Implementing Scalable Publish-Subscribe in a Managed Runtime Environment

Krzysztof Ostrowski, Ken Birman
Cornell University

Abstract

The reliable multicast, publish-subscribe, and group communication paradigms 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? We focused on answering these questions by analyzing the performance of QuickSilver Scalable Multicast (QSM), a new multicast protocol and a system we've built entirely in .NET. We found that memory-related overheads and scheduling-related phenomena dominate the behavior of our system, and that most problems can be alleviated by techniques such as restructuring the protocol stack to limit caching and buffering, and introducing a custom, priority-based scheduling policy.

1. Motivation

The work[1] reported on in this paper[2] 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. Thus, the question of how to design a high performance, reliable, and scalable multicast platform to run within a managed runtime environment is an important one. And indeed, our work demonstrates that aspects such as garbage collection or multithreading are important factors limiting scalability and performance of such systems in managed runtimes.

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 [3]. Other forms of reliable multicast, such as SRM [4] or RMTP [7], have been successfully used in a variety of high-performance streaming scenarios.

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, one may think of a *publish-subscribe* topic as if it

[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, and is available at [16].
[2] The contents of this paper overlap with the introduction and the conclusions of our separate conference paper [14].

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 publish-subscribe 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.

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

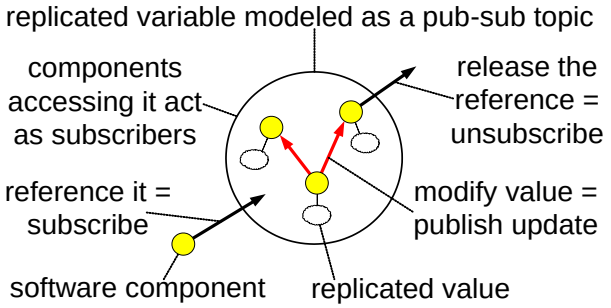

Figure 1. Publish-subscribe services are generalizations of a replicated, writable variable.

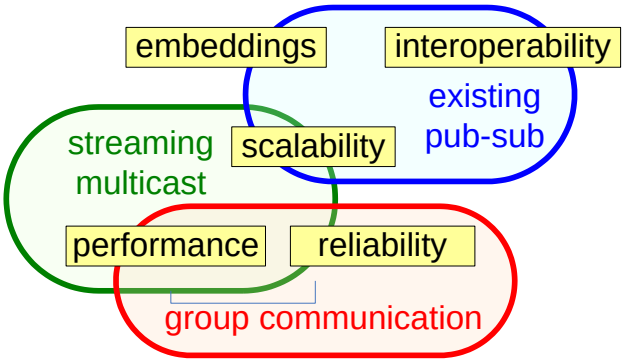

Figure 2. Existing publish-subscribe technologies are insufficient.

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.

In [9], [11], [12] we argue 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 [5] such as Spread [1]. 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 ([9], [11]) that JGroups [2], 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 already 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 [13], 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 ([10]), while group communication systems employ proprietary APIs, e.g. requiring the developer to learn a new and domain-specific vocabulary.

Much research has been dedicated into making the multicast more developer-friendly. One well known approach is the fault-tolerant CORBA [8] 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.

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

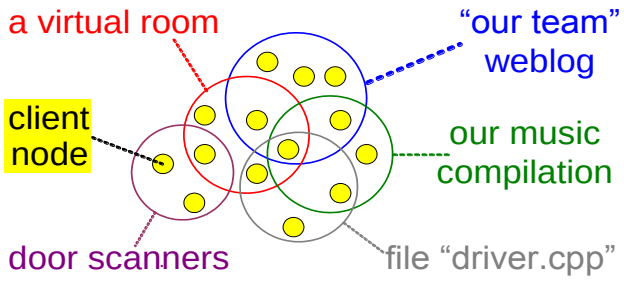

Figure 3. In large systems, there may be large numbers of topics. Different topics might represent different types of services that would require a truly large-scale platform to offer a variety of reliability and security guarantees.

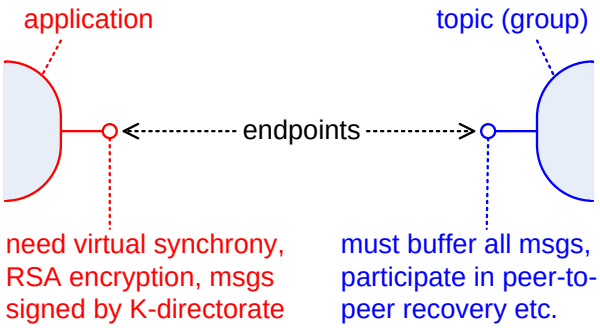

Figure 4. Matching typed topic and application endpoints.

types 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.

2. Evaluation

In the preceding section, we motivated the questions of how the decision to run in a managed runtime environment affects the performance and scalability of a high-performance, scalable, reliable multicast engine. What are the dominant phenomena and issues that arise? What mechanisms can be used to alleviate these problems? What support from the managed runtime could facilitate building such systems?

We base the discussion throughout the remainder of this paper on our experiences while building and evaluating Quicksilver Scalable Multicast (QSM). Unlike QS/2, QSM offers only a high-performance multicast substrate with an ACK-based reliability property similar to [4] 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 of our observations are specific to our system and protocol, we believe that our conclusions 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. Moreover, as we have 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.

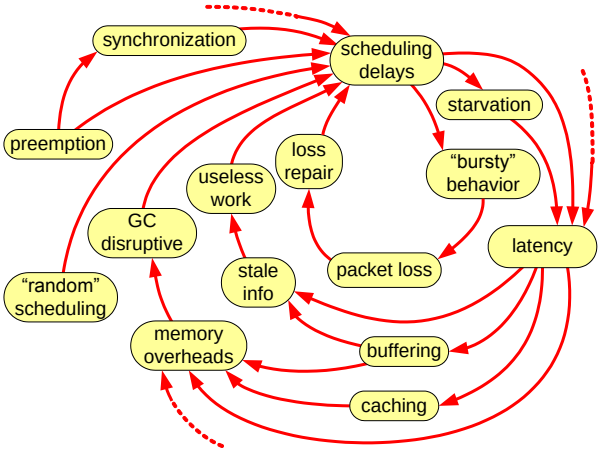

Figure 5. A variety of forces controlling the behavior of the system form a self-reinforcing vicious cycle that has the damaging potential to inflate any temporary perturbances to the level where they start to hurt performance. All these forces are ultimately tied to memory overheads and various sorts of delays or latencies.

For reasons of brevity, we omit the details of the protocol and architecture; additional details, including more discussion of the rationale behind our design choices and a more comprehensive comparison with related work can be found in [9], [11], [12]. The detailed evaluation of the factors affecting the performance and scalability of QSM in a managed environment, and a discussion of techniques that could be used to alleviate them, can be found in [14]. Generalizations and the continuation of this work are described in [10], [13], [15].

There are a number of forces at play in QSM, some of them mutually reinforcing, thus leading to a feedback loop that has a potential to inflate disruptive events caused by losses or by busy applications to the level where they hurt performance (Figure 5). Ultimately, all these phenomena are 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. These delays may come from a variety of sources; for example, various scheduling anomalies, but also the overhead of memory allocation and garbage collection in a busy system. Our experiments demonstrate [14] that even small delays, such as caused by increased overhead of updating data structures, or context switching, can be effectively "inflated" by the protocol, thus resulting in high latencies for critical tasks. Thus, even seemingly low-priority tasks, such as collecting acknowledgements, may be critical and require low latency, typically because of the high memory- and scheduling-related overheads in managed environments.

The key to achieving high performance and very good scalability, at the most general level, lies in a combination of approaches: reducing the "scheduling delays", and reducing the extent to which these small delays are inflated by the protocol, such as by making the protocol more hierarchical, and by avoiding unnecessary synchronization.

To alleviate the "scheduling delays" in QSM, we employed a number of techniques.

First, we eliminated the multithreading entirely, re-implemented the system in a purely event-driven fashion, and built our own scheduler. This allowed us to eliminate the overheads related to context switching, preemption, and the use of synchronization primitives. It also allows us to define our time-sharing and priority processing scheme, which alleviated the priority inversions and reduce delays for important tasks. The scheduling-related changes have tremendously improved the performance of our system.

Scalability, on the other hand, has turned out to be affected mainly by memory-related overheads. Consequently, we eliminated most of the buffering to keep memory footprint small. On the sender side, we re-architected our protocol stack to be entirely "pull"-based, i.e. to postpone the actual creation of messages until "just-in-time" for transmission. On the received side, we employed cooperative caching between topic subscribers. Finally, we found that with thousands of topics, the combined memory footprint of the protocol stack components can be significant; indeed, this factor is the primary obstacle in scaling with the number of topics. Eliminating buffering helped to keep these components light-weight.

Finally, we found that at high data rates, such systems are inherently prone to instabilities. While designing the protocol stack to be pull-based helped to make the system less sensitive by keeping the nodes better synchronized with each other, and avoid redundant work such as acting upon stale forwarding requests, we found that small perturbances, which eventually result in convoy phenomena, are impossible to avoid altogether. Consequently, we designed our system to be convoy-aware and delay-tolerant.

3. 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.

4. 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.

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

[4] 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.

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

[6] 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.

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

[8] S. Maffeis and 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.

[9] K. Ostrowski, K. Birman, and A. Phanishayee. The Power of Indirection: Achieving Multicast Scalability by Mapping Groups to Regional Underlays. Cornell University Technical Report, TR2006-2064, November 2005.

[10] K. Ostrowski and 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.

[11] K. Ostrowski, K. Birman, and A. Phanishayee. QuickSilver Scalable Multicast. Cornell University Technical Report, TR2006-2063, April 2006.

[12] K. Ostrowski and K. Birman. Scalable Group Communication System for Scalable Trust. In Proceedings of the First ACM Workshop on Scalable Trusted Computing (ACM STC 2006), Fairfax, VA, November 2006.

[13] K. Ostrowski, K. Birman, and D. Dolev. Properties Framework and Typed Endpoints for Scalable Group Communication. Cornell University Technical Report, TR2006-2062, July 2006.

[14] K. Ostrowski and K. Birman. Scalable Publish-Subscribe in a Managed Framework. In submission, November 2006.

[15] K. Ostrowski, K. Birman, and D. Dolev. Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. In submission, December 2006.

[16] QuickSilver Scalable Multicast. A distribution and publications: http://www.cs.cornell.edu/projects/quicksilver/QSM/