A Correctness Proof for a Byzantine-Fault-Tolerant Read/Write Atomic Memory with Dynamic Replica Membership

Rodrigo Rodrigues and Barbara Liskov
MIT Computer Science and Artificial Intelligence Laboratory
rodrigo,liskov@csail.mit.edu

Abstract

We prove correctness of a Byzantine-fault-tolerant replication algorithm for a read/write atomic memory that supports a dynamic replica set.

1 Introduction

This paper describes a correctness proof for a read/write atomic memory that tolerates Byzantine faults, and allows reconfigurations in parallel with the execution of normal requests, i.e., the set of replicas for a particular data item may change throughout the execution of the algorithm.

We present a trace-based proof of the correctness of the algorithm, and correctness conditions that are more complex than usual due to the presence of arbitrary faults and a dynamic membership of the system.

The remainder of the paper is organized as follows. Section 2 presents the system model. Section 3 describes the main algorithm. Section 4 presents the correctness condition. Section 5 proves the correctness of the system, and we conclude in Section 6.

2 Model

The goal of our algorithm is to provide a Byzantine-fault-tolerant implementation of a *read/write atomic object* [2]. Our atomic object implementation uses replication to enable concurrent sharing of the variable by many *clients* in a distributed system. It ensures atomicity when clients do not fail. The algorithm provides certain guarantees in the presence of Byzantine-faulty clients, namely that after the faulty client stops the system eventually recovers and start behaving correctly again. However the proof of this property is outside the scope of this paper, so we will focus on the correct client case.

We consider a universe of servers, S, that may participate in the system at different time intervals, and an arbitrary number of (possibly distinct) clients. The algorithm does not require knowledge of the possible set of participants, S.

We consider a problem in which the clients perform read and write operations on a variable x that is replicated on a subset of the servers. This subset may change throughout the execution.

We assume an asynchronous distributed system where nodes are connected by a network that may fail to deliver messages, delay them, duplicate them, deliver them out of order, and there are no known bounds on message delays or on the time for nodes to execute operations.

We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily except for the following restrictions. Nodes can use unforgeable digital signatures to authenticate communication; and they can use collision-resistant hash functions. These assumptions are defined in more detail next.

- **Unforgeable signatures:** Any node, x, can authenticate messages it sends by signing them. We denote a message m signed by x as $< m >_{\sigma_x}$. And (with high probability) no node other than x can send $< m >_{\sigma_x}$ (either directly or as part of another message) on the multicast channel for any value of m, unless it is just repeating a message that has been sent before.

- **Collision-resistant hash functions** Any node can compute a digest $D(m)$ of a message m such that (with high probability) it is impossible to find two distinct messages m and m' such that $D(m) \neq D(m')$.

These assumptions are probabilistic but there exist signature schemes (e.g., [4]) and hash functions (e.g, [1]) for which they are believed to hold with very high probability. Therefore, we will assume that they hold with probability one in the rest of the paper.

2.1 Trusted Configurations

We assume there is a trusted source of configuration information. This source periodically produces a description of the current configuration (i.e., the current set of replicas for variable x), authenticates it using public key cryptography and attempts to disseminate this information. Associated with each configuration is a sequential epoch number, e. Thus, configuration description messages have the following format:

$<$NEW-CONFIG, $c, e >_{\sigma_{trusted\ source}}$, where $c \in 2^C, e \in \mathcal{N}$

Note that the configuration description must not only include the set of replicas, but also their respective public keys, so that communication with those replicas can be authenticated. Therefore, we assume that $C = S \times P$, where P is the set of possible public keys.

Nodes joining the system for the first time must obtain the public key for the trusted source using some out-of-band mechanism. We also assume that incoming nodes can contact the trusted source to obtain the current configuration (or validate a configuration obtained from any other node in the system, as described next).

We present a possible implementation of such a source using a Byzantine agreement protocol in a separate publication [5]. The fact that we have designed and implemented such a source in a Byzantine-fault-tolerant fashion demonstrates the practicality of these assumptions.

Note that we do not assume any synchrony bounds in the process of disseminating of the current configuration: there may be an arbitrary delay between the instant when the new configuration is produced and the instant when node i receives it. However, we do assume that the configuration description is authenticated in such a way that any node possessing such a description can forward it to any other node in the system.

We assume that data is self-verifying, i.e., information that only clients can create and to which clients can detect any attempted modification by a faulty node (e.g., data can be digitally signed by a set of authorized writers whose public keys are known). The self-verifying data includes a version number, or timestamp, that is produced by the client, and cannot be forged by faulty replicas (if the data is signed, this signature must also cover the timestamp).

3 Algorithm

The algorithm uses the following variables at each node n, either a client or a server:

$epoch_n \in \mathcal{N}$, initially 0
$config_n \in 2^C$, initially c_0
$previous\_config_n \in 2^C$, initially $\{\}$.

Additionally, replicas maintain the following information about the latest value of the replicated variable, x:

$val_n \in V$, initially v_0
$ts_n \in \mathcal{N}$, initially 0

Timestamps are assigned by a client to each replica of the variable x, when the client writes the replica. Our protocols require that different clients choose different timestamps. This can be achieved by appending a client identifier to an integer timestamp.

Operations must be completed in quorums. These can be arbitrary constructions for dissemination quorum systems [3], but for simplicity of presentation we will assume that at each instant in time there are $3f + 1$ replicas of data item x and quorums are any subset of those replicas of cardinality $2f + 1$, where f is a threshold set a priori that describes the maximum number of faults in each replica set.

The read and write operations are implemented as follows.

For a client, c, to write value v_{new} to variable x, it performs the following sequence of actions.

1. Send $<$READ, $epoch_c, nonce >$ message to all replicas.
2. Collect replies in a quorum Q.
3. If all replies are in the form $<$READ-REPLY, $val, ts, nonce >_{\sigma_i}$ (where σ_i represents a correct signature from the sending replica, i), choose a timestamp ts_{new} greater than the highest timestamp it read. Otherwise, if some reply is of the form $<$READ-REPLY, ERR_NEED_CONFIG, $nonce >_{\sigma_i}$, then send replica i the current configuration description, remove replica i from Q and try to form a new quorum (wait for more replies). If some reply is of the form $<$READ-REPLY, ERR_UPGRADE_CONFIG, $next\_config >$, the client verifies the authenticity of $next\_config$, upgrades its configuration and restarts. If the configuration or the reply is malformed, remove the reply from Q and wait for more replies.
4. Send $<$WRITE, $ts_{new}, v_{new}, nonce >_{\sigma_c}$ to all replicas.
5. Collect replies in a quorum Q'.
6. If all replies are in the form $<$WRITE-REPLY, ACK, $nonce >_{\sigma_i}$, the operation is complete. Otherwise, if some reply is of the form $<$WRITE-REPLY, ERR_NEED_CONFIG, $nonce >_{\sigma_i}$, then send replica i the current configuration description, remove replica i from Q' and try to form a new quorum (wait for more replies). If some reply is of the form $<$WRITE-REPLY, ERR_UPGRADE_CONFIG, $next\_config >$, the client verifies the authenticity of $next\_config$, upgrades its configuration and restarts. If the configuration or the reply is malformed, remove the reply from Q' and wait for more replies.

For a client, c, to read the value of variable x, it performs the following sequence of actions.

1. Send <READ, $epoch_c, nonce$ > message to all replicas.

2. Collect replies in a quorum Q.

3. If all replies are in the form <READ-REPLY, $val, ts, nonce >_{\sigma_i}$, choose the pair it finds with maximum timestamp, $< v, t >$. If the data v does not verify correctly, set $< v, t >$ to the reply with the next highest timestamp. Otherwise, if some reply is of the form <READ-REPLY, ERR_NEED_CONFIG, $nonce >_{\sigma_i}$, then send replica i the current configuration description, remove replica i from Q and try to form a new quorum (wait for more replies). If some reply is of the form <READ-REPLY, ERR_UPGRADE_CONFIG, $next\_config$ >, the client verifies the authenticity of $next\_config$, upgrades its configuration and restarts. If the configuration or the reply is malformed, remove the reply from Q and wait for more replies.

4. If all replies in Q agree on the timestamp t, then return v.

5. Otherwise send <WRITE, $t, v, nonce >_{\sigma_c}$ to all replicas.

6. Collect replies in a quorum Q'.

7. If all replies are in the form <WRITE-REPLY, ACK, $nonce >_{\sigma_i}$, the operation is complete and the result is $< v, t >$. Otherwise, if some reply is of the form <WRITE-REPLY, ERR_NEED_CONFIG, $nonce >_{\sigma_i}$, then send replica i the current configuration description, remove replica i from Q' and try to form a new quorum (wait for more replies). If some reply is of the form <WRITE-REPLY, ERR_UPGRADE_CONFIG, $next\_config$ >, the client verifies the authenticity of $next\_config$, upgrades its configuration and restarts. If the configuration or the reply is malformed, remove the reply from Q' and wait for more replies.

On the server side, when replica i receives a request, it validates the epoch identifier. If it is smaller than its current epoch, $epoch_i$, it issues a reply in the form <READ/WRITE-REPLY,ERR_UPGRADE_CONFIG, $config_i >_{\sigma_i}$. (Recall that the next configuration is authenticated by the trusted configuration source.) If it is greater than its current epoch it issues a reply in the form <WRITE-REPLY, ERR_NEED_CONFIG, $nonce >_{\sigma_i}$ If the epoch number is the same as its own, normal case ensues as follows.

In the case of a read request, return <READ-REPLY, $val_i, ts_i, nonce >_{\sigma_i}$.

In the case of a write request, update its local $< val_i, ts_i >$ to the received values $< v, t >$ only if $t > ts_i$. In any case, return <WRITE-REPLY, ACK, $nonce >_{\sigma_i}$.

3.1 State Transfer

Whenever a node, i, in epoch e receives a valid message with an authenticated configuration for epoch $e+1$ (via a gossip protocol or a reply to a read or write of type ERR_UPGRADE_CONFIG) it upgrades to epoch $e+1$, setting $previous\_config_i$ to $config_i$, and $config_i$ to the incoming configuration, and $epoch_i$ to $e+1$.

At this point, node i may discover it is no longer a replica of x, in which case it stops serving requests for it; or that it has become one of the replicas responsible for data item x. In the latter case, state transfer must take place from the old replica set.

For this purpose, node i sends a <STATE-TRANSFER-READ, $e{+}1 >_{\sigma_i}$ to all replicas in $previous\_config_i$. A replica j that was responsible for x in epoch e will only execute this request after upgrading to epoch $e+1$ (issuing a ERR_NEED_CONFIG reply if necessary). Unlike a normal read, the STATE-TRANSFER-READ is executed in epoch $e+1$ despite the fact that replica j may no longer be responsible for x.

State transfer reads are executed exactly as normal reads except there is no write back phase (only points 1–3 in the read protocol are executed) and $< val_i, ts_i >$ gets set to the output of the read, $< v, t >$. After this, normal protocol operation ensues: replica i will accept read, write, and state transfer reads for the next epoch.

Note that between the moment that replica i upgrades to an epoch, and the moment that replica i concludes transferring state from the previous epochs, requests for reads and writes in the new epoch must be held, and handled only when state transfer concludes.

3.2 Garbage-Collection of Old Configurations

The algorithm described above only maintains two configurations at each replica. This raises the issue of whether it is safe for replicas to discard the configuration for epoch $e-1$ when they receive the configuration for epoch $e+1$.

The issue is that the configuration for epoch $e-1$ might still be needed in the future, to bring slow replicas up-to-date. Consider the following example of such a situation.

Replica i is a replica for data item x in epochs e, $e+2$, and $e+3$, but not in epoch $e+1$. All other nodes in the system move between epochs until they reach epoch $e+3$, not using replica i for state transfer, since all messages directed to it were lost. In this situation, when replica i receives the information about epoch $e+3$, it cannot proceed directly to this epoch, since it does not know which intermediate epochs require it to transfer state, nor who to transfer this state from.

A strawman solution would be for replica i to read the data from the remaining replicas in epoch $e+3$, but if this set already contained f faulty replicas, then replica i would constitute the $f+1$st fault (albeit not Byzantine).

One might also think that this situation would be solved by having all the replicas in the system maintain configurations for epochs $e+1$ and $e+2$ until replica i reached those epochs. The problem with this approach is that a Byzantine-faulty replica could exploit this by pretending to be slow and not having heard about these epochs, which would lead to unbounded storage requirements at nonfaulty replicas.

Thus our solution is to only accept nodes into the configuration for epoch $e+1$ if they have proved that they have entered epoch e. Thus, when a nonfaulty node in epoch e hears that it is a member of the configuration for epoch $e+k, k > 1$, it knows that it must not have been part of the configuration for epoch $e+k-1$, and can restart itself by performing state transfer from the replicas in configuration $e+k-1$. This scheme can be generalized to maintain a constant number of consecutive configurations instead of only two.

Note that the same argument applies to state transfer between consecutive epochs. In order to avoid nonfaulty replicas having to maintain configuration information about several epochs until state transfer for previous epochs conclude, we assume that if replica i has not finished transferring state between epoch $e-2$ and epoch $e-1$ when epoch e is created, it will not be part of the configuration for epoch e.

Again, even though this may cause good replicas to be forced out of the replica set, this is unlikely since we assume that reconfigurations are infrequent and network outages eventually get repaired.

Note that the assumptions made above about nonfaulty nodes receving the configuration information and finishing state transfer for an epoch before the configuration for the subsequent epoch is produced are reasonable given our low rate of reconfiguration assumption. In a separate paper [5] we present an implementation of the trusted configuration source that meets these requirements by forcing nodes to contact it showing their current stable epoch number in order to proceed to the next epoch.

4 Correctness Conditions

The correctness of the system is based on the assumption that no more than f failures occur in the replica group for as long as that group is "needed". Obviously a group is needed while all its members are in the current epoch. But groups from older epochs are also needed – until their objects have been copied to the new responsible replicas. Thus we define the following correctness condition.

Correctness condition C1: For any replica group $g_e \in 2^S$ for epoch e that is produced throughout the execution of the system, g_e contains no more that f faulty processes between the moment when the first non-faulty replica for data item x in epoch e enters that epoch, until the last non-faulty replica for the same data in epoch $e+1$ finishes state transfer.

Note that this definition is not complete due to the following problem. A correct, but slow client can have a stale configuration and contact an old replica group, g, long after the group has finished transferring state to the subsequent epoch. In the meanwhile, the number of faults in the group may have exceeded f, which suffices for the group to forge a stale reply to the client.

This can be solved by having a periodic (yet possibly infrequent) check by the clients that their configuration is up-to-date. This can be done by contacting the trusted source directly in order to obtain current configuration information. Assume that every $T_{contact}$ units of time the client contacts the trusted source, and halts the execution (including outstanding requests) if it does not hear a reply for longer than $\delta_{contact}$. Also assume that this contact always outputs the latest configuration that had been issued at the time the query was produced, or a later configuration. After adding this behavior to the algorithm, we obtain the following additional correctness condition.

Correctness condition C2: Any replica group g_e contains at most f faults between the moment it is created and $T_{contact} + \delta_{contact}$ after the next replica group is created.

Note that the algorithm only provides atomic semantics for non-faulty clients, as mentioned in Section 2. For instance, a malicious client can write different values with the same timestamp to all replicas, thereby causing different replies for subsequent read with that particular timestamp. We intentionally allow this, since a faulty client can always constantly rewrite its own data, therefore giving a similar view of its contents. Furthermore, this avoids the performance penalty of running Byzantine agreement on the values proposed by the client. Therefore, our correctness proof assumes that clients follow the protocol.

5 Correctness Proof

To prove that the algorithm described in Sections 3 and 4 implements a read/write atomic object we will use a lemma that provides an alternative sufficient condition to show atomicity.

Lemma 1 (from [2]) Let β be a (finite or infinite) sequence of actions of a read/write atomic object external interface. Suppose that β is well-formed for each client i (cf. [2]), and contains no incomplete operations. Let Π be the set of all operations in β.

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. For any operation $\pi \in \Pi$, there are only finitely many operations ϕ such that $\phi \prec \pi$.
2. If the response event for π precedes the invocation event for ϕ in β, then it cannot be the case that $\phi \prec \pi$.
3. If π is a $write$ operation and ϕ is any operation in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$.
4. The value returned by each $read$ operation is the value written by the last preceding $write$ operation according to $\prec$ (or v_0, if there is no such $write$).

Then β satisfies the atomicity property (as defined in [2]).

We use this Lemma to prove atomicity of the algorithm. First we prove a few handy lemmmas.

Lemma 2 If a read or write operation completes in epoch e (i.e., its last phase executes in a quorum of the configuration for epoch e) with timestamp t, then state transfer operations between epoch e and $e+1$ read a timestamp that is greater or equal than t.

Proof. Consider the quorum where the last phase (i.e., either the write phase or the unanimous read) completes and an arbitrary read quorum that is used for state transfer. Those quorums intersect in at least one nonfaulty replica (given correctness condition C1) and that replica will return that write, unless it has overwritten with a write with a higher timestamp (since data items are only overwritten by writes with higher timestamps at nonfaulty replicas). Since a state transfer read outputs the highest valid timestamp it sees, the lemma is true. ▯

Lemma 3 If state transfer between epochs e and $e+1$ outputs timestamps greater or equal than t (for reads by nonfaulty replicas), then state transfer reads for subsequent epochs output timestamps greater or equal than t.

Proof. Every nonfaulty replica will only overwrite the data value it read during state transfer with another data if it has a higher timestamp. Therefore, when state transfer occurs for subsequent epochs, at least $f+1$ nonfaulty replicas will be contacted (by correctness condition C1) and those replicas will output data with timestamp t or higher. ▯

Lemma 4 If an operation, π, completes in epoch e (i.e., its second phase, or the unanimous read phase execute in that epoch), then for every operation ϕ such that the response event for π preceeds the invocation event for ϕ, it cannot be the case that any phase of ϕ executes in epoch $e', e' < e$.

Proof. Assume, for the sake of contradiction, that operation π precedes operation ϕ, and ϕ completes in an earlier epoch (or one of its phases does so). If an operation completed in epoch e, then all nonfaulty replicas in quorum Q where the operation executed had upgraded to epoch e at the time the reply was produced. These nonfaulty replicas will not downgrade their epoch numbers, and by correctness condition C2, subsequent operations that try to use older configurations will use quorums that contain at least $f+1$ nonfaulty replicas. One of those replicas must have upgraded to the next configuration, otherwise state transfer would not have succeeded and the operation in epoch e could not occur. This contradicts the fact that a subsequent operation executes in epoch e'. ▯

Lemma 5 All timestamp values for distinct write operations are distinct.

Proof. For writes by different clients this is true due to timestamp construction. For writes by the same client, by well-formedness, these operations occur sequentially. So all we must argue is that the read phase of the latest write "sees" the timestamp of the previous write, or a higher timestamp. If the write phase of the first write and the read phase of the second write occur in the same epoch, then this is true due to quorum intersection (the write quorum for the first write and the read quorum for the first phase of the second write intersect in at least one non-faulty replica, given correctness condition C1) and because replicas store increasing version numbers. Therefore that non-faulty replica stored a timestamp greater or equal than the one for the first write, and due to the way timestamps are chosen for write operations the timestamp chosen for the later write is greater than the timestamp stored by that replica. Now suppose that the write phase of the first write executed in epoch e, and the read phase of the second write executed in epoch e'. By Lemma 4, it must be that case that $e' > e$. In this case, Lemmas 2 and 3 will imply that state transfer for epoch e' read a timestamp greater or equal than the timestamp of the first write. Since nonfaulty replicas never overwrite with a smaller timestamp, the read phase will contact at least $f + 1$ nonfaulty replicas (given correctness conditions C1 and C2) that will present a valid timestamp greater or equal than the timestamp of the first write, and the same argument as above applies. $\square$

Now we are ready to prove atomicity of the algorithm.

Theorem 1 The algorithm described before is a read/write atomic object guaranteeing wait-free termination.

Proof. Well-formedness and wait-free termination as defined in [2] are easy to see.

For atomicity, we use Lemma 1. We define a partial ordering on operations in Π. Namely, we say that $\pi \prec \phi$ if either of the following applies:

1. $timestamp(\pi) < timestamp(\phi)$
2. $timestamp(\pi) = timestamp(\phi)$, π is a $write$, and ϕ is a $read$

where $timestamp(\pi)$ is defined as the timestamp written in the second phase of operation π, or unanimously read in the first phase of a read.

It is enough to verify that this satisfies the four conditions needed for Lemma 1.

1. For any operation $\pi \in \Pi$, there are only finitely many operations ϕ such that $\phi \prec \pi$.

   Suppose for the sake of contradiction that operation π has infinitely many $\prec$ predecessors. Lemma 5 implies that it cannot have infinitely many predecessors that are $write$ operations, so it must have infinitely many predecessors that are $read$ operations. Without loss of generality, we may assume that π is a $write$.

   Then there must be infinitely many $read$ operations with the same timestamp, t, where t is smaller than $timestamp(\pi)$. But the fact that π completes in the execution of the algorithm implies that $timestamp(\pi)$ gets written to a quorum in an epoch e. After this happens, any $read$ operation that is subsequently invoked either completes its read phase in epoch e or in an epoch $e' > e$ (by Lemma 4). If it completes the read phase in epoch e, it will read $timestamp(\pi)$ or a higher timestamp, due to quorum intersection and the fact that replicas store increasing version numbers. If it completes in an epoch $e' > e$, Lemmas 2 and 3, and the fact that nonfaulty replicas store increasing version numbers imply that the read phase in epoch e' will read $timestamp(\pi)$ or a higher timestamp. This contradicts the existence of infinitely many $read$ operations with timestamp t.

2. If the response event for π precedes the invocation event for ϕ in β, then it cannot be the case that $\phi \prec \pi$.

   By Lemma 3, both phases of ϕ must complete in an epoch greater or equal than the epoch in which π completes.

   Suppose that the write phase of π executes in the same epoch as the read phase of ϕ. In this case, quorum intersection will ensure that the read phase of ϕ will see a value greater or equal than $timestamp(\pi)$ which suffices to show the implication above for this case. (Note that some operations do not have a write phase, but only if it is a read where all replicas agree on the version number, in which case the same argument applies to the unanimous read phase of π instead of its write phase.)

   Otherwise, the write phase (or unanimous read) of π executes in an epoch e earlier than the epoch of the read phase of ϕ, $e' > e$. In this case, Lemmas 2 and 3 will imply that state transfer for epoch e' read a timestamp greater or equal than the timestamp of the first write. Since nonfaulty replicas never overwrite with a smaller timestamp, the read phase will contact at least $f + 1$ nonfaulty replicas that will present a valid timestamp greater or equal than the $timestamp(\pi)$, and the same argument as above applies.

3. If π is a $write$ operation and ϕ is any operation in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$.

   By Lemma 5, all $write$ operations obtain distinct timestamps. This implies that all of the $write$s are totally ordered, and also that each $read$ is ordered with respect to all the $write$s.

4. The value returned by each $read$ operation is the value written by the last preceding $write$ operation according to $\prec$ (or v_0, if there is no such $write$).

   Let π be a $read$ operation. The value v returned by π is just the value that π finds associated with the largest timestamp, t, among the replies in quorum Q. This values also becomes the timestamp associated with π. By the unforgeability of data items, there can only be two cases:

   - Value v has been written by some $write$ operation ϕ with timestamp t.

     In this case, the ordering definition ensures that ϕ is the last $write$ preceding π in the $\prec$ order, as needed.

   - $v = v_0$ and $t = 0$.

     In this case, the ordering definition ensures that there are no $write$s preceding π in the $\prec$ order, as needed.

$\square$

6 Conclusions

We presented an algorithm for a dynamic Byzantine-fault-tolerant read/write atomic object, the correctness conditions under which that algorithm provides atomicity, and a correctness proof.

More details about the practical aspects of the implementation can be found in a separate publication [5]. In the future, we intend to provide a more formal proof resorting to I/O automata [2].

References

[1] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[2] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[3] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[4] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

[5] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Submitted for publication.