Bulletin Board: A Scalable and Robust Eventually Consistent Shared Memory over a Peer-to-Peer Overlay*

Vita Bortnikov, Gregory Chockler, Alexey Roytman, Mike Spreitzer
IBM Research

October 30, 2009

Abstract

We present the design and early experience with a completely new implementation of the *Bulletin Board*, a topic-based distributed shared memory service employed by commercial-grade application middleware, to achieve robustness and administrative simplicity with adequate latency and costs at the required throughput and scale. To facilitate scalability, only weak consistency is provided. For robustness and ease of use, the implementation is designed in a fully peer-to-peer fashion leveraging the weakly consistent group communication services provided by a semi-structured overlay network. We discuss issues in providing good (while not perfect) stability and reliability at tolerable cost. We address scalability issues, such as supporting large numbers of processes, large subscription spaces, and complex interest patterns. We also consider comprehensive API instrumentation.

1 Introduction

WebSphere Virtual Enterprise (WVE) is one of the key products in the IBM's middleware portfolio. Its primary function is to manage physical (or virtual) resources in a farm of servers hosting a collection of clustered web applications.

The core component of WVE is a communication substrate, called *Bulletin Board (BB)*, supporting an eventually consistent topic-based shared memory abstraction.It is used by various distributed controllers (which are the "brain" of the system) to support a variety of control loops involving agents and application containers running on each one of the managed machines. The BB is used only for management overhead, of which there is a relatively fixed load, related only qualitatively to the application activity. The BB needs to provide a reliable shared memory with adequate latency and acceptable costs for the system size and throughput needed for that management work.

The implementation of the BB in the early releases of the product was based on a virtually synchronous group communication layer [3] with a limited scaling capacity (from tens to about 100 server processes) and inherent robustness issues. The scaling limits were overcome by a federation technique that exacerbated the robustness problems and introduced significant administrative complexity. Noticing that the BB provides only eventual consistency, and that the product already included an eventually consistent P2P overlay, we supposed it would be easy to re-implement the BB based on that overlay (and possible to extend this work to make a more scalable BB based on a better overlay later). In this paper we report on our experience with the first phase of this effort, which focused on quickly producing a new implementation capable of supporting moderately large server farms including on the order of 1000 of server processes.

Robustness and administrative simplicity were the primary concerns affecting our design choices. Intended dynamicity as well as failures (such as process crashes and network partitions) and flaky processes are common at the scales we target; these should disrupt the system operation to the minimum extent possible, and the recovery should be autonomous involving minimum possible (if any) human intervention.

In addition, WVE supports a dynamic system in which (even in the absence of faults) starts and stops of application server processes, infrastructure processes, and even machines can be common. The BB should therefore be incrementally scalable allowing automated addition, removal, and restart of processes with minimum configuration effort. Also, accurate capacity planning is difficult, and customers sometimes do extreme acceptance tests, so it is also necessary to degrade gracefully under general overload.

The above led us to focus on decentralized approaches with a built-in ability to cope with dynamic changes in

*This work is partially supported by the EU IST project CoMiFin FP7-ICT-225407/2008.

an autonomous fashion (other approaches are surveyed in Section 2). Our implementation is built in a fully peer-to-peer fashion with each process maintaining a portion of the shared memory state pertaining to its topics of interest. It leverages an in-house *semi-structured* overlay network substrate, called the *Service Overlay Network (SON)* [8, 7] to track process subscriptions and writer connectivity (see Section 6).

In order to inform our choice with respect to the update propagation strategy, we studied the topic connectivity patterns exhibited in typical WVE deployments (see Section 5). The study revealed that although each WVE process can write and be subscribed to many topics, the distribution of the number of processes related (either as writers or as subscribers) to a given topic is *bimodal* (i.e., a topic's popularity is either very high or low to modest). That allowed us to design a simple and effective update propagation mechanism, which depending on the number of receivers, employed either SON broadcast or iterative unicast (see Section 6).

As an initial assessment of the impact of the overlay-based design on runtime costs (such as CPU), and update propagation latencies, we studied the performance of the BB implementation in medium-sized settings (see Section 7). Our study was conducted using a stand-alone version of the implementation which was separated from the rest of the WVE components and environment, driven by traces taken from a real WVE environment. Our findings show that the write/notify part of the implementation has tolerable CPU cost (under 10%) for the needed throughput at the needed scale and adequate latencies (almost all under 1 second). Further we noticed that two specific parts of the implementation/usage showed CPU cost per unit of work growing with system size; we mention some ideas on how to address that. Our findings also show that the subscription management part of the implementation needs a fundamental change to fully realize our goals: it exchanges information at too large a granularity. Performance evaluation in larger settings as well as a comprehensive robustness study is a subject of ongoing work.

The rest of this paper is organized as follows: Related work is surveyed in Section 2. Section 3 outlines the most important safety and liveness properties of the BB layer. Section 5 briefly describes typical BB usage pattern in WVE installations. The BB implementation and lessons learned are discussed in Section 6. Performance study is presented in Section 7. Section 8 concludes the paper.

2 Related Work

The initial BB implementation was built on top of a virtually synchronous group communication layer [3], whose implementation requires tight coordination among the processes, and therefore prone to performance and stability problems at large scales.

In contrast, the techniques based on probabilistic shared state maintenance (such as [2, 10]) are scalable and robust. However, they lack sufficient determinism to address the latency and reliability needs of the management control loops employed by the WVE product.

The group communication systems based on IP Multicast (IPMC) [6, 5, 4, 9] off-load the message processing burden from the application layer, and could therefore, be an attractive alternative to the overlay-based design. At present, however, this technology is not yet sufficiently mature to support adequate stability and security in the presence of large number of groups. The recent work [11] looking into improving the IPMC robustness in data centers might lead us to reconsider this approach in the future.

Another known technique for pub/sub communication is based on a carefully designed and managed backbone of servers [1]. Our goal of low-to-no administration would require adding a technique for automatically designing and managing that backbone. We figured it would be easier to meet our needs with a technique that simply uses direct communication (taking the overlay's broadcast as a given primitive) from writer to subscribers. Some of the robustness problems with the first implementation of the BB occur when critical intermediary processes suffer problems that do not affect the writer and subscriber.

3 System Model

We consider an asynchronous distributed system consisting of the set of processes $P = \{p_1, p_2, \ldots, p_n\}$ where each pair of processes is connected by an intermittently reliable communication channel. We do not mean "process" as in Unix or Windows, but rather the fragment of one of those that is a part of the BB's distributed implementation. For brevity we expunge all concerns related to local concurrency and assume the presence of a discrete global clock. This clock is external to the system, and the processes do not have access to it. We model an execution of the system as a series of *events*, each of which happens at a distinct point in time.

A process can be in one of three states: *stopped*, *started*, or *dysfunctional* (corresponding to a running process that is in some degree of local distress). Six kinds of events model the transitions among those states; the initial state is stopped. Each undirected pair of processes can be in one of three connectivity states: *disconnected*, meaning the network is completely unable to deliver service (not even through intermediaries) in some direction between the two processes, *connected*, meaning the two processes

can communicate in a reliable and timely way in both directions, or *partial*, covering the in-between states. Six kinds of events model the transitions among those states; the initial state is connected.

4 The Service Definition

The BB external interface The BB supports *write-subscribe* communication semantics, which is a hybrid of pub/sub and read-write. The function is fundamentally a memory, but the data is pushed out through notifications rather than pulled out through reads; unlike pub/sub, there is no requirement for every subscriber to be notified of each individual update. Formally, we have a collection of processes P that interact with each other through a collection T of *topics*. We fix V to be the set of values, with a distinguished value $\perp$. Process p_i can write a value $v \in V$ to a topic $t \in T$; doing so corresponds to a *write*$(v,t)_i$ event. For subsequent simplicity, before the real events of an execution we insert a set of *write*$(\perp,t)_i$ events for all $t \in T$ and $p_i \in P$.

p_i can register (resp., unregister) its interest to be notified of changes to $t's$ content; doing so corresponds to a *subscribe*$(t)_i$ (resp., *unsubscribe*$(t)_i$) event. The subscribers to a particular topic t get informed of t's content through *notify*$(M,t)_i$ events where M is a total function from P to V such that $M[p_i] \in V$ holds a value written by process p_i. It is positively desired that the BB deliver a notification with $M[p_j] = \perp$ in response to p_j stopping.

BB Correctness Properties Let A be an implementation of the BB, and E be an execution of A. A is correct if for each p_i, p_j, and t, there exists a total function $\mathsf{readsFrom}_{i,j,t}$ from the set of all the $\{$*notify*$(\_,t)_i\}$ events to the set of all the events of the form *write*$(\_,t)_j$ in E such that all of the following properties hold:

Property 1 (Integrity)**.** *Let W = write$(v,t)_j$ and N = notify$(M,t)_i$ be two events in E. If W = $\mathsf{readsFrom}_{i,j,t}(N)$, then, e precedes N in E, and either $M[j] = v$ or $M[j] = \perp$.*

Property 2 (Order)**.** *Let N_1, and N_2 be two notify events in E such that $\mathsf{readsFrom}_{i,j,t}(N_1) = e_1$, $\mathsf{readsFrom}_{i,j,t}(N_2) = e_2$, and $e_1 \neq e_2$. Then, N_1 precedes N_2 iff e_1 precedes e_2 in E.*

Property 3 (Notification Validity)**.** *Let N be a notify event for topic t occurring at process p_i. Then, N is preceded by S = subscribe$(t)_i$ such that p_i neither stops nor unsubscribes from t between S and N.*

The Eventual Inclusion property below has two parts. The first one captures the essence of the eventual consistency semantics. It asserts that in the absence of local and connectivity problems, each value v written to a topic t is eventually delivered to each permanently connected and functional subscriber of t. The second part requires that transient problems have consequences for only a limited time. Beyond this property, it is desired that the BB not deliver a lot more notifications where $M[p_j] = \perp$ than required.

Property 4 (Eventual Inclusion)**.** *Let τ be a time such that all of the following conditions are permanently true after τ in E: (1) p_i is subscribed to t, (2) both p_i and p_j are in the started state, and (3) p_i and p_j are connected. Then, there exists time $\tau' > \tau$ such that the following holds for each event $e \in \{$write$(_,t)_j$, notify$(M,t)_i\}$ occurring after τ':*

- *If e = write$(v,t)_j$, then there exists a notify event N = notify$(M,t)_i$ such that $\mathsf{readsFrom}_{i,j,t}(N) = W'$, and either $W' = W$, or W' follows W at p_j, and*
- *If e = notify$(M,t)_i$, and $\mathsf{readsFrom}(e)$ = write$(v,t)_j$, then $M[j] = v$.*

The next property asserts that if p_j was writing to a topic t, and then, either permanently stopped, or became permanently disconnected from a process p_i, then this fact is eventually reported at each subscriber p_i of t (by delivering a notification with $M[j] = \perp$ at p_i).

Property 5 (Eventual Exclusion)**.** *Suppose that after some point in E the following conditions are permanently true: (1) either p_j is stopped, or p_i and p_j are disconnected, (2) p_i is in the started state, and (3) p_i is subscribed to t. Let N = notify$(M,t)_i$ where $M[j] \neq \perp$, be a notify event occurring at time τ in E. Then, there exists time $\tau' > \tau$ such that both of the following hold:*

- *The event occurring at τ' in E is notify$(M',t)_i$ with $M'[j] = \perp$, and*
- *For all notify events notify$(M'',t)_i$ occurring at p_i after τ', $M''[j] = \perp$.*

5 Workload Characterization

We studied the connectivity and throughput patterns in the way WVE uses the BB, for example WVE configurations constructed in our lab and based on actual customer usage patterns. Note that WVE typically has multiple server

processes on a given server machine, an effect that we expect to increase as the number of cores per machine increases.

WVE makes and cancels subscriptions only as consequences (some of them rather indirect) of (a) process starts and stops and (b) commencement of various categories of application traffic. The establishment of writing relationships has the same correlation with activity. We studied the connectivity and throughput at times well removed from such changes.

At a given time, a given topic can be described by a pair of popularity numbers: the number of subscribing processes and the number of writing processes. We found the distribution of subscribing popularity was bi-modal: each topic was either (a) subscribed by all or almost all of the processes in the system or (b) subscribed by a modest number of processes, at most about the number of processes in an appserver cluster[1] or on a given single machine. We found the same bi-modality in the writing popularity. Both distributions, as well as the joint distribution, were heavily skewed towards low popularity.

We also looked in the other direction, at the numbers of topics to which a given process is related. We found the average process subscribing hundreds to and writing to tens of topics.

The very highly subscribed topics had no throughput at times well removed from process starts and stops. Process starts and stops did provoke writes and write retractions to these topics, mainly by the processes starting and stopping.

6 The BB Implementation

The challenge was to quickly produce a reliable distributed shared memory that needs little to no administration and delivers adequate latency with acceptable costs for the throughput and scale needed to do certain management tasks in WVE.

The BB implementation consists of two parts, the Data Layer and the Interest-Aware Membership (IAM) service, with the former using the latter and both using the SON overlay as shown in Figure 1.

The overlay network maintained by SON is a superposition of the following two graphs: (1) a random graph with nearly constant vertex degree approximating a k-regular random graph [12] and (2) a global deterministic ring. There is a vertex for each process in the system. The random topology establishes, with high probability, that the overlay is strongly connected and has a logarithmic diameter; the ring guarantees eventual connectivity.

[1]These systems typically scale up by adding more appserver clusters of modest size, rather than making large appserver clusters.

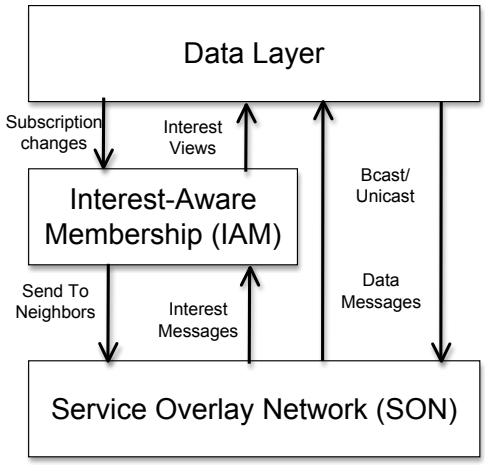

Figure 1: The BB Architecture

The SON overlay supports the node membership, FIFO broadcast (implemented by flooding messages along the overlay edges), and neighbor-to-neighbor communication services as well as assistance in creating unicast TCP connections.

The IAM layer utilizes the SON neighbor-to-neighbor communication services to implement an anti-entropy-style subscription tracking protocol and integrates SON's membership service. The Data Layer leverages SON's broadcast and unicast services for disseminating value updates, and uses IAM for keeping track of the membership and subscriptions. Below, we focus on the implementation of the Data Layer. For brevity we de-emphasize the discussion of the IAM implementation. The details about the SON services and their implementation can be found elsewhere [8, 7].

6.1 Highlights and Lessons Learned

Below, we enumerate and discuss some more interesting aspects of the BB implementation and the lessons learned.

Supporting the BB safety and liveness The BB implementation must behave correctly (as described in Section 3) in the presence of all possible combinations of failures including process stops (whether clean or not) and starts, network connectivity outages (either short-lived or long-term), and message delays. Since unlike virtually synchronous group communication service [3], SON only guarantees best-effort message delivery, and does not guarantees coherence between membership views and message send/receive events, the BB layer had to include

the necessary logic to ensure correctness is never violated, as discussed below.

BB Liveness We use multiple mechanisms to cope with the fact that SON does not provide 100% reliable messaging. Just one is sufficient for reliability (ignoring latency and cost): each write is periodically re-sent (until overwritten, of course), and the period is long (10 minutes). Most of the BB contents are overwritten before this happens, so the periodic re-sending affects few topics. SON broadcast is so reliable that no further steps are taken for those. For unicasts, TCP provides reliability while a connection lasts. Because lower layers do not allow our BB to distinguish clean connection closures from others, we added an explicit TCP connection close handshake to ensure that all outstanding updates have been successfully transmitted. The bad news is that the topics for which those periodic re-sends actually happen are ones for which it is most costly: a few topics to which almost every process writes (albeit just once at startup) and almost every process subscribes. This means the total system cost of the re-sends per unit time grows proportionally to the square of the system size. A better approach would be to eliminate that mechanism in favor of more reliability in the other mechanisms.

BB Safety Each newly issued update operation is tagged with the pair consisting of the sender's name and a monotonically increasing[2] sequence number. The BB stores the latest sequence number along with the latest contents for each location, so this simple tagging scheme is sufficient to guarantee update ordering as long as the set of the shared memory locations can only grow. If the locations can also be eliminated — and worse yet, if they can later be re-created — then a little extra care is needed in order to reclaim storage in this totally decentralized system. It is indeed important to reclaim storage because WVE is intended for a dynamic environment; we need the implementation to eventually forget about canceled subscriptions and locally uninteresting writes.

To distinguish a process restart from a connectivity loss and regain we tag each message with not only the sender's process name but also a unique and monotonically increasing *epoch number* assigned during process startup.

To identify canceled and later recreated subscriptions, we also tag each interest item with a short nonce. A retracted write is implemented by a write carrying a special *tombstone* value. These are deleted after lingering in the system for a configured timeout as judged on local clocks. A write that originated in a remote process is deleted locally after a long time with no receipt of a resend. We do not have to worry about confusing propagation of writes to processes that have already forgotten them because the BB layer only propagates writes one step, from writer to each remote subscriber.

[2]among the writes to one of the memory locations, remember each has a single writer.

The cost of overlay-based broadcast Broadcast by overlay flooding should be used sparingly. The performance cost (in terms of network and CPU usage) is high. This is due to the great redundancy in message transmissions (the total cost is some multiple of k messages at a process with k neighbors) and the facts that forwarding is accomplished at the application level and involves serialization/deserialization in the process. This is exemplified by the relatively high proportion of CPU spent on broadcast of relatively few writes in the experimental study. Circumventing options include more conservative threshold, gossip or tree-based (requires more complex reliability mechanisms) communication pattern instead of complete flooding, avoid deserializing redundant messages.

Selecting the update propagation method The updates are propagated either by SON broadcast or through iterative unicast over direct TCP connections. The choice of whether to use broadcast (as opposed to iterative unicast) is made by testing (at the send time) whether the number of intended receivers exceeds a dynamic quantity called the broadcast threshold. That threshold is derived from the system size by a configured monomial. Iterative unicast always uses fewer messages in the system as a whole, but broadcast can spread the load around better and reduce latency — but only if the system is large. The default configuration of the threshold monomial is thus set to produce a large threshold for small system sizes and to grow slowly with system size.

Flow control and overload protection Flow control really needs to be done. In initial experiments we saw transient overloads at new subscribers to topics with very many writers. To avoid that we added a random delay to the procedure by which a writer reacts to new interest. We also saw that a slow process P could cause memory exhaustion in another process Q that is sending to P (SON's TCP abstraction unconditionally accepts messages for eventual delivery, buffering ones that can not be written immediately.) To avoid that we modified the iterative unicast dissemination to not send a write over a TCP

connection that is still being used for a previous transmission; we plan to go further and add explicit acknowledgments in our protocol.

The SON message broadcast operation is very robust against problems that are localized and/or short-lived, but simple flooding has its limits. If a broadcast originating at A and needing to be delivered at B runs into trouble (that lasts longer than a flooded message lingers in the system) while being transmitted from C to D, A and B will remain unaware of the problem and have no clue if any recovery is needed. We originally expected SON to avoid such problems by excluding flaky processes from the overlay. We later came to realize our expectations would not (and probably could not) be fully met. We hope to replace the BB's use of SON broadcast with anti-entropy, which can be fundamentally robust and stable with limited resource usage — the only lossage during overloads will be in latency. The challenge will be to do anti-entropy for a memory with the large number of locations used for popular BB topics. In the $4\times70+7$ scenario there were about 1,100 such locations.

Message compression We found it advantageous to compress messages in our protocol that are (before compression) longer than fit in one Ethernet packet.

Testing and debugging Testing and debugging is a huge challenge. One helpful technique is to produce a tabular log of the events at the BB API for post-mortem study. The log tables are loaded into a SQL database. SQL statements derive tables of instances of missing notifications and excessive latency. This makes it easy to comprehensively comb through a large amount of logs looking for problems.

7 Performance Evaluation

We have done an organized study of runtime costs and latency distribution, and report on it here. We have also done some informal testing of robustness against wedged processes, with good results; time and space constraints preclude reporting a thorough robustness study here.

topology	core-ms/pair	CPU %	lat. $<$ 1 sec
$2 \times 35 + 5$	1.3	0.47%	100%
$3 \times 47 + 6$	2.1	0.84%	99.998%
$4 \times 52 + 7$	2.9	1.22%	99.9%
$4 \times 70 + 7$	5.2	2.92%	94.7%

Table 1: CPU Cost, Latency Distribution

We created WVE systems of 75, 147, 215, and 287 processes — about a 1:2:3:4 ratio — and studied CPU usage and latency results from a simple scenario that involved starting the appservers one at a time and then running some application load for about 10 minutes (during which time no subscriptions were added or removed). Table 1 summarizes the results. We see the latencies are well controlled, and the CPU cost is growing with system size.

The topology column summarizes the structure of the WVE system, as the product of a number of machines times the number of appservers per machine plus the number of infrastructure processes.

The performance measurements were obtained by using a stand-alone version of the BB implementation to replay the tabular API event traces captured from the runs of the real WVE system. The latency results are from the real WVE system, and CPU measurements were taken from the replays. The CPU usage is reported in units of core-milliseconds per unit of work, to study the growth with system size, and in units of percentage of total CPU power used, to show this is modest. We have four 16-core machines, and used 2, 3, 4, and 4 (respectively) in the four scenarios; since the ratio of number of processes to number of available cores is not constant across these scenarios, you would not hope for constant CPU utilization.

To focus on the costs of the BB implementation, we first did a crippled replay of each topology. The crippled replay included the costs of the overlay, the replay framework, and the BB subscription operations; the BB write operations were no-ops. The cost of the crippled replay was subtracted from the cost of the full cost of the regular replays, leaving just the cost of delivering the updates. The CPU cost of the crippled replay was less than half of the cost of a full replay. We ran three full replays of each topology. The standard deviation of the CPU cost results was about 1% of the average in each case except the smallest, in which it was about 3.5%.

The CPU usage is the quotient of the total amount of CPU work the system did delivering updates during the roughly 10 minutes of light application load divided by a BB usage metric. That metric is the number of (write, matching subscription) pairs in that same interval of time; this metric is well over the numbers of subscriptions and writes, and so is a good summary of how hard the clients are using the BB.

After studying the original traces and trace data from the replay we realized that during the roughly 10 minutes of light application load there is a large amount of refreshing of writes to unscalable connectivity patterns (that is, topics to which almost every process subscribes and almost every process writes). Even though the writes occur only around process startup time, their refreshes continue indefinitely. Lesson: refreshes can escalate a communi-

cation pattern that merely costs $O(N^2)$ memory into one that also costs $O(N^2)$ CPU power.

topology	core-ms/pair	CPU %
$2 \times 35 + 5$	0.78	0.27%
$3 \times 47 + 6$	0.74	0.30%
$4 \times 52 + 7$	0.90	0.38%
$4 \times 70 + 7$	2.11	1.19%

Table 2: CPU cost without refreshes

We then investigated the runtime CPU cost when the refreshes are simply deleted from the implementation; Table 2 summarizes the results. We found the core-ms per pair during the roughly 10 minutes of light application load to be about 1 in the three smaller trials but about 2 in the largest trial. Further study of the traces revealed that during the roughly 10 minutes of light application load the largest topology includes 116 writes whose popularity is above the broadcast threshold and the other topologies have no such writes. Even though the sizes of these writes was short, so that the more efficient of the overlay's two flooding modes was used, the number of messages used in the system for those flood operations was roughly similar to the number of messages used for the non-flooded deliveries. We tweaked up the broadcast threshold, so that no writes during the roughly 10 minutes of light application load were flooded, and saw the average amount of core-ms per pair for the largest topology drop to about 1 and the CPU utilization drop to 0.57%.

8 Conclusions

We presented our experience with improving robustness and reducing administration complexity of the BB communication layer of the WVE middleware. Our implementation is based on the peer-to-peer approach leveraging the services provided by the SON overlay network. The preliminary performance study using the API traces from typical production environments demonstrates viability of our approach in medium-sized settings. The performance evaluation in larger deployments as well as the detailed robustness study are under way.

Our ongoing effort focuses on reducing the update propagation costs using the anti-entropy based message dissemination mechanisms. We also plan to radically improve scalability of our implementation by augmenting it with a dynamically managed hierarchy.

References

[1] M. Astley, J. Auerbach, S. Bhola, G. Buttner, M. Kaplan, K. Miller, J. Robert Saccone, R. Strom, D. C. Sturman, M. J. Ward, and Y. Zhao. Rc23103: Achieving scalability and throughput in a publish/subscribe system. *IBM Research Technical Reports*, 2004.

[2] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of SOSP'87*, pages 123–138, New York, NY, USA, 1987. ACM.

[4] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Commun. ACM*, 39(4):64–70, 1996.

[5] K. Ostrowski, K. Birman, and D. Dolev. Live distributed objects: Enabling the active web. *IEEE Internet Computing*, 11(6):72–78, 2007.

[6] K. Ostrowski, K. Birman, and D. Dolev. Quicksilver scalable multicast (qsm). In *NCA*, pages 9–18, 2008.

[7] C. Tang, R. N. Chang, and E. So. A distributed service management infrastructure for enterprise data centers based on peer-to-peer technology. In *IEEE SCC*, pages 52–59, 2006.

[8] C. Tang, R. N. Chang, and C. Ward. Gocast: Gossip-enhanced overlay multicast for fast and dependable group communication. In *DSN*, pages 140–149, 2005.

[9] R. van Renesse, K. Birman, and S. Maffeis. Horus: a flexible group communication system. *Commun. ACM*, 39(4):76–83, 1996.

[10] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.

[11] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, and Y. Tock. Dr. Multicast: Rx for datacenter communication scalability. In *HotNets VII: Seventh ACM Workshop on Hot Topics in Networks*. ACM, 2008.

[12] N. Wormald. Models of random regular graphs. *Surveys in Combinatorics*, 276:239–298, 1999.