.

4.4. Overheads in a Lightly-Loaded System

So far the evaluation focused on scenarios where the system is highly loaded, with unlimited multicast rates and perturbances. In all cases, we have linked memory-related overheads or scheduling delays to degraded performance. Turning the question around: how does the system behave when lightly loaded? Do similar phenomena occur?

To answer this, we now vary the multicast rate. Figure 16 showed the load on receiver to grow roughly linearly, as indeed we expected given the linearly increasing load, negligible loss rates and the nearly flat curve of memory consumption (Figure 37), the latter thanks to 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, results in the increase of the amount of unacknowledged data (Figure 37), resulting in the increase of memory usage. The latter results in higher overheads: for example, the time spent in the garbage collector grows from 50% for the lower to 60% for the highest rates (not shown here). Combined with the already linearly growing CPU usage due to the increasing volume of traffic, these overheads cause the super-linear growth of CPU overhead shown on Figure 16. The growth of the number of unacknowledged requests and the resulting overheads appears to be super-linear as well at highest rates, due to the sharply rising token roundtrip time. The issue here is that the amount of I/O to be processed increases, much like in some of the earlier scenarios, tokens tend to be delayed by the growing amount of multicast traffic. We confirm the latter by looking at the end-to-end latency (Figure 38). Generally, we expect the latency to decrease as the sending rate increases because the system works more smoothly, avoiding context switching overheads and the extra latencies caused by the small amount of buffering in our protocols stack. However, after the rate exceeds 6000 packets/s, with larger packets the latency starts increasing again, due to the longer pipeline at the receive side and other phenomena mentioned above. The same is not true for small packets (Figure 38), for then 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 39). As the rate changes from 7000 to 8000, the latter on the receiver increases from 1.5ms to 3ms, and on the sender, from 7ms to 45ms.

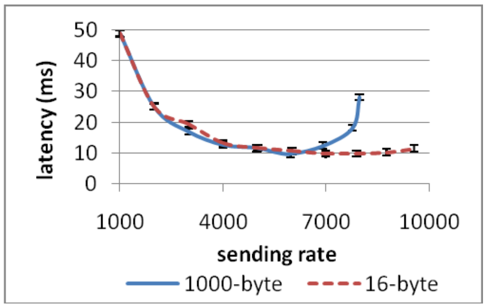

Figure 38. The send-to-receive latency for varying rate, with various message sizes.

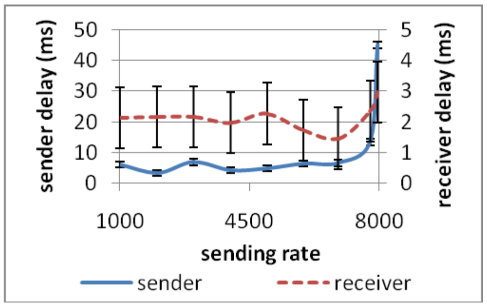

Figure 39. Alarm firing delays on sender and receiver as a function of sending rate.

4.5. Memory Footprint of the Protocol Stack

In the last set of experiments, we focus on scalability with the number of topics. A single sender multicasts to a varying number of topics in a round-robin fashion. All nodes subscribe to all topics, a contrived scenario that lets us focus purely on the effect of having multiple topics. Indeed, in our scenario, varying the number of topics affects *only the sender*. All topics map to a single region, and since the sender indexes messages destined to a region across topics, to form a single sequence ([1], [3]), the recovery protocol is oblivious to the number of topics. On the other hand, the sender must maintain a number per-topic protocol stack elements. This only affects the memory footprint, so any 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 ACK to vary with the number of topics, and until about 3500 topics they don't (Figure 40). However, in this range memory consumption on the sender grows (Figure 41), and so does the time spent in CLR (Figure 42), hurting throughput (Figure 43). Inspection of the managed heap in a debugger shows that the growth in memory usage in this range is caused not by messages, but by the per-topic elements of the protocol stack. Each maintains a queues, dictionaries, strings, small structures for profiling etc. With thousands of topics, these add up to tens of megabytes.

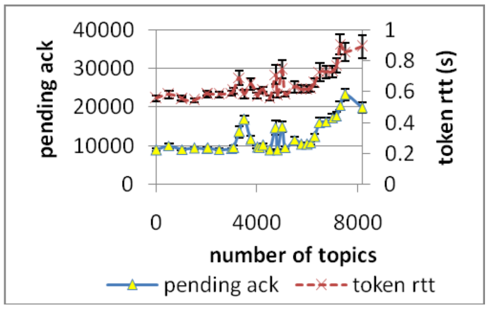

Figure 40. Number of messages pending ACK and token roundtrip time as a function of the number of topics.

We further confirm our intuition by turning on additional tracing in the per-topic components. This tracing is lightweight and has little effect on CPU, but it increases memory footprint with data structures that are actively updated once a second, which burdens the GC. Indeed, this decreases throughput (Figure 43, “heavyweight”).

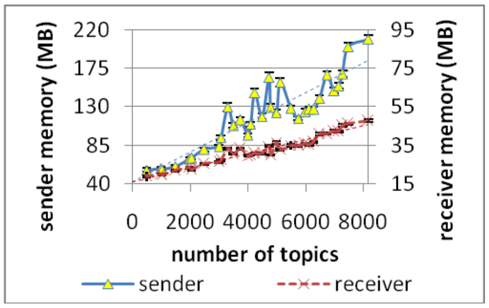

Figure 41. Memory usage grows with the number of topics. Beyond a certain threshold, the system is increasingly unstable.

It is worth noting that the memory usage values reported here are averages. Throughout the experiment, memory usage oscillates, and the peak values are typically 50-100% higher. With only 512MB total memory, already a 100MB average (and 200MB peak) memory footprint of the test process alone can be significant. With 8192 topics, peak footprint approaches 360MB, and the system is almost approaching the threshold beyond which it would start swapping. However, even with 3500-4000 topics we notice 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 topics. At around this point, we also start to see occasional bursts of packet losses (not shown), up to thousands of packets long, and noticeably, but not precisely correlated at different receivers. This triggers bursts of forwarding that aggravate the issue.

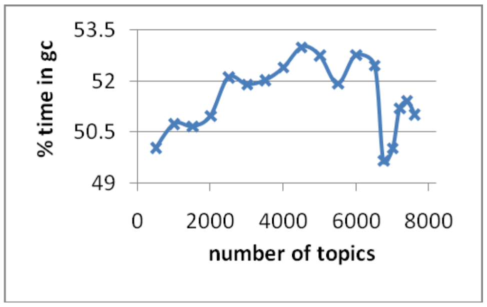

Figure 42. Time spent in the CLR code.

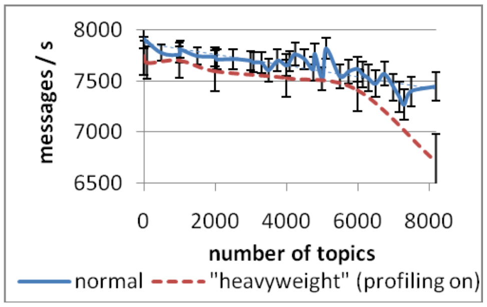

Figure 43. Throughput decreases with the number of topics (1 sender, 110 receivers, all topics have the same subscribers).

All these effects are ultimately rooted in the fact that the sender node is more loaded and less responsive. A 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 topics, 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 topics (Figure 44). Receiver performance indicators such as delays in firing timer event or CPU utilization don't show any noticeable trend.

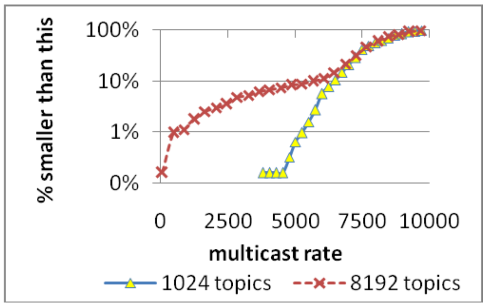

Figure 44. Cumulative distribution of the multicast rates for 1K and 8K topics.

The distribution of token roundtrip times for different numbers of topics shows that the increase of the token roundtrip time is caused almost entirely by only 50% of the tokens that are delayed the most (Figure 45), which points to disruptive events as the culprit, rather than a predictable increase of the token processing overhead. Tokens are commonly delayed on the sender. With many thousands of topics, 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 topics (Figure 46). At the very least, this decreases the efficiency of the process since tokens do partially redundant work. In extreme cases, we observe tokens disappearing. The latter is caused by a mechanism we have built into our system to avoid token convoys (such convoys carry out-of-date aggregates that trigger long bursts of redundant forwarding, destabilizing the system [3]). Finally, if a burst of losses occur, tokens may also be delayed at the receivers, as discussed earlier in the paper, although losses seem to occur rarely and they don't appear to be a major factor affecting performance. In a typical 10-minute run we observe a few bursts of losses. They cause spikes in the number of messages pending ACK, but rarely to such a level as to trigger the flow control mechanism.

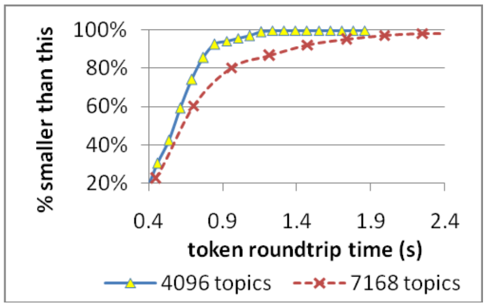

Figure 45. Token roundtrip times for 4K and 7K topics (cumulative distribution).

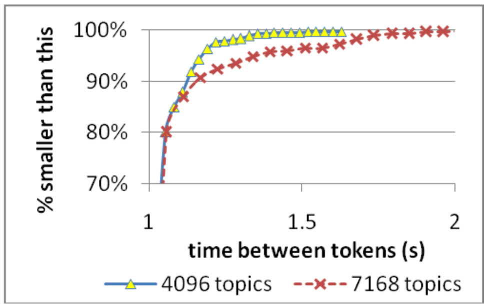

Figure 46. Intervals between the subsequent tokens (cumulative distribution).

5. Discussion

In the preceding section we identified some of the factors that affect performance and scalability of QSM. There are a number of forces at play, some of them mutually reinforcing, thus leading to a feedback loop that has a potential to inflate disruptive events caused by losses or busy applications to the level where they hurt performance (Figure 47). All these phenomena are ultimately tied to delays and latencies, which act as the common link through which the vicious cycle can sustains itself. The biggest sources of latency, at least in large configurations, are the protocol, and what we refer to as "scheduling delays". The latter represent the overall, cumulative time "penalty" imposed on important tasks such as processing a token. These delays may come from a variety of sources, some of which are discussed below. As demonstrated in our experiments, even small delays may be effectively "inflated" by the protocol, thus resulting in high latencies for critical tasks. As our experiments show, even seemingly low-priority tasks, such as collecting acknowledgements, may be critical and require low latency because of the high memory-related overheads in managed environments. The key to achieving high performance and very good scalability, at the most general level, thus lies in a combination of the following approaches.

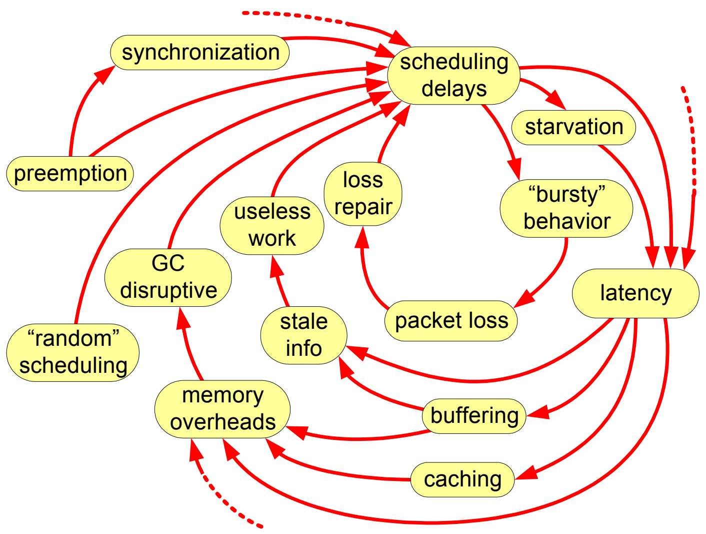

Figure 47. A variety of "forces" controlling the behavior of the system form a self-reinforcing "vicious cycle" that has the potential to inflate any temporary perturbances to the level where they hurt performance. All these "forces" are ultimately tied to memory overheads and various sorts of delays and latencies.

1. **Reduce "Scheduling Delays"**. This can be achieved by a variety of techniques discussed in this section.
2. **Reduce the "Inflating" Effect**. This effect occurs, for example, if any part of the protocol involves a component growing linearly with system size, as is was the case in QSM for the inter-partition token ring, thus causing latencies to add up. The issue may be alleviated by using a deeper hierarchy (e.g. more levels of token rings).
3. **Reduce "Hidden" Latencies**. An example of a hidden latency is one resulting from making the protocol unnecessarily synchronous, as it is the case in QSM, where inter-partition tokens are synchronized with the intra-partition tokens. The benefits of such synchronization are negligible, and the cost is the unnecessary waiting and a higher aggregation latency overall. In systems like ours, the more concurrency in the protocol, the better.

Below is a summary of techniques used in QSM or suggested by our experiments that can reduce scheduling delays.

1. **Avoid Multi-Threading**. The majority of tasks in QSM are short, predictable, and terminating, and preemption represents pure overhead. Preemption also requires synchronization, which leads to contention, and causes trivi-

al tasks to take on the order of milliseconds. We mentioned that an early version of QSM was multithreaded; single-threading dramatically improved performance (and eliminated concurrency bugs).

2. **Prioritize Processing**. Priority inversions such as those caused when a high volume of data delays tokens, forwarding or timer-based events, hurt performance and make the system unstable. In a high-performance system, transient problems are inevitable. The problem can be overcome by replacing "random" scheduling decisions (such as processing events in the order they show up) with our own priority-based, time-shared event processing policy. This made our system capable of stabilizing itself. As our experiments suggest, further benefits could be gained by making priorities more fine-grained, and by applying them on the send path too.
3. **Keep Components Lightweight**. All of our experiments point to memory overheads as one of the major reasons for degraded performance and scalability, with fluctuations as small as 10MB translating to 5% throughput penalty. While the effect may not be as significant in case of slowly-changing data structures, it is evident in our system with objects allocated and data structures updated thousands of times per second. As we have demonstrated, in a scalable system the memory footprint of the elements of the protocol stack should be kept low.
4. **Avoid Buffering**. Even a few thousand unacknowledged messages can affect performance by increasing memory overheads. Buffering messages in a protocol stack can easily aggravate the problem by orders of magnitude, especially in a scalable system that may need to maintain thousands of protocol stack elements, such as representing individual topics, each potentially with its own message buffer. In these cases, the damaging effect on performance far outweighs the benefits of buffering. Our "pull" protocol stack architecture reduced buffering in QSM to minimum, and enabled a smooth flow control between QSM and the OS.
5. **Limit Caching**. Caching, such as for the purpose of loss recovery, can be as disruptive as buffering, for the exact same reasons. Our system actually can't run at the highest rate if every receiver is caching every message. In practice, caching on more than a few nodes is counter-productive. Cooperative caching allowed us to make memory use on receivers basically a non-issue; indeed, our system works faster with 80 nodes than with just 20.
6. **Act on Fresh State**. Priority processing, and creating messages "just-in-time" for transmission in our "pull" protocol stack, both reduce the end-to-end latency for control packets, thus maximizing the "freshness" of the information the recipient acts upon. This is of critical importance particularly to the control traffic, where acting upon stale forwarding request or NAKs in earlier versions of QSM led to long bursts of redundant forwarding.

7. **Design With Delays In Mind**. One source of delays in our experiments arises from disruptive activity of other threads, such as the garbage collector, the application using QSM, or OS services running in background. Delays on the order of 10ms are common even in unperturbed runs, on an otherwise idle system with no applications besides QSM. These delays make it hard to e.g. efficiently implement accurate rate control without introducing significant "burstiness" because they affect any processing based on timeouts. In QSM, we implemented rate control as an adaptive mechanism that "overcompensates" for the delays inherent in the system ([1], [3]).
8. **Anticipate Convoys**. Convoy phenomena, such as when message cleanup is delayed by long burst of I/O or when a chain of tokens triggers a burst of forwarding, lead to starvation, priority inversion, and redundant work. In QSM, we had to resort to a number of mechanisms that explicitly prevent these from happening, such as quanta for I/O and timer events, terminating some of the tokens if they pile up etc.
9. **Beware Of Hidden Starvation**. Sustaining high performance sometimes depends on the application promptly handling events such as the successful completion of multicast, handling of errors etc. When such critical interactions are delayed, e.g. due to high volume of I/O, priority inversions may occur, where resources held by the application (e.g. memory) are not reclaimed, resulting in a degraded performance. Seemingly low-priority interactions with the application can thus sometimes turn out to be on the critical path.

We conclude the discussion by shifting focus from the overheads of QSM to some of the overheads that the system would incur if implemented as an unmanaged library. First, we speculate that this would result in buffering, for otherwise, the system would need to marshal calls from unmanaged to managed code through COM wrappers to pull data just in time for transmission, which leads to unacceptable overhead. Buffering, in turn, would lead to problems much like those discussed earlier. Secondly, we believe that message cleanup latency would be larger, in part because of having to marshal calls across COM domain boundaries, and in part because it would necessarily have to be done in a different thread than the thread running in context of the unmanaged engine, and thus it would involve the extra delay of rescheduling. These delays, in turn, would cause messages in the application to lag behind, and increase garbage collection overhead in the managed application process; potentially even more disruptive because it would involves a context switch to a different process. Also, it would prevent efficient use of scatter-gather I/O by requiring the transmitted data to be marshaled across domains, thus introducing the huge overhead of the extra memory copy on the critical path and increasing memory usage. Finally, efficient debugging, profiling, and exception handling are harder in unmanaged systems, and building a large system such as QSM in C++ would be a challenge. Moreover, it is much harder to ensure secure execution of dynamically loaded code, such as a "protocol driver" dynamically downloaded from the network ([2]), or provided by the application.

6. Conclusions

The premise of our work is that publish-subscribe and multicast can only achieve their promise if deeply integrated with managed environments. Doing so posed challenges to us as protocol and system designers, which were the primary focus of our paper. A central insight is that in managed settings, maintaining as small a memory footprint as possible is a key to high performance. With effort, QSM is able to achieve remarkable scalability and stability even at very high loads. We believe the techniques used would also be applicable in other systems and settings.

7. Acknowledgements

We are grateful to Mahesh Balakrishnan, Ranveer Chandra, Danny Dolev, Maya Haridasan, Tudor Marian, Greg Morrisett, Robbert van Renesse, Einar Vollset, and Hakim Weatherspoon for the feedback they provided.

8. References

[1] K. Ostrowski, K. Birman, A. Phanishayee. The Power of Indirection: Achieving Multicast Scalability by Mapping Groups to Regional Underlays. Cornell University Tech. Report, November 2005, http://www.cs.cornell.edu/~krzys/QSM-2005.pdf

[2] K. Ostrowski, K. Birman. Extensible Web Services Architecture for Notification in Large-Scale Systems. In Proceedings of the IEEE International Conference on Web Services (ICWS 2006), Chicago, IL, September 2006, pp. 383-392.

[3] K. Ostrowski, K.Birman, A. Phanishayee. QuickSilver Scalable Multicast. Cornell University Technical Report, April 2006, http://www.cs.cornell.edu/~krzys/QSM-2006.pdf

[4] K. Ostrowski, K. Birman, D. Dolev. Properties Framework and Typed Endpoints for Scalable Group Communication. Cornell University Technical Report, July 2006, http://www.cs.cornell.edu/~krzys/PropertiesFx.pdf

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

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

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

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

[9] QuickSilver distribution and the list of publications: http://www.cs.cornell.edu/projects/quicksilver/QSM/

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

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

[12] 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.

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

[14] Z. Xiao. Efficient Error Recovery for Reliable Multicast. Ph.D. Dissertation, Cornell University, January 2001.

[15] 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.