Operation Liveness and Gossip Management in a Dynamic Distributed Atomic Data Service [*]

(Extended Abstract)

Vincent C. Gramoli [†‡] Peter M. Musiał [‡] Alexander A. Shvartsman [‡§]

Abstract

This paper presents performance-oriented refinements and distributed implementation of a reconfigurable linearizable data service for read/write atomic objects. This service is based on the work of Lynch and Shvartsman, and it guarantees consistency under dynamic conditions involving asynchrony, message loss, and node arrivals, departures, and failures. To achieve fault tolerance and availability the service replicates objects at several dynamically, changeable network nodes, to which we refer as *owners*. All-to-all gossip protocol is used to keep replicas up to date and to maintain the list of the owners. However, when gossip is unconstrained and communication bandwidth is limited, network congestion may degrade system's performance. Moreover, we identify a problem where under certain scenarios read/write operations may become delayed or blocked. This paper introduces a more practical algorithm that introduces two refinements. First, we reduce communication cost by restricting the all-to-all gossip pattern to replica owners, based on the local decisions of the participating nodes. In this setting we analyze the latency of read/write operations. Second, we present a solution that allows blocked (or delayed) operations to resume processing and complete successfully. We restate the conditional analysis accordingly. Finally, we engineered a complete distributed system implementing this service and we present empirical results that illustrate the advantages of our approach.

1 Introduction

Linearizable (atomic) shared data services provide building blocks that make the construction of dynamic distributed systems easier. However, combining linearizability with efficiency in practical algorithms for dynamic networks is difficult. A reconfigurable linearizable data service for read/write objects, called RAMBO, was formalized by Lynch and Shvartsman [12]. This service guarantees consistency under dynamic conditions involving asynchrony, message loss, node crashes, and new node arrivals. To achieve fault tolerance and availability, RAMBO replicates objects at several, dynamically changeable, network locations. Gossip among the participants is used to keep replicas up to date, and to maintain configuration information, defined in terms of the membership of the current configuration, and the set of read quorums and the set of write quorums of the members. The algorithm allows all-to-all exchange of information, and when gossip is unconstrained and communication bandwidth is limited, network congestion may degrade the performance of the system. One way to solve this problem is to constrain the gossip patterns. This presents a problem for nodes that do not keep sufficiently up to date. The read or write operations performed by such nodes may become delayed or blocked when the nodes attempt to access data using obsolete quorum configurations that includes nodes that are slow, or that have failed or left the system.

The work overviewed in this abstract pursues methodical development that makes the RAMBO service more practical and results in a substantially more efficient distributed system. We present two refinements aimed at dynamic settings that improve the service in two ways. First we reduce the communication costs of the service by restricting its gossip pattern using the local knowledge of the participating nodes. This preserves the linearizability of the service, but alters its performance. We present a conditional analysis of read/write operation latency in this setting under suitable assumptions. Second, we address the problem of stalled or delayed read and write operations. We allow the nodes of the system to make local decisions (e.g., time-out based) to reset potentially stalled read and write operations transparently to the system users. The local reset enables the operations to use the most up-to-date configuration information. We formally prove correctness of this algorithm, and

[*] This work is supported in part by the NSF Grants 9984778, 9988304, 0121277, and 0311368.

[†] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France.

[‡] Dep. of Comp. Sci. and Engi., University of Connecticut, Storrs, CT 06269, USA.

[§] CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

present an analysis of read/write operation latency. The resulting system reduces the overall communication burden, and improves the liveness of read and write operations. Based on this formal development, we engineered a distributed system implementing the improved Rambo service and we discuss preliminary performance results of our system. Due to space restriction, the proofs are omitted in this extended abstract.

Background. Several approaches have been used to implement consistent data in (static) distributed systems. Starting with the work of Gifford [8] and Thomas [15], many algorithms have used collections of intersecting sets of objects replicas (such as quorums) to solve the consistency problem. Upfal and Wigderson [16] use majority sets of readers and writers to emulate shared memory. Vitányi and Awerbuch [4] use matrices of registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [3] use majorities of processors to implement shared objects in static message passing systems. Extension for limited reconfiguration of quorum systems have also been explored [6, 11].

Virtually synchronous services [5], and group communication services (GCS) in general [1], can also be used to implement consistent data services, e.g., by implementing a global totally ordered broadcast. While the universe of processors in a GCS can evolve, in most implementations, forming a new view takes a substantial time, and client operations are interrupted during view formation. However the dynamic algorithms, such as the algorithm presented in this work and [12, 10, 7], allow reads and writes to make progress during reconfiguration. arbitrarily.

Reconfigurable storage algorithms are finding their way into practical implementations [2, 14]. The new algorithm presented here has the potential of making further impact on system development.

Document structure. In Section 2 we give a brief description of the Rambo algorithm. In Section 3 we present the method that allows us to reduce number of gossip messages, and introduce the resulting complication. In Section 4, we propose algorithmic modification that allows blocked operations to resume. The conditional analysis of the resulting algorithm is presented in Section 5. Experimental results are discussed in Section 6. Final remarks are stated in Section 7.

2 Algorithm

In this section, we present a description of the new algorithm. An overview of the algorithm appears in Figure 1.

System Model and Definitions. We use the message-passing model with asynchronous processors that have unique identifiers (the set of processor identifiers need not be finite). Processors may crash. They communicate via point-to-point asynchronous channels. In normal operation any processor can send a message to any other processor. In safety (atomicity) proofs we do not make *any* assumptions about the length of time it takes for a message to be delivered. To evaluate the performance of the algorithms, we assume that all point-to-point messages are delivered and in bounded time, or not delivered at all.

In order to ensure fault tolerance, data is replicated at several nodes in the network. The key challenge, then, is to maintain the consistency among the replicas, even as the underlying set of replicas may be changing. The algorithm uses *configurations* to maintain consistency, and *reconfiguration* to modify the set of replicas. Formally, a configuration consists of three components: (i) a set of *members*(c), the nodes participating in the configuration, (ii) *read-quorums*(c), a set of quorums, and (iii) *write-quorums*(c), a set of quorums. (A *quorum* is simply a subset of the participating nodes.) We require that the read quorums and write quorums intersect: formally, for every $R \in$ *read-quorums*(c) and $W \in$ *write-quorums*(c), the intersection $R \cap W \neq \emptyset$. During normal operation, there is a single active configuration; during reconfiguration, when the set of replicas is changing, there may be two or more active configurations. Throughout the algorithm, each node maintains a set of *active configurations* – in a set called *configs*. During the reconfiguration operation, a new configuration is added to the set. During the configuration upgrade operation, one or more obsolete configurations may be marked as *removed.* Once a node learns that a configuration has been marked as *removed* it will not be used by that node in any consequent read/write operation.

Read and Write Operations. Read and write operations proceed by accessing the currently active configurations. Each replica maintains a *tag* and a *value* for the data being replicated. Each read or write operation requires two phases: **RW-Phase-1** to *query* the replicas, learning the most up-to-date tag and value, and **RW-Phase-2** to *propagate* the tag and value to the replicas. In a *query* phase, the initiator contacts one read quorum from each active configuration, and remembers the largest tag and its associated value. In a *propagate* phase, read operations and write operations behave differently: a write operation chooses a new tag that is strictly larger than the one discovered in the query phase, and sends the new tag and read() or write(v) operation at node i:

- **RW-Phase-1a:** Node i sends $\langle RW1a, tag, value, configs\rangle$ message to a read quorum of every active configuration. Node i stores the set of active configurations in *op-configs*.
- **RW-Phase-1b:** If node j receives a $\langle RW1a, t, v, c\rangle$ message from i. If t is larger than its own tag, then j updates it tag and value. Node j searches set c for new configurations as well as configurations marked as *removed*, new information is added to *configs*. Next, j sends a $\langle RW1b, tag, value, configs\rangle$ message back to node i.
- **RW-Phase-2a:** If node i receives a $\langle RW1b, t, v, c\rangle$ message from j, it updates its tag, value and the set of active configurations. If c contains configurations previously unknown to i, then the phase is restarted. If it receives $RW1b$ messages from a read quorum of *all* configurations in *op-configs*, then the first phase is complete. Next, i sends $\langle RW2a, t, tag', value', configs\rangle$ message to a write quorum of every active configuration, where tag' and $value'$ depend on whether it is a read or a write operation: in the case of a read, they are just equal to the local *tag* and *value*; in the case of a write, they are a newly chosen tag, and v, the value to write. Node i resets *op-configs* to the set of active configurations.
- **RW-Phase-2b:** If node j receives a $\langle RW2a, t, v, c\rangle$ message from i, then it updates its tag, value and *configs*, and sends to i a $\langle RW2b, tag, value, configs\rangle$ message.
- **RW-Done:** If node i receives a $\langle RW2b, t, v, c\rangle$ message, it adds any new configurations from c to its set of active configurations and to *op-configs*. If it receives an $RW2b$ message from a write quorum of *all* configurations in *op-configs*, then the read or write operation is complete and the tag is marked confirmed. If it is a read operation, node i returns its current value to client.

cfg-upgrade(k) at node i (similar to the phases of read/write operations):

- **UPG-Phase-1a:** Node i chooses an index k, such that k is a configuration identifier that ends the prefix of the sequence of configurations known to i, where there are zero or more configurations up to some ℓ that have been marked as removed, and all configurations with index $\ell <$ and $< k$ are active. Next, i sends a $\langle UPG1a, tag, value\rangle$ message to a read and a write quorum of every active configuration. Node i stores the set of active configurations in *upg-configs*.
- **UPG-Phase-1b:** If node j receives a $\langle UPG1a, t, v\rangle$ message from i, it updates its tag and value, and sends a $\langle UPG1b, t, tag, value\rangle$ message back to node i.
- **UPG-Phase-2a:** If node i receives a $\langle UPG1b, t, tag, value\rangle$ message from j, it updates its tag and value. If it receives $UPG1b$ messages from at least one read and one write quorum of each configuration in *upg-configs*, then the first phase is complete. Then, i sends a $\langle UPG2a, tag, value\rangle$ message to a write quorum of configuration $c(k)$.
- **UPG-Phase-2b:** If node j receives a $\langle UPG2a, t, v\rangle$ message from i, then it updates its tag and value and sends to i a $\langle UPG2b, t, v\rangle$ message.
- **UPG-Done:** If node i receives a $\langle UPG2b, t, v\rangle$ message. If it receives an $UPG2b$ message from a write quorum of configuration $c(k)$, then the upgrade operation is complete. Node i marks all configurations with identifier smaller than k as removed.

join(J) at node i (set J contains the “seed” nodes, if J is empty, then node i is considered as a creator of the service):

- **Join-Phase-1a:** Node i sends $\langle J1b, i\rangle$ messages to each node in J.
- **Join-Phase-1b:** When node j receives $\langle J1b, i\rangle$ from i, then it marks i as a service participant and replies to i with $\langle J2, t, v, configs\rangle$.
- **Join-Phase-2:** Node i receives a $\langle J2, t, v, c\rangle$ message, it updates it tag, value, and the set of configurations. Node i is now active and may actively participate in the service.

Figure 1: Description of the various phases of the read, write, *configuration upgrade*, and *join* protocols. The read, write, and configuration upgrade protocol require two phases (four message delays), while the *join* protocol requires only two message delays.

new value to a write quorum; a read operation simply sends the tag and value discovered in the query phase to a write quorum.

One complication arises: sometime during a phase, a new configuration may become active. In this case, the read or write operation must access the new configuration as well as the old one. In order to accomplish this, read or write operations save the set of currently active configurations, *op-configs*, when a phase begins; a reconfiguration can only add configurations to this set — none are removed during the phase.

Configuration Upgrade. Configurations go through three stages: *proposal*, *installation*, and *upgrade*. First, a configuration is proposed by a recon event. If the proposal is successful, the *Recon* service achieves consensus on the new configuration, and notifies participants with decide events. When every non-failed member of the prior configuration has been notified, the configuration is installed. The configuration is upgraded when every configuration with a smaller index has been removed. Once a configuration has been upgraded, it is responsible for maintaining the data. Upgrades are performed by the configuration upgrade operations. Each upgrade operation requires two phases, a query phase and a propagate phase. During the first phase members of read quorum and write quorum from the old configurations are contacted in order to obtain the latest object information and to notify about the lastly installed configuration. In other words, a **UPG-Phase-1b** phase occurs when “fresh” responses from members of read-quorum and write-quorum of old configuration are collected. In the second phase, the latest object information obtained in the query phase is propagated to the members of the write-quorum of the new configuration. This means that the **UPG-Phase-2b** phase occurs when “fresh” responses from members of the write-quorum of the new configuration are collected. This ensures that the latest object information is propagated to the new configuration, as to ensure objects survivability.

3 Managing gossip

In the Rambo algorithm, the participating network nodes are able to send gossip messages at arbitrary time. This all-to-all gossip is used to propagate information through the system and gather information needed by read and write operations. While this simple protocol is very robust in the face of failures and delays, it places a high communication burden on the underlying network. We assume that the network has a limited bandwidth and we want to reduce the amount of gossip messaging without negatively impacting the performance of individual read and write operations.

We constrain the gossip pattern by assigning either *owner* or *client* role to each participating node. A node considers itself an owner (locally) if it is a member of some active configuration. Otherwise the node considers itself to be a client. It is possible for a node to consider itself a client while another, say more up to date node, may consider the same node an owner. We restrict the periodic gossip pattern by allowing only owners to send gossip messages to other owners.

By allowing the owner nodes to gossip we ensure that all members of active configurations are up to date — for performance reasons it is important for these nodes to have the latest object and configuration information. Of course both the owners and the clients are allowed to perform read and write operations. Because the atomicity of Rambo does not depend on any specific gossip pattern [12, 10, 7], restricting gossip preserves correctness (safety) of Rambo.

The reductions in gossip messaging, stated at a high level, is as follows: if at some time the number of participating nodes is n, and the number of owner nodes is n_o, where $n_o < n$, then the savings in gossip messaging per communication round can be substantial, given that all-to-all gossip requires n^2 messages in the original system, while the new number of gossip messages is only n_o^2. If in a steady-state execution (cf. [12]) there is a constant number of quorum configurations per node, and if the configurations involve quorums of size $O(\sqrt{n})$ (such as those constructed using rows and columns of a square matrix formed by the node identifiers), then the amount of gossip per round becomes linear in n.

The following claim holds since atomicity of Rambo does not depend on any specific gossip pattern [12, 10, 7].

Claim 3.1 *Restricting gossip preserves correctness (safety) of* Rambo*.*

4 Improving Liveness

A read or a write operation performed by a node has two communication phases. In each phase, the node obtains recent information from at least one quorum of each locally known configuration. If a client node is out of date, it will try to contact obsolete configurations that might contain disabled quorums (resp. contain slow nodes), hence blocking the operation forever (resp. delaying it).

Whatever the reason for delayed or blocked operations, if the node performing the operation learns in the meantime of the existence of newer configurations, it may be able to speed-up, or unblock the stalled operation. Consequently we allow for a node to "restart" an operation following a timeout. The restart occurs transparently to the users of the service by resetting certain data structures. Note that in some case, from the standpoint of performance, it may be counterproductive to restart some operations, it is always the case that restarted operations use more up-to-date configurations when such configurations are available.

Given the asynchronous nature of the system, a node can not determine if its operation is delayed or blocked. Hence in a pragmatic implementation of our service the point when the restart occurs is determined by an adaptive time-out on read and write operations. This time-out can be based on the prior performance of these operations. In all cases the correctness of the service is formally preserved.

To avoid the scenario where an operation is blocked because all members of some removed configuration leave or fail, a client node should ensure that its configuration information is updated. This is accomplished as a by-product of client nodes performing read or write operations, or directly by occasional gossip when a client is idle for a long time.

5 Conditional Performance Analysis

A conditional latency analysis of read and write operations in Rambo is presented in [12, 10, 7, 9]. Here, for *restricted gossip* under similar assumptions, we show a complementary conditional latency analysis for operations initiated at nodes that do not have the latest configuration information. For the purpose of latency analysis, we further restrict the sending pattern of the owner nodes: we assume that each node sends messages at certain regular intervals.

In order for the algorithm to make progress in an otherwise asynchronous system we need to make a series of assumptions about the network delays, the connectivity, and the failure patterns. For example, our goal is to model a system that becomes stable at some (unknown) point during the execution. Let α be a (timed) execution and α' a finite prefix of α during which the network may be unreliable and unstable. That is after α' the network is reliable and delivers state messages in a timely fashion. We refer to $\ell time(\alpha')$ as the time of the last event of α'. In particular, we assume that following $\ell time(\alpha')$: (i) all local clocks progress at the same rate, for the sake of analysis and not required by the algorithm, (ii) messages are not lost and are received in at most d time, where d is a constant unknown to the algorithm, (iii) gossip rounds start immediately, where d is the time interval between two consecutive rounds, (iv) all enabled actions are processed with zero time passing on the local clock. In addition we assume that (i) members of any active configuration joined the service before the installation to which they belong to is installed, (ii) the reconfiguration is not too frequent, (iii) configurations remain viable sufficiently long to allow removal, and (iv) configurations leave a footprint, where at least one participant remains alive sufficiently long for all nodes to learn about its configuration.

In RAMBO, each installed configuration is sequentially numbered. We introduce the *obsolescence index* of a node i, denoted as Δ_i, and defined as the difference between the index of the most recently installed configuration in the system and the index of the latest locally known active configuration. Let Δ be the largest obsolescence index of any node. Formally, $\Delta = \max_i\{\Delta_i\}$. We show that under some reasonable assumptions, the latency of read and write operations is degraded at most by an additive term linear in Δ.

Out-of-date clients. We now describe the scenario under which a node can become out of date, which justifies the introduction of the *obsolescence index* of a node. Specifically, the owner nodes gossip only among each other, consequently the client nodes do not receive any updates and are not notified about the configuration changes.

Assume that $c(h)$ and $c(h+1)$ are configurations installed in the system following α', for some $h \geq 1$. Let t represent the time when $c(h)$ is installed. Assuming no failures are present and that there is a $12d$ spacing between subsequent reconfiguration attempts. Then the configuration $c(h+1)$ is installed at least $t' \geq t + 12d$. By definition, a configuration $c(h+1)$ is installed when every non-failed member of configuration $c(h)$ is notified. Members of $c(h)$ consider themselves as owners and will send gossip messages to other owners, also including members of $c(h+1)$. Therefore, at most $t' + d + e$, for some $e \geq 0$ and d time after $c(h+1)$ is installed, each non-failed member of $c(h+1)$ has the up to date configuration information and considers itself as an owner.

Observe that after time t' each member of $c(h)$ is allowed to start configuration upgrade operation whose target is $h+1$. Therefore, $c(h)$ is removed from the system at most at time $t'+5d+e$ (i.e. $4d$ time after $c(h+1)$ is installed). Note, we allow node failures, however the configuration-viability ensures us that there is a sufficiently long interval where communication with at least one quorum is possible until the configuration is marked as removed.

Lastly, notice that client nodes are not notified that new configuration is installed and upgraded, hence the obsolescence index for each client node increases by one. For each installed configuration following α', the described sequence of events is similar.

Bounding operation latency. Now we present the main results of this section, which is an upper bound on the latency of read/write operations. In the following theorem, we use similar timing assumptions as in the analysis performed in [12, 10, 7, 9]. Abandoning all-to-all gossip has its cost: a node may become arbitrarily out of date. To ensure that all read/write operations terminate, we make the assumption that each configuration leaves an everlasting footprint, meaning that once a configuration is installed at least one of its members remains alive until the algorithm terminates. This is the unfortunate cost of reducing gossip. However, in a real system a client node may periodically engage in gossip or perform a read operation, which will prevent it from becoming catastrophically out of date. The decision as to how often this periodic behavior should occur should be left to the developer of the application.

Theorem 5.1 *Let $t > \ell time(\alpha')+e+8d$. Assume i is a node that receives a $\mathsf{join\text{-}ack}_i$ prior to time $t-e-8d$, and that neither fails nor departs in α' until after time $t + (2\lceil\frac{6\Delta'}{5}\rceil + 12)d$, where Δ' is the value of Δ at time t. Then if a read or write operation starts at node i at time t, it completes by time $t + (2\lceil\frac{6\Delta'}{5}\rceil + 12)d$.*

6 Experimental Results

We engineered a complete implementation of the data service on a network of workstations. This development is an example of an approach to software engineering in which formal algorithm design is followed by a methodical translation of the abstract algorithm specification in IOA to distributed Java code using the techniques we developed for this purpose [13]. This approach substantially reduces the human error associated with informal intuitive mapping of specifications to detailed design.

For the purpose of demonstrating the benefits of our refined system, we compare its operation latency to that of the original system. We use a cluster of ten heterogeneous Linux machines connected with a fast 100Mbps network switch where all node instances communicate using TCP communication library.

We illustrate the following scenario. Thirty (virtual) nodes join the service and start with a single configuration with majority quorums. A single client continuously performs read and write operations. The gossip interval is set to one second. We measure the average operation latency while the number of configuration members is increased from one to twenty-nine.

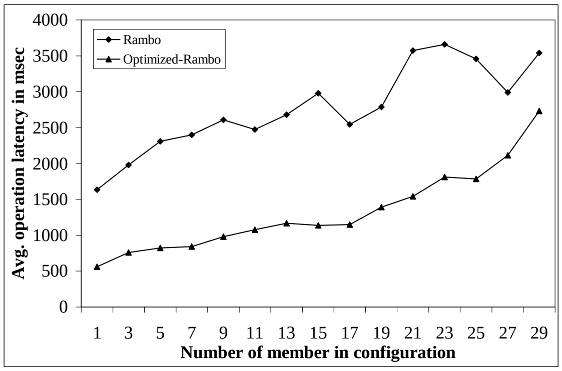

Figure 2: Operation latency vs. configuration size.

Experimental results, presented in the figure show that the operations in the refined system complete on average three times faster than in the original. In both algorithms, the operation latency increases with the number of configuration members, since the number of nodes that must be contacted per communication phase increases. Note that in the new algorithm the client node itself does not send gossip messages, hence the only communication is due to messages generated during read/write operations.

Note that we are able to observe a noticeable improvement despite the small number of participants and modest quorum size. Consequently, we expect the performance benefit to be magnified in systems with a large number of participants.

7 Discussion

Rambo is an atomic memory service designed for highly dynamic networks. In this work we refine the service using a restricted gossip pattern based on local knowledge, and we analyze the operation latency of the service. We improve the liveness of operations by formally refining the algorithms and we present a conditional latency analysis. A methodical evaluation of the performance of the new service is currently in progress. Future plans include further formal optimizations and study of specialized gossip patterns to explore the topology of the underlying network in a variety of settings.

References

[1] Special issue on group communication services. *Communications of the ACM*, 39(4), 1996.

[2] J.R. Albrecht and S. Yasushi. RAMBO for dummies. Technical Report HPL-2005-39, HP Labs, 2005.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.

[4] B. Awerbuch and P. Vitanyi. Atomic shared register access by asynchronous hardware. In *Proc. of 27th IEEE Symposium on Foundations of Computer Science*, pages 233–243, 1986.

[5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, December 1987.

[6] B. Englert and A.A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proc. of International Conference on Distributed Computer Systems*, pages 454–463, 2000.

[7] C. Georgiou, P.M. Musial, and A.A. Shvartsman. Long-lived RAMBO: Trading knowledge for communication. In *Proc. of 11'th Colloquium on Structural Information and Communication Complexity*, pages 185–196. Springer, 2004.

[8] D.K. Gifford. Weighted voting for replicated data. In *Proc. of 7th ACM Symp. on Oper. Sys. Princ.*, pages 150–162, 1979.

[9] S. Gilbert. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. Master's thesis, MIT, August 2003.

[10] S. Gilbert, N. Lynch, and A.A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of International Conference on Dependable Systems and Networks*, pages 259–268, 2003.

[11] N. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of 27th Int-l Symp. on Fault-Tolerant Comp.*, pages 272–281, 1997.

[12] N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

[13] P.M. Musial and A.A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. In *Proc. of 18'th International Parallel and Distributed Symposium — FTPDS WS*, page 208b, 2004.

[14] Y. Saito, S. Frølund, A.C Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS04*, pages 48–58, oct 2004.

[15] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys.*, 4(2):180–209, 1979.

[16] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, 1987.