Sharing Memory with Semi-Byzantine Clients and Faulty Storage Servers

Hagit Attiya
Department of Computer Science, Technion
hagit@cs.technion.ac.il

Amir Bar-Or
Department of Computer Science, Technion
and HP ESDM Laboratory, Haifa
abaror@cs.technion.ac.il

Abstract

This paper presents several fault-tolerant simulations of a single-writer multi-reader regular register in storage systems. One simulation tolerates fail-stop failures of storage servers and require a majority of nonfaulty servers, while the other simulation tolerates Byzantine failures and requires that two-thirds of the servers to be nonfaulty. A construction of Afek et al. [2] is used to mask semi-Byzantine *failures of clients that result in erroneous write operations.*

The simulations are used to derive Paxos algorithms that tolerate semi-Byzantine failures of clients as well as fail-stop or Byzantine failures of storage servers.

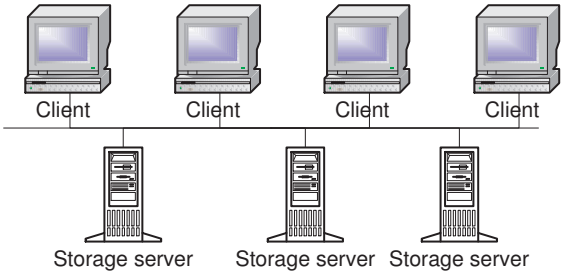

Figure 1. System with clients and storage servers.

1. Introduction

Fault-tolerance crucially depends on some sort of *redundancy* to cover-up for and replace failed components. Traditionally, redundancy is achieved by superfluous processing elements. In modern distributed systems, *storage* provides an alternative source of redundancy for fault-tolerance. Such systems include *client processes* connected to n *storage servers*, which can range from simple disks, to active disks [18, 1] and object stores [19]. (See Figure 1.) The systems are inherently *asynchronous*, since clients' and servers' speed varies and message delay is unpredictable.

Most methodologies for exploiting redundancy assume faulty behavior is "well-behaved", namely, failed components stop operating. Unfortunately, this *fail-stop* fault model does not capture faults caused by malicious attacks, software errors, or corrupted storage. For example, it is possible that a process crashing during a block-write to the disk will corrupt the data on this block. To play it safe, it is best to assume that both clients and storage servers may suffer *Byzantine* failures and send incorrect information [13].

This paper proposes a systematic approach to algorithm design in storage systems, by simulating *shared memory* accessed by simple read and write operations.[1] This is a well-known methodology for obtaining simple solutions to distributed problems [3].

Clients accessing the shared-memory are running application programs that attempt to perform a shared task. For such applications, it is reasonable to assume that programs are well-debugged and do not suffer malicious, arbitrary failures. Yet, corrupt data may be written to remote shared storage. For example, a client may fail while executing an update operation leaving a corrupted disk block, network switches and storage hardware can fail in various ways, and messages from the client to the storage server may be lost or re-ordered. Such *semi-Byzantine* failures can be modeled following the *faulty shared-memory* model [2], in which failures are atomic write operations of arbitrary data.

We show how to mask semi-Byzantine clients by employing a simulation of an *atomic* single-writer multi-reader register using $20f + 8$ atomic SWMR registers [2].

To reduce the simulation cost, the simulations on a

[1] Stronger operations, such as compare&swap, cannot be simulated since they would allow to solve consensus, which is impossible in our model, unless sophisticated storage devices are used [6]. Our assumptions about storage servers are basic and satisfied by even the weakest servers, e.g., simple disks.

message-passing system with storage servers support only *single-writer multi-reader (SWMR) regular* registers [11]. Although regular registers provide weaker guarantees than the more-familiar *atomic* registers, they often suffice for the correctness of shared-memory algorithms. Two notable examples are the Bakery algorithm for mutual exclusion [10], and the Paxos algorithm for state-machine replication [8].

Fail-stop storage servers can be integrated into this scheme with the shared-memory simulation of Attiya et al. [3]. This immediately imply a simulation of SWMR regular register with semi-Byzantine clients and fail-stop storage servers, assuming that at least a majority of the storage servers do not fail. An execution of a read or a write operation requires $O(n)$ messages and $O(f)$ steps.

To handle Byzantine failures at the servers, we show how to simulate a regular SWMR register in the presence of Byzantine storage servers. The simulation requires at least $3t + 1$ storage servers, where t is the number of storage servers that may suffer Byzantine failures. A drawback of our construction is the possibility of non-terminating reads; however, this happens only when the number of overlapping writes is unbounded. In *leader-oriented* algorithms, eventually there is a single leader, which is the only process performing read and write operations; in such algorithms, the read operations of our construction terminate.

Combining with our construction for semi-Byzantine clients, we get a simulation of SWMR regular register with semi-Byzantine clients and Byzantine storage servers.

Figure 2 presents a high-level view of our constructions, which appear in Section 3.

There are other shared-memory simulations in the presence of Byzantine failures. Malkhi and Reiter [15] introduce two classes of quorum systems: *Masking* quorum systems tolerate t Byzantine failures with $4t + 1$ servers, while *dissemination* quorum systems assume self-verifying data and require only $3t + 1$ servers. Martin et al. [16] describe SBQ-L, a simulation of a multi-writer multi-reader atomic register in an asynchronous message-passing system with Byzantine servers. SBQ-L requires $3t + 1$ servers where t is the number of Byzantine failures; they show this is optimal for providing safe semantics. SBQ-L use an unusual communication pattern that requires that servers maintain subscription list of read operations and send updated values to the readers until a read operation commit. Work in progress by Abraham et al. [9] is studying wait-free constructions for $3t + 1$ processes, which tolerate t arbitrary failures.

Our construction is more scalable than Martin et al. [16], since storage servers need not keep information about readers. Unlike Malkhi and Reiter [15], our construction requires only $3t + 1$ servers without any assumptions about the data . In the implementation of Martin et al. [16], read operations have an unbounded message complexity when the number of overlapping writes is unbounded.

To illustrate the benefits of our approach, we consider the quintessential leader-oriented algorithm: the *Paxos* algorithm for state machine replication [12]. As in the consensus problem, processes start with independent inputs and have to *agree* on the same output value, which must be the input of some process. Since consensus is unsolvable in an asynchronous system, even with one faulty process [7], the Paxos algorithm makes progress only when the system is stable with a unique leader (but it never produces erroneous results).

The original Paxos algorithms [12, 17], for asynchronous message-passing systems, tolerate fail-stop failures of a minority of the processes (which is optimal). *Disk Paxos* [8], motivated by systems with network attached disks, was designed for an asynchronous system with fail-stop storage servers and clients. The algorithm makes progress if one non-faulty client can read and write from a majority of the storage servers. A key step in the derivation of disk Paxos is a *shared-memory* Paxos algorithm, designed for clients accessing ordinary single-writer multi-reader regular registers [8].

By applying our simulations to the shared-memory Paxos algorithm (described in Appendix A), we derive new Paxos algorithms that tolerate semi-Byzantine clients and fail-stop or Byzantine storage servers. (These results appear in Section 4.) Their cost multiplies the cost of disk Paxos by a function of f (the number of faulty writes to be tolerated). For fail-stop failures, the number of servers has to be at least $2t + 1$, as for disk Paxos; $3t + 1$ servers are needed when failures can be Byzantine, which is optimal.

Byzantine Paxos [5] is part of a replicated state machine. It provides safety and liveness if where there are $3t + 1$ processes and at most t of the processes suffer Byzantine failures; the algorithm sends $O(n^2)$ messages. The minimal number of processes that is required to provide safety with t Byzantine failures is $3t + 1$ [4].

2. The System Model

Our simulations implement a single-writer multi-reader (SWMR) regular register [11]. This is a well-known shared memory object accessible by read and write operations; a single process may perform write operations to the register, but all processes can perform read operations. It is guaranteed that a read(R) operation from a SWMR regular register R returns the value v of the latest preceding write(R,v) operation or the value of some overlapping write(R,v) operation.

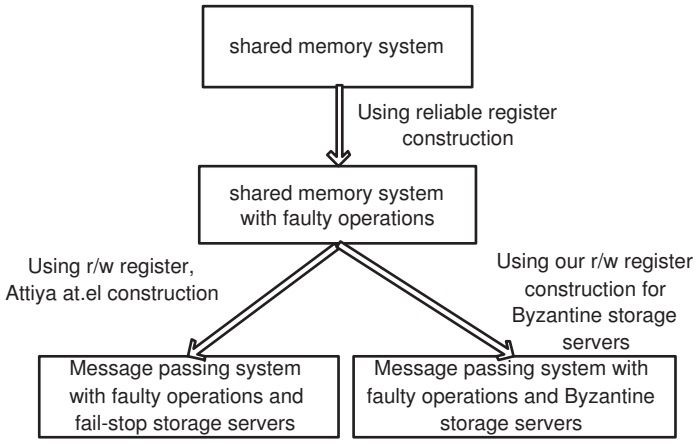

Figure 2. Road map for the constructions presented in the paper.

A *semi-Byzantine* client fails either by stopping or by executing some memory access operations incorrectly. A register written by a *faulty* operation may contain corrupted data. Memory objects are protected by *access control lists (ACL)* [14], which guarantee that only a single client has a write privilege while other clients only have a read privilege. A faulty register can thus be modeled as a non-faulty register suffering from erroneous write operations, following the *faulty shared-memory model* [2].

A sequence of write and read operations is f*-faulty* if it can be made legal (for a regular register) by inserting f write operations. This is equivalent to assuming that the sequence contains at most f erroneous write operations.

An algorithm is an f-reliable implementation of a regular register if any f-faulty execution of the algorithm returns the expected results. In fact, we require the algorithm to tolerate f erroneous write operations in each *invocation* of an operation. Notice that this is a much weaker than requiring that the total number of faulty operations in an execution is at most f.

Our target system is an asynchronous system where some number of clients and n storage servers are connected by a complete network (Figure 1). Storage is accessed by sending a request to a storage server (or all servers); after performing the access operation, the server sends a response (an acknowledgment, possibly with a value) to the requesting client. The network is reliable: every message is eventually delivered.

We consider both simple *fail-stop* failures of storage servers, where a failed server stops taking steps completely, and *Byzantine* failures of storage servers, where a failed server may behave arbitrarily, possibly sending incorrect and inconsistent data. In both cases, t denotes the maximum number of failures that may occur during the execution of the system.

3. Simulating a Single-Writer Multi-Reader Regular Register

3.1. Handling Semi-Byzantine Clients

As mentioned before, we employ a simulation of an f-reliable SWMR atomic register in an asynchronous shared memory system with semi-Byzantine clients [2].

Theorem 3.1 *There is an f-reliable simulation of a single-writer multi-reader atomic register from $20f + 8$ atomic SWMR registers.*

3.2. Handling Fail-Stop Storage Servers

The simulation of Attiya et al. [3] is used to deploy the construction of Theorem 3.1 in a system with fail-stop storage servers and semi-Byzantine clients.

This construction goes as follows: In order to write a value, a process sends the value together with a version number to all storage servers and waits to receive acknowledgments from a majority of them. To read, a process requests all storage servers, waits to receive answers from a majority of them and takes the value with the largest version number. Since the two majorities include at least one single common storage server, a read must obtain the last version for which a write has completed.

Algorithm 1 Simulation of a SWMR regular register with t Byzantine storage servers

```
function write(x)
W1: send ⟨STORE,x⟩ to all servers
W2: loop
W3:     receive ⟨ack⟩ from server S
W4: until ⟨ack⟩ is received from n − t servers

function read()
R1: loop
R2:     send (READ) to all servers
R3:     loop
R4:         answer[S] = receive ⟨VALUE,x⟩
                from server S
R5:     until received from n − t servers
R6: until (one value appears n − t times in answer[])
```

We combine this simulation with the algorithm of Theorem 3.1. Every write or read operation of the $20f + 8$ SWMR regular registers, requires $2n$ messages, where n is the number of storage servers. Thus, the message complexity is $40nf + 8n$ and the time complexity is $20f + 8$. Write and read messages can be aggregated in order to reduce the message complexity to $2n$. This gives the following theorem:

Theorem 3.2 *There exists an simulation of single-writer multi-reader regular register in a system with semi-Byzantine clients and fail-stop storage servers, assuming that a majority of the storage servers do not fail.*

3.3. Handling Byzantine Storage Servers

We first show how to simulate a SWMR regular register in an asynchronous message-passing system with Byzantine storage servers. Disks have their own mechanism's to ensure data correctness. Yet, as disk technology advances and disks become smarter, the possibilities for mal-behavior increases, and Byzantine failures should be considered for disks as well.

Our simulation assumes that $n > 3t$, where t is a bound on the number of faulty storage servers. The construction always guarantee safety; termination is guaranteed if a read operation overlaps a finite number of write operations. In the next section, we show that this construction suffices for the Paxos algorithm.

Algorithm 1 describes the simulation.

Lemma 3.3 *(a) A read operation in Algorithm 1 returns the value of the latest preceding or an overlapping write operation.*
(b) A read operation terminates if there is a finite number of overlapping write operations.

Proof: (a) A write operation W completes after receiving ack from $n - t$ servers; at least $n - 2t$ of these servers are non-faulty. Consider a read operation R that starts after W completes; R completes only if it receives messages with the same value from $n - t$ servers. Since $n > 3t$ the two sets intersect, thus the returned value is the value written by W or a later operation.

(b) Let W' be the last write operation that overlaps a read operation R. Since the network is reliable, eventually all non-faulty servers receive the message of W. Eventually, R receives $n - t$ messages from all non-faulty processes that were sent after they received the value written by W. At this point, R has $n - t$ copies of the same value and terminates. ■

A write operation always terminates since it only waits for $n - t$ servers, and its message complexity is $2n$. If there are no concurrent writes, Algorithm 1 sends $2n$ messages for each read operation. Thus the message complexity of the algorithm when there are no concurrent writes is $O(n)$ and its time complexity is $O(1)$.

Theorem 3.4 *Algorithm 1 is a simulation of single-writer multi-reader regular register in a system with $n > 3t$ storage servers, where t is the maximal number of Byzantine failures.*

Combining this simulation with the construction of Theorem 3.1 gives the following theorem:

Theorem 3.5 *There is a simulation of single-writer multi-reader regular register in a system with semi-Byzantine clients and $n > 3t$ storage servers, where t is the number of Byzantine storage servers.*

4. Application: Paxos Algorithms

This section applies our simulations to Paxos algorithms [12]. At the core of Paxos lies a consensus algorithm called *Synod*. The Synod algorithm always guarantees safety, but makes progress only if the system is stable enough to elect a single leader. Gafni and Lamport [8] describe *shared-memory Paxos*, a version of the algorithm using single-writer, multi-reader regular registers (see Appendix A). We extend shared-memory Paxos to handle failures of clients and storage servers.

4.1. Paxos with Fail-Stop Storage Servers and Semi-Byzantine Clients

This algorithm extends shared-memory Paxos by using the simulation of a regular SWMR register on an asynchronous message passing system with semi-Byzantine clients and fail-stop storage servers (Theorem 3.2). The resulting algorithm tolerates f faulty operations executed in a single phase of the client, since a client writes once in a single round of the Synod algorithm.

The new algorithm has a better resistance to failures than disk Paxos. Consider, for example, the case where the leader client of disk Paxos fails during a write operation to its register. The value of the register is unpredictable, and in the worst case, may contain the highest round number of all registers and possibly, an invalid value as the suggested consensus value. When a new leader is elected, it chooses the invalid value as the consensus value, since the register contains the highest phase number, violating the validity property.

The message and space complexity of the resulting algorithm is $20f + 8$ times the complexity of shared-memory Paxos. Figure 3 describes the evolution of disk Paxos with semi-Byzantine clients from shared-memory Paxos. In shared memory Paxos, each process writes only to its dedicated register. In the first evolution step, each regular register is replaced with a construction of a reliable regular register from a set of faulty shared registers; this provides the Paxos algorithm with resilience to semi-Byzantine clients. The second evolution step, shifts the algorithm to a message-passing system using a SWMR register abstraction according to the failure model of the storage servers.

If we forbid a client process to continue after its first faulty operation, then it suffices to use 28 registers (f=1) in order to construct a reliable register. This can be relevant in a system where a failing client always stop, but the result of its last write operation is unpredictable. In this case, our algorithm increases the space and time complexity of disk Paxos algorithm only by a *constant* multiplicative factor.

4.2. Paxos with Byzantine Storage Servers and Semi-Byzantine Clients

We can extend shared-memory Paxos by applying the simulation of a regular SWMR register in an asynchronous message-passing system with Byzantine storage servers and semi-Byzantine clients (Theorem 3.4).

The Paxos algorithm may not terminate when executed by several processes; it is designed for the best-case in which only a single process executes the algorithm. A leader election algorithm is executed in a different thread and is expected to select a single leader. Only prospective leaders write in the Paxos algorithm; once a leader is selected, all participants except the chosen leader stop executing the Paxos algorithm.

This property of the Paxos algorithm allows to mask the drawback of non-terminating reads: Eventually all other participants terminate the Paxos algorithm and stop executing write operations; this allows the read operations of the chosen leader to terminate, as there are no concurrent writes.

The resulting algorithm tolerates f faulty operations of a client and t Byzantine storage servers.

The simulation masking semi-Byzantine clients does not handle Byzantine storage servers, since all registers share all of the disks. On the other hand, Algorithm 1 does not tolerate Byzantine clients. Martin et al. [16] describe how to handle Byzantine client failures, but this extension requires that the storage servers perform a complex algorithm. It requires $O(n^2)$ messages between servers for each write operation of the client, and its message complexity is not bounded when an unbounded number of write operations overlap a read operation.

The space requirements (per disk) of the algorithm tolerating Byzantine failures are the same as for fail-stop storage servers. However, the algorithm requires at least $3t + 1$ storage servers, where t is the number of failure, instead of only $2t + 1$ fail-stop servers. Since in the Paxos algorithm, each client writes only once in a phase, the time complexity remains as in the algorithm from the previous subsection, which tolerate semi-Byzantine clients and fail-stop storage servers

5. Conclusions and Future Work

We present a shared-memory simulation over storage systems, that can sustain Byzantine failures of disk writes, that is, corrupted write operations, as well as fail-stop or Byzantine failures of storage servers.

Several research question remain open. First, it might be possible to find a simulation of a *regular* SWMR register in the presence of faulty write operations, which uses fewer lower-level registers than the implementation of Afek et al. [2]. Second, it would be interesting to find a *terminating* simulation of single-writer multi-reader register with $n > 3t$ storage servers, where t is the number of Byzantine storage servers. The simulation should not employ server-side information about clients, as is done in the simulation of Martin et al. [16].

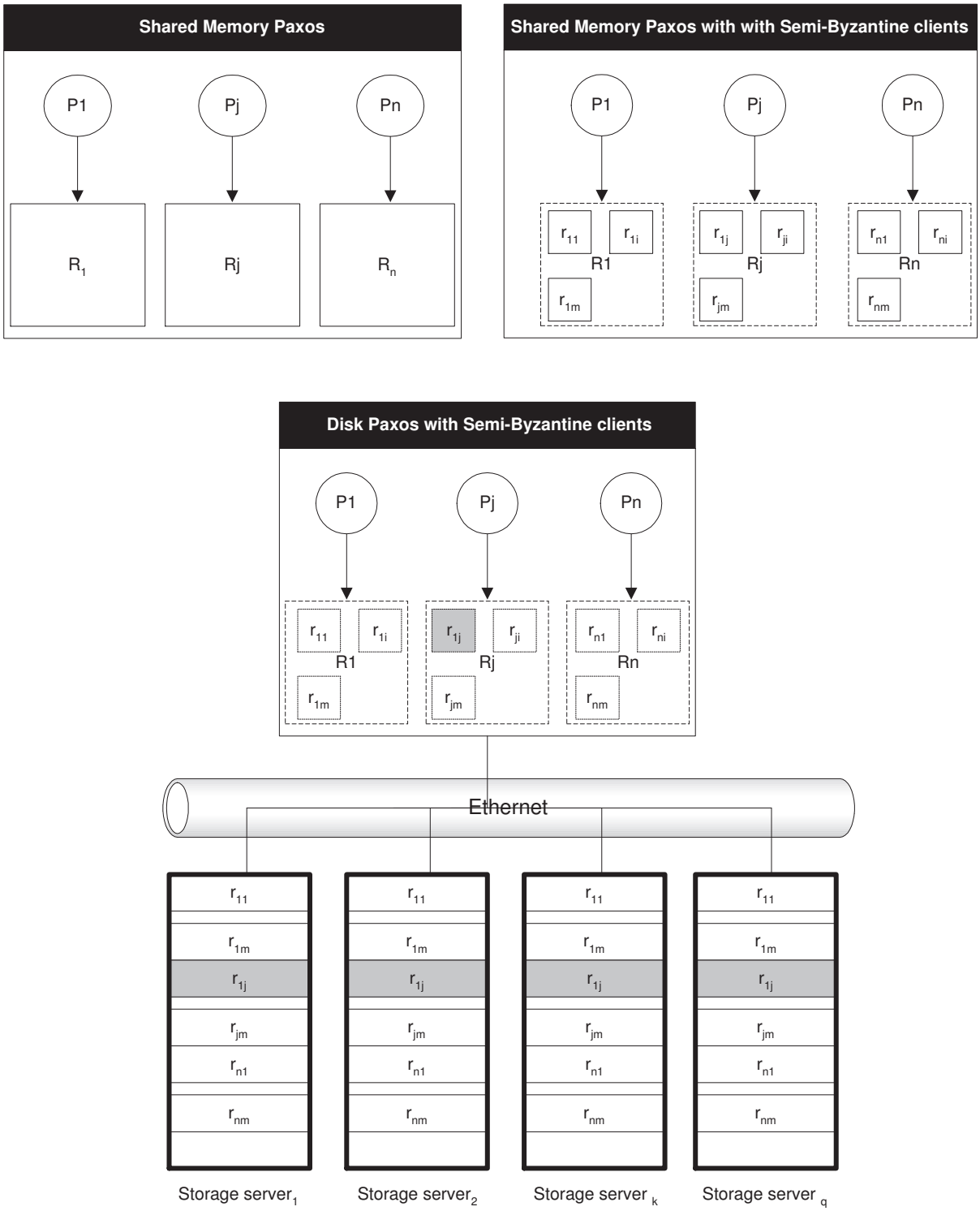

Figure 3. Outline of the algorithm.

Castro and Liskov [5] present a Paxos algorithm that tolerates Byzantine processes failures, assuming there are at least $3t + 1$ processes. Our algorithm proceeds even if only one client is alive, since we address only semi-Byzantine failures, and replace process redundancy with storage redundancy.

Chockler and Malkhi [6] presented Active Disk Paxos that supports unlimited number of processes with fail-stop failures. The algorithm requires one strong memory object (*read-modify-write*) in every disk. Like the shared-memory simulation of Attiya et al. [3], they filter non-responsive storage servers by waiting for responses only from a majority of the processes. It would be interesting to exploit active disks to handle Byzantine failures, e.g., by implementing read and write locks, to avoid the requirement on the ratio of faulty processes.

Other research opportunities are suggested by emerging architectures, such as ObjectStore, in which storage servers employ security protocol at objects granularity, rather per disk. This opens up possibilities of simplifying and strengthening many algorithms, including Byzantine Paxos. These algorithms using identification and authentication protocols can now be deployed in a shared memory model with secured shared register.

References

[1] A. Acharya, M. Uysal, and J. H. Saltz. Active disks: Programming model, algorithms and evaluation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, 1998.

[2] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6):1231–1274, 1995.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(2):824–840, 1985.

[5] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[6] G. V. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, July 2002.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[8] E. Gafni and L. Lamport. Disk paxos. In *International Symposium on Distributed Computing*, pages 330–344, 2000.

[9] I. K. Ittai Abraham, Gregory Chockler and D. Malkhi. Optimal resilience wait-free storage from byzantine components. 2003.

[10] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[11] L. Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[12] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[13] L. Lamport and R. E. S. an Marshall C. Pease. The byzantine generals problem. *ACM Trans. on Prog. Lang. and Sys.*, 4(3):382–401, 1982.

[14] D. Malkhi, M. Merritt, M. K. Reiter, and G. Taubenfeld. Objects shared by byzantine processes. In *DISC*, pages 345–359, 2000.

[15] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, pages 569–578, May 1997.

[16] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. Technical Report TR-02-38, University of Texas at Austin, Department of Computer Sciences, August 2002.

[17] B. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly available distributed systems. In *Seventh ACM Symposium on Principles of Distributed Computing*, 1988.

[18] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[19] SNIA. Storage networking industry association. http://www.snia.org/.

A. Shared-memory Paxos

We state present the code and discuss the correctness of the shared-memory Paxos, which originally appeared in [8].

The input value of process p is denoted $input_p$. A process repeatedly executes rounds, numbered by positive increasing integers. Different processes use different round numbers, e.g., process j could use round numbers $j, j + N, j + 2N, \ldots$.

Each rounds consists of two phases; in the first phase, the process chooses a value v and in the second phase the process tries to commit it. The algorithm maintains consistent views of the chosen value. Thus if a process p has chosen a value and wrote it in phase 2, then all processes in later stages will choose the same value and will try to commit it even if p fails before committing.

The regular register R_p of process p has the following fields:

Algorithm 2 Shared Memory Paxos

Set max_round := 0;
Set chosen_val := NotAnInput

Phase 1:
write r to R_p.*cRound*
for every process q do
    read R_q
    if R_q.*cRound* $> r$ then abort
    if R_q.*val* $\neq \perp$ then
        if $R_q.varlRound >$ *max_round* then
            max_round := R_q.*lRound*
            chosen_val := R_q.*val*
    if *chosen_val* $= \perp$ then
        chosen_val := *input*$_p$

Phase 2:
write $\langle r, r,$ *chosen_val*$\rangle$ to R_p
for every process q do
    read R_q.*cRound*
    if R_q.*cRound* $> r$ then abort
commit *chosen_val*

cRound The current round number.

lRound The largest round number for which p reached phase 2, initially 0.

val The value that p tried to commit in round *lRound*, initially a special value $\perp$.

Algorithm 2 presents the pseudo-code of shared-memory Paxos. We next outline the basic properties of the algorithm.

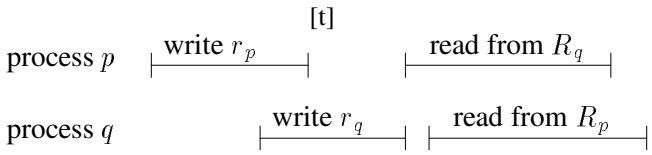

Figure 4. Proof of agreement in Paxos algorithm.

If p is the only process executing the algorithm, then p eventually reaches a round r that is greater than the *cRound* value of all other processes. When this happens, p does not abort round r and commits to a value at the end of phase 2.

The committed value is either process p's input or a value some other process's register. Simple induction shows this register contains only input values, which implies the validity of the algorithm.

To argue agreement, assume that process p commits v_p in round r_p and process q commits v_q in round r_q; Since each process has different round numbers, $r_p \neq r_q$; without loss of generality, $r_p < r_q$. Note that the read of process p in phase 2 of round r_p does not return the value r_q (otherwise, p aborts and does not commit). Thus, the read of p does not follow the write of q, which implies that the read of q follows the write of p (see Figure4). Thus, q must read r_p from R_p and it does not decide on v_q.