Efficient Fork-Linearizable Access to Untrusted Shared Memory

Christian Cachin
IBM Research
Zurich Research Laboratory
Ruschlikon, Switzerland
cca@zurich.ibm.com

abhi shelat
IBM Research
Zurich Research Laboratory
Ruschlikon, Switzerland
abs@zurich.ibm.com

Alexander Shraer
Dept. of Electrical Engineering
Technion
Haifa, Israel
shralex@tx.technion.ac.il

ABSTRACT

When data is stored on a faulty server that is accessed concurrently by multiple clients, the server may present inconsistent data to different clients. For example, the server might complete a write operation of one client, but respond with stale data to another client. Mazières and Shasha (PODC 2002) introduced the notion of *fork-consistency*, also called *fork-linearizability*, which ensures that the operations seen by every client are linearizable and guarantees that if the server causes the views of two clients to differ in a single operation, they may never again see each other's updates after that without the server being exposed as faulty. In this paper, we improve the communication complexity of their fork-linearizable storage access protocol with n clients from $\Omega(n^2)$ to $O(n)$. We also prove that in every such protocol, a reader must wait for a concurrent writer. This explains a seeming limitation of their and of our improved protocol. Furthermore, we give novel characterizations of fork-linearizability and prove that it is neither stronger nor weaker than sequential consistency.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Performance, Reliability, Theory

Keywords

Fork-consistency, Storage emulations, Arbitrary failures

1. INTRODUCTION

Many users no longer keep all their data on local storage. Instead, their data often resides on remote, online service providers. Systems with such remotely stored information include network filesystems, online collaboration servers such as Wikis, source code repositories using versioning tools like CVS, and web-based email providers. Users rely on the provider to maintain the integrity of the stored data, but there is no generally available technology that allows a user to easily verify that no subtle modification has been introduced to the data. In other words, users must trust the storage provider.

When the users locally maintain even a small amount of trusted memory, the trust in the storage provider can be greatly reduced using well-known cryptographic methods. A single, isolated user may verify the integrity of its remotely stored data by keeping a short *hash value* of the data in its local memory. When the volume of data is large, this method is usually implemented using a Merkle hash tree. But in multi-user environments, integrity should be guaranteed between a writer and multiple readers, for which hashing alone is not enough. *Digital signatures* achieve data integrity against modifications by the server when the users sign all their data. Every user only needs to store an authenticated signature public-key of the others or the root certificate of a public-key infrastructure in its trusted memory.

Neither of the above methods rules out all attacks by a faulty or malicious server. Even if all data is signed during write operations, the server might present the modifications in a different order to a client, it may decide to omit a recent update to some user and present a different subset of the write operations to another user.

In the model considered here, there are no common clocks and the clients do not communicate with each other. Some of the above attacks can therefore *never* be prevented. In particular, the server may use an outdated value in the reply to a reader and omit a more recent update. Mazières and Shasha [18] present a protocol called *SUNDR* that does not prevent such attacks, but makes them easily detectable. The SUNDR protocol ensures that whenever the server causes the views of two clients to differ in a single operation, the two clients may never again see each other's updates after that. Such partitioning can easily be detected through out-of-band communication.

Mazières and Shasha [18] introduced the notion of *fork consistency* for the properties that their protocol provides to a set of clients concurrently accessing the server. Fork-consistency ensures that every client sees a *linearizable* [11] history of read and write operations, i.e., one that is consistent with all operations observed by the client, such that the operations seen in the histories of all clients can be arranged in a "forking tree." Oprea and Reiter [19] suggested the name *fork-linearizability* for fork-consistency.

Contribution. In this paper, we make two contributions. First, we investigate the notion of fork-linearizability of shared memory consisting of read/write registers. We show that every protocol emulating fork-linearizable shared memory on a possibly faulty server involves executions, where the server is correct but one client cannot complete an operation because it must wait for another client to perform some computation steps. In other words, no such emulation is obstruction-free [10]. This result explains a seeming limita-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODC'07, August 12–15, 2007, Portland, Oregon, USA.
Copyright 2007 ACM 978-1-59593-616-5/07/0008 ...$5.00.

tion of the SUNDR protocol and of our protocol, where a reader must wait for a concurrent writer. This also explains why any asynchronous protocol emulating fork-linearizable shared memory, which guarantees that all operations by correct clients complete if the server is correct, must assume that clients do not fail. Next, we show that sequential consistency cannot be guaranteed with a possibly malicious server, and that fork-linearizability is neither stronger nor weaker than sequential consistency in the sense that fork-linearizable executions may not be sequentially consistent and vice versa. These results give further motivation for studying forking conditions. Furthermore, we give a "global" definition of fork-linearizability in terms of a single sequential history of operations, of which the clients observe subsequences that respect the sequential specification.

Second, we provide an efficient protocol for emulating fork-linearizable shared memory on an untrusted server. Our protocol is inspired by the SUNDR protocol [18] and is also based on vector timestamps. In a system with n clients, our protocol improves the communication complexity to $O(n)$ from $\Omega(n^2)$ in the SUNDR protocol. Intuitively, our improvement results from relying only on the vector timestamp of the client that executed the most recent operation, instead of relying on the vector timestamps of all clients.

Analogously to the work about SUNDR [18], we present two protocols: a protocol that proceeds in "lock-step," where the server blocks during every client operation, and a "concurrent" protocol, where the clients may proceed at their own speed and interact concurrently with the server, up to the limitation mentioned above. We note that even though our protocol follows the general pattern of the SUNDR protocol, we need a new proof that it guarantees fork-linearizability. Our efficiency improvement comes at the cost of introducing certain vulnerabilities that may be exploited by faulty clients colluding with the server. The SUNDR protocol prevents some of these attacks by using n vector timestamps, although Mazières and Shasha [18] do not formally state any properties about the SUNDR protocol with faulty clients.

To see the significance of our contribution, consider a widely used Internet-based storage system with thousands of registered users. Suppose that $n = 10000$. Our algorithm and the SUNDR protocol both require that at least one timestamp per user is sent during every operation, i.e., typically at least 4 Bytes. Thus, whereas our algorithm will send $4n = 40KB$ per operation, the SUNDR protocol will require to send $4n^2 = 380MB$ per operation.

Related work. The SUNDR protocol [18] has been implemented in a practical distributed filesystem called SUNDR [14], which provides fork-linearizable semantics for its "fetch" and "modify" operations. SUNDR demonstrates that ensuring fork-linearizability in network filesystems is practical. There are many distributed filesystems that rely on digital signatures for checking the integrity of the stored data, but only SUNDR prevents attacks on the consistency of the client views through fork-linearizability.

Oprea and Reiter [19] generalize fork-linearizability and introduce, among other notions, the interesting concept of fork-sequential-consistency, where the sequence of read and write operations seen by every client is only required to be sequentially consistent. They investigate cryptographic filesystems, where a high-level encrypted file object is implemented by a file-key object and a file-data object stored on an untrusted server.

There is a rich literature on verifying the correctness of untrusted memories without concurrent access by using hashing. It ranges from the fundamental work of Blum *et al.* [3] to investigations on "incremental" hash functions [5], motivated by the goal to construct secure processors with untrusted main memory [6]. Hash trees have also been used in several filesystems, starting with Fu's work [8] and the SFSRO filesystem [7].

Finally, it is worth pointing out that this paper does *not* address the question of emulating shared memory on a set of storage servers, of which a fraction may fail or deviate in arbitrary ways from their specification [17, 1]. These emulations do not provide any guarantees with a majority of faulty servers, unlike the protocols considered here.

Outlook. Our work can be extended in several directions. Our proof that waiting is necessary in fork-linearizable executions does not apply to fork-sequentially-consistent executions. Thus, protocols emulating the weaker notion of fork-sequential-consistency (defined in Section 2) are potentially more efficient and more robust. Another important avenue for future work is to consider faulty clients and to investigate communication-efficient protocols that achieve forking consistency conditions in this model. Some simple precautions by the server, such as checking signatures and orderings, may already prevent many attacks by malicious clients. But formal consistency notions taking into account faulty clients are necessary to capture this realistic situation. It would also be interesting to provide lower bounds on the communication complexity of fork-linearizable emulations and to investigate the use of other methods for ensuring causal order between operations.

Organization of the paper. The remainder of the paper is organized as follows. Definitions are presented in Section 2. The three results that characterize fork-linearizability are contained in Section 3. Section 4 describes the lock-step protocol to emulate fork-linearizable shared memory, and Section 5 presents the emulation allowing concurrent operations.

2. DEFINITIONS

2.1 System Model

The system consists of n clients $C_1, \ldots, C_n$ and a server S that are modeled as I/O Automata [16, 15]. Clients and servers are collectively called *parties*. All clients are assumed to be correct and to follow the protocol. The server, however, may be faulty and deviate arbitrarily from its protocol, exhibiting so-called "Byzantine" [20] or "NR-arbitrary" faults [12].

The parties interact by sending messages over an asynchronous network which consists of reliable point-to-point links. We wish to design a protocol where the server provides a *shared functionality* F to the clients; the functionality is defined using terminology from the "shared memory model" of distributed computation. F is similar to a shared object, but may violate liveness because the server that implements it may be faulty. Our goal will be to show that the protocol constrains the server so that it simulates to the clients an interaction with F.

The clients interact with the functionality F by accessing *operations* provided by F. An operation is defined in terms of two *events* occuring at the client, denoting the *invocation* and the *completion* of the operation. These two events are sometimes also called *request* and *response*, respectively. An operation is *invoked* on a client at some point in time and *completes* at a later point in time when a response from F reaches the client. An operation o_1 is said to *precede* another operation o_2 whenever o_1 completes before o_2 is invoked. Two operations are called *sequential* if one of them precedes the other one and *concurrent* otherwise. A sequence of events is called *sequential* if it only contains sequential operations.

An *execution* of the system consists of a sequence of events and internal state transitions at the parties and is asynchronous.

The clients generate arbitrary sequences of requests for F, but we assume that clients always *interact sequentially* with a given functionality, i.e., the sequence of events on a client consists of alternating invocations and matching responses, starting with an invocation. A functionality may further require that clients *comply* with *problem-specific restrictions* on the allowed sequences of requests.

The functionality F is defined via a *sequential specification*, which indicates the behavior of F when all interactions between the clients and F are sequential.

2.2 Read/Write Registers

The basic functionality that we consider is a *read/write register* X. It is defined as follows. The register stores a value v from a domain $\mathcal{V}$ and offers a *read* and a *write* operation. Initially, every register contains a special value $\perp \notin V$. When a client C_i invokes the read operation, denoted $read_i(X)$, the functionality responds by returning a value v, denoted $read_i(X) \rightarrow v$. When C_i invokes the write operation with a value v, denoted $write_i(X, v)$, the response of X is an acknowledgment OK. The sequential specification of X requires that each read operation from X returns the value written by the most recent preceding write operation, if there is one, and the initial value otherwise. We assume that the values written to any particular register are *unique*. This can easily be implemented by including the identity of the writer and a sequence number together with the stored value.

Registers come in several variations [13] depending on whether one or more clients can invoke its operations. In this paper, we consider single-writer/multi-reader (SWMR) registers, where for every register, only a designated "writer" may invoke the write operation, but any client may invoke the read operation. The functionality considered here consists of n SWMR read/write registers $X_1, \ldots, X_n$. The registers are usually identified by their indices and may be accessed independently of each other.

2.3 Consistency Conditions

When a shared functionality is accessed by concurrent operations, the sequential specification alone may not be powerful enough to provide a meaningful semantics to the clients. In this subsection, we define three different *consistency conditions* with respect to a shared functionality under concurrent access. Two of them are well-known [2]: *sequential consistency* and *linearizability*. The third one, *fork-linearizability*, has been introduced by Mazières and Shasha [18] under the name of *fork-consistency* in the context of storage systems that may deviate from their specification. Implementing a fork-linearizable shared memory with an untrusted server is the main goal of this work.

A consistency condition is expressed in terms of the sequence of events that the shared functionality may exhibit in an execution, as observed by the clients. Such a sequence is also called a *history*; a history σ is said to be *complete* whenever all invocations in σ have a matching response.

Definition 1 (Preservation of real-time order). A permutation π of a sequence of events σ is said to *preserve the real-time order* of σ if for every operation o that precedes an operation o' in σ, the operation o also precedes o' in π.

An important consistency condition is linearizability [11], which guarantees that all operations occur "atomically," i.e., appear to be executed at a single point in time.

Definition 2 (Linearizability [11]). A sequence of events σ observed by the clients demonstrates *linearizability* with respect to a functionality F if and only if there exists a sequential permutation π of σ such that:

1. π preserves the real-time order of σ; and
2. The operations of π satisfy the sequential specification of F.

In other words, a sequence of operations on a functionality is linearizable if there is a way to reorder the operations into a sequential execution that respects the semantics of the functionality and that respects the ordering of events as seen by a global observer.

Sequential consistency is a weaker notion than linearizability and only imposes a total order on the events observed by every client in isolation.

Definition 3 (Sequential consistency). A sequence of events σ observed by the clients demonstrates *sequential consistency* with respect to a functionality F if and only if there exists a sequential permutation π of σ such that:

1. For every client C_i, the restriction of π to the events occuring at C_i preserves the real-time order of σ restricted to the events occurring at C_i; and
2. The operations of π satisfy the sequential specification of F.

Neither linearizability nor sequential consistency can be achieved when F is implemented on a Byzantine server (at least not for functionalities F where some operations do not commute). For instance, suppose that C_i was the last client to execute an operation on F; no matter what protocol the clients use to interact with the server, a faulty server might roll back its internal memory to the point in time before executing the operation on behalf of C_i, and pretend to a client C_j that C_i's operation did not occur. As long as C_j and C_i do not communicate with each other, neither party can detect this violation and thus neither definition can be satisfied (for a formal proof, see Section 3.3).

Mazières and Shasha [18] called such behavior a *forking attack*. They postulate that *forking two (sets of) clients* by introducing discrepancies between the events observed by them is the *only* way in which a faulty server may violate the consistency of a functionality that it provides. In particular, it should be ruled out that the server causes any common operation to be observed by two distinct clients after they have been forked, i.e., to join the sequences of observed operations again. In the above example, C_i should not see any data written by C_j after the forking attack.

The notion of *fork-linearizability* [18] captures this intuition by requiring that the history of events occuring at every client satisfies the conditions of linearizability and that for any operation visible to multiple clients, the history of events occuring before the operation is the same.

Definition 4 (Fork-Linearizability). A sequence of events σ observed by the clients is called *fork-linearizable* with respect to a functionality F if and only if for each client C_i, there exists a subsequence σ_i of σ consisting only of completed operations and a sequential permutation π_i of σ_i such that:

1. All completed operations in σ occurring at client C_i are contained in σ_i; and
2. π_i preserves the real-time order of σ_i; and
3. The operations of π_i satisfy the sequential specification of F; and
4. For every $o \in \pi_i \cap \pi_j$, the sequence of events that precede o in π_i is the same as the sequence of events that precede o in π_j.

Note that a fork-linearizable history that does not fork and satisfies $\pi_i = \pi$ for all clients is linearizable. Moreover, conditions 2 and 3 imply that each σ_i is linearizable with respect to F.

Oprea and Reiter [19] consider forking attacks not only for linearizable implementations but also with other consistency conditions. For instance, they define the notion of fork-sequential-consistency; but we do not address it in this work.

2.4 Byzantine Emulations

Our goal is to provide a protocol for the clients that emulates a functionality F with the help of server S under a given consistency condition. Such a protocol P consists of n identical algorithms running locally on every client and one algorithm running on the server (when it is correct). The algorithms may send messages to each other over the network. We define an emulation in terms of events observed by the clients when they run P, and require that the sequence of these events correspond to a possible interaction of the clients with F.

Our notion of *Byzantine emulation* is derived from the definition of a fault-tolerant implementation of a shared object by Jayanti et al. [12]. It differs from the latter with respect to handling non-responsive faults of the server and by allowing forking attacks. If the server is faulty, any functionality that it should emulate may have operations that do not complete, causing the emulation to violate liveness; furthermore, the server may fork two clients in arbitrary ways. The intuition behind our notion of Byzantine emulation is that introducing forks and violating liveness are also the only possible ways in which the protocol execution may differ from an interaction of the clients with F. If the server is correct, of course, the emulation has to satisfy liveness and must not fork.

An execution of a system is called *admissible* when the requests generated by the clients comply with the problem-specific restrictions for F and the execution satisfies "fairness." *Fairness* means, informally, that the execution does not halt prematurely, when there are still steps to be taken or messages to be delivered; we refer to the standard literature for a formal definition [15, 2].

Definition 5 (Fork-linearizable Byzantine emulation). We say that a protocol P for a set of clients *emulates* a functionality F *on a Byzantine server S with fork-linearizability* if and only if in every admissible execution of P, the sequence of events observed by the clients is fork-linearizable with respect to F. Moreover, if S is correct, then every admissible execution is complete and has a linearizable history.

We remark that for other consistency conditions Γ such as sequential consistency, the notion of a *fork-Γ-consistent Byzantine emulation* may be defined analogously.

2.5 Cryptographic Primitives

The protocols of this paper require *hash functions* and *digital signatures* from cryptography. Because the focus of this work is on concurrency and correctness and not on cryptography, we model both as ideal functionalities implemented by a trusted entity.

A hash function maps a bit string of arbitrary length to a short, unique representation. The functionality provides only a single operation H; its invocation takes a bit string x as parameter and returns an integer h with the response. The implementation maintains a list L of all x that have been queried so far. When H is invoked with $x \in L$, then H responds with the index of x in L; otherwise, H adds x to L at the end and returns its index. This ideal implementation models only collision-resistance but no other properties of real hash functions. The server may also invoke H.

The functionality of the digital signature scheme provides two operations, *sign* and *verify*. The invocation of *sign* takes an index $i \in \{1, \ldots, n\}$ and a string $m \in \{0,1\}^*$ as parameters and returns a signature $s \in \{0,1\}^*$ with the response. The *verify* operation takes the index i of a client, a putative signature s, and a string $m \in \{0,1\}^*$ as parameters and returns a Boolean value $b \in \{\text{FALSE}, \text{TRUE}\}$ with the response. Its implementation satisfies that *verify*$(i, s, m) = \text{TRUE}$ for all $i \in \{1, \ldots, n\}$ and $m \in \{0,1\}^*$ if and only if C_i has executed *sign*$(i, m) \rightarrow s$ before, and *verify*$(i, s, m) = \text{FALSE}$ otherwise. Only C_i may invoke *sign*$(i, \cdot)$ and S cannot invoke *sign*. Every party may invoke *verify*. In the following we denote *sign*(i, m) by *sign*$_i(m)$ and *verify*(i, s, m) by *verify*$_i(s, m)$.

3. ON FORK-LINEARIZABILITY

This section contains three results that characterize the notion of fork-linearizability. We first show that no protocol for fork-linearizable shared memory emulation on a faulty server is obstruction free, then provide an alternative definition of fork-linearizability, and finally demonstrate that fork-linearizability is incomparable with sequential consistency.

3.1 Waiting is Necessary

It is well-understood that lock-based algorithms for synchronizing concurrent access to shared data are problematic and that *wait-free* [9] synchronization methods are desirable and often more efficient. A wait-free algorithm ensures that any client may complete any operation in a finite number of steps, regardless of the execution speeds of the other clients. Weaker progress conditions have also been introduced, and include *lock-freedom*, *fw-termination* [1], and *obstruction freedom* [10]. In an obstruction-free algorithm, every operation of a correct client is guaranteed to complete eventually when the client is allowed to take enough steps alone, without other clients taking steps, i.e., when there is no contention.

In this section, we show that any emulation of fork-linearizable shared memory with a possibly faulty server must involve executions where some client is delayed by another client. The basic idea of the proof is to consider a read operation by P_j and then a write operation by P_i along with a series of concurrent read operations by P_j. The first read operation must return the old value of P_i's register. But if one of P_j's reads occurs after the very last message from P_i to the server, then fork-linearizability requires the read operation to return the newly written value. Thus, at some point the read operations of P_j switch from returning the old value to returning the new value. We argue formally that the point at which this switch occurs creates an opportunity for a faulty server to violate fork-consistency. The only solution around this problem is for the concurrent reader to wait.

Theorem 1. *Let P be a protocol for emulating $n \geq 1$ SWMR registers on a Byzantine server S with fork-linearizability. Then there is an execution of P where S is correct and an operation of some client cannot complete unless another client takes some steps.*

Proof. Towards a contradiction, assume that in all states of every execution of P in which S is correct and a client C_i has invoked an operation o that has not yet completed, there is a continuation of the execution that includes the completion of o and that consists entirely of events and transitions of S and of C_i.

Recall that protocol P describes the asynchronous interaction of C_i with S for emulating o. We assume w.l.o.g. that the emulation of o consists of an exchange of messages $a_1, b_1, a_2, b_2, \ldots, b_k, a_{k+1}$ between S and C_i, where C_i first sends a_1, and for $j = 1, 2, \ldots, k$,

the server sends b_j in response to receiving a_j and C_i sends a_{j+1} in response to receiving b_j. Message b_k is the last message from S and o completes only after C_i receives it; message a_{k+1} may be missing, in which case the emulation of o ends with b_k.

We construct an execution α, in which S is correct. The execution is shown in Figure 1 and consists of operations by clients C_1 and C_2 that access only one register X_1. First, C_1 executes $w_1^1 = \textit{write}_1^1(X_1, u) \rightarrow \text{OK}$. Let s_0 denote the point in time when the server receives the last message from C_1 in the emulation of w_1^1. After w_1^1 completes, C_2 invokes an operation $r_2^1 = \textit{read}_2^1(X_1)$ that returns u. Subsequently, C_1 executes a write operation $w_1^2 = \textit{write}_1^2(X_1, v) \rightarrow \text{OK}$, which eventually completes because S is correct. Operation w_1^2 consists of messages $a_1, b_1, \ldots, b_k$, and possibly a_{k+1}, as defined above. Let $s_1, \ldots, s_k$ denote the points in time when S sends $b_1, \ldots, b_k$, respectively. The points s_0, s_1, $\ldots$, s_k are marked by dots in Figure 1.

Concurrently to the execution of w_1^2, client C_2 performs a sequence of read operations $r_2^2, \ldots, r_2^k$, such that r_2^m executes between s_{m-1} and s_m for $m = 2, \ldots, k$. By the assumption of the theorem, every operation r_2^m can terminate without any steps by C_1 and before S receives a_m. Finally, C_2 invokes another read operation r_2^{k+1} after s_k that completes before a_{k+1} reaches S (if a_{k+1} exists).

Observe the values returned by the read operations $r_2^1, r_2^2, \ldots,$ r_2^{k+1} of C_2. Since the server is correct, the execution is linearizable. Hence, the first read r_2^1 must return u because it occurs sequentially after w_1^1 and before w_1^2. The last read r_2^{k+1} might return either u or v by linearizability alone, because it is concurrent to w_1^2 and two concurrent operations may be ordered either way. But we now show that r_2^{k+1} cannot return u under the condition that P produces only linearizable executions when S is correct.

Claim 1.1. *Operation r_2^{k+1} in execution α returns v.*

Proof. Towards a contradiction, assume that r_2^{k+1} returns u. Consider another execution α', in which the server is correct and which is identical to α up to the following difference: Operation w_1^2 completes in α' *before* r_2^{k+1} is invoked, but a_{k+1} (if it exists) still arrives after the completion of r_2^{k+1}. Client C_2 cannot distinguish execution α' from α and returns u as in α. But this violates linearizability, which must be preserved in α' because S is correct. □

Thus, the first read r_2^1 returns u and the last read r_2^{k+1} returns v. Since α is linearizable, there exists a point in time (the "linearization point" of w_1^2) at which the reads by C_2 switch from returning u to returning v. We let $z > 1$ be the index of the first read that returns v, i.e., $r_2^1, \ldots, r_2^{z-1}$ return u and $r_2^z, \ldots, r_2^{k+1}$ return v. However, we next show that r_2^z cannot return v under the condition that P produces only fork-linearizable executions when S is faulty.

Claim 1.2. *Operation r_2^z in execution α cannot return v.*

Proof. Assume towards a contradiction that r_2^z in α returns v. We construct an execution β, in which S is correct. First, C_1 executes w_1^1 as in α. If $z > 2$, then the continuation of β is identical to α up to the point s_{z-2}, when S has received a_{z-2} and sent b_{z-2}, and after operation r_2^{z-2} by C_2 has completed. After s_{z-2}, no further operations by C_2 occur and w_1^2 completes by steps of S and C_1 alone. Otherwise, if $z = 2$, the continuation of β after s_0 consists only of w_1^2 and there are no read operations by C_2.

We next construct an execution γ, in which S deviates from the protocol. Execution γ starts out by performing all steps of β, thus, client C_1 cannot distinguish these two runs. After w_1^2 has completed, C_2 invokes r_2^{z-1} and then r_2^z. Notice that a faulty S can construct the state at point s_{z-2} just like in α, since β is a prefix of γ, and because β is also a prefix of α up to s_{z-2}. Thus, S can emulate r_2^{z-1} to C_2 in the same way as the correct S in α, and r_2^{z-1} returns u as in α.

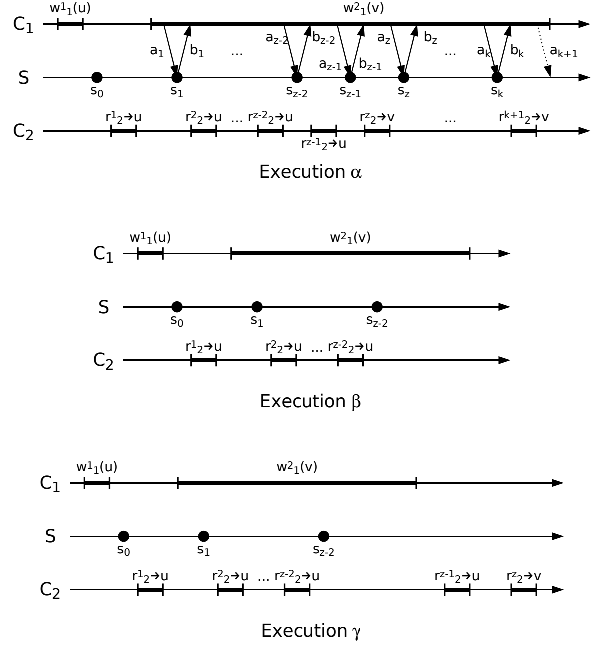

Figure 1: Three executions α, β, and γ with $z > 2$, as described in the text. C_1 cannot distinguish execution γ from β and C_2 cannot distinguish γ from α.

But now, the faulty server may also reconstruct the state of S at point s_{z-1} in α. Note that this state may only depend on the state of S at s_{z-2}, on operation r_2^{z-1}, and on message a_{z-1}. The server possesses the same information also in γ: the state at s_{z-2} and operation r_2^{z-1} are exactly as in α by construction, and message a_{z-1} from C_1 in α does *not depend* on operation r_2^{z-1} by C_2 because a_{z-1} may only depend on b_{z-2} that was sent before the invocation of r_2^{z-1}. Given the state at point s_{z-1} in α, the server can emulate r_2^z to C_2 in the same way as the correct S in α, and r_2^z returns v as in α.

Note that C_1 cannot distinguish execution γ from β and C_2 cannot distinguish γ from a prefix of α. Executions α and β are both linearizable with a correct server. But γ is not fork-linearizable because the subsequences σ_2 and π_2 according to Definition 4 would have to include all operations of γ, and γ is not linearizable. Hence, the faulty server can violate fork-linearizability in γ, contradicting the requirement that protocol P allows only fork-linearizable executions. □

We have shown that no read operation among $r_2^2, \ldots, r_2^{k+1}$ can be the first to return v. Thus, r_2^{k+1} in execution α returns neither u nor v. This contradicts our assumption that in every execution with a correct server, any operation of a client may always complete without waiting for another client to take any steps. □

This result explains why in the concurrent algorithm of Mazières and Shasha [18] and in our concurrent algorithm of Section 5, a read operation is blocked until a concurrent write operation has completed. Let us extend the standard terminology [9, 12] and call

an emulation protocol using a Byzantine server S *wait-free* if every client C_i that has invoked an operation can complete the operation together with a correct S from any state of its execution, even when no other client takes any steps. Theorem 1 implies that no fork-linearizable emulation of $n \geq 1$ SWMR registers on a Byzantine server is wait-free.

If we consider a slightly more general model, where clients may fail by crashing, we also obtain the following corollary.

Corollary 2. *No protocol for emulating $n \geq 1$ SWMR registers on a Byzantine server with fork-linearizability is obstruction-free.*

Proof. According to Theorem 1, there exist executions in which the server is correct and a client C_i must wait for another client C_j to take steps. Now, if C_j crashes, C_i remains blocked forever because it has no way to distinguish the situation from a situation with a slow network. This situation is obviously obstruction-free since a crashed client does not take any steps. □

3.2 Global Fork-Linearizability

The existing definition of fork-linearizability requires that all operations of a history σ can be arranged in a "forking tree" such that the history on every branch, represented by σ_i, is linearizable; that is, for every C_i, there exists a sequential permutation π_i of σ_i that preserves the real-time order of σ_i and satisfies the sequential specification of the functionality. We show here that this notion of fork-linearizability is equivalent to the seemingly stronger notion in which there exists a "global," sequential permutation π of σ that respects the real-time order of σ. The histories π_i are merely subsequences of π. This clarifies the notion of fork-linearizability and may lead to simpler arguments about protocols emulating fork-linearizable behavior.

Definition 6 (Global Fork-Linearizability). A sequence of events σ observed by the clients demonstrates *global fork-linearizability* with respect to a functionality F if and only if there exists a *sequential* permutation π of σ such that:

1. π preserves the real-time order of σ; and
2. For each client C_i, there exists a subsequence π_i of π such that:
   (a) All events in π occurring at client C_i are contained in π_i; and
   (b) The operations of π_i satisfy the sequential specification of F; and
   (c) For every $o \in \pi_i \cap \pi_j$, the sequence of events that precede o in π_i is the same as the sequence of events that precede o in π_j.

We prove the following theorem in the full version [4].

Theorem 3. *A sequence of events is fork-linearizable if and only if it is globally fork-linearizable.*

3.3 Comparing with Sequential Consistency

Recall that every linearizable history is trivially fork-linearizable and that there is no protocol that provides a linearizable emulation of even one SWMR register on a Byzantine server S. But this does not rule out that S may emulate a register with a weaker consistency notion. Sequential consistency, for example, does not have to preserve the real-time order of operations. It would be acceptable for a correct server to return old register values, as long as it preserves the relative order in which it shows them to every client. However, we show in the following theorem that a faulty server may also violate sequential consistency when it emulates more than one register.

Theorem 4. *There is no protocol that emulates $n > 1$ SWMR registers on a Byzantine server with sequential consistency.*

Proof. For any protocol P which emulates two SWMR registers X_1 and X_2, we demonstrate an execution λ involving a faulty server S which violates sequential consistency.

The execution consists of four operations by the clients C_1 and C_2. Client C_1 executes $write_1(X_1, v) \rightarrow$ OK and $read_1(X_2) \rightarrow \perp$. The server interacts with C_1 as if it was the only client executing any operation. Concurrently, C_2 executes $write_2(X_2, v) \rightarrow$ OK and $read_2(X_1) \rightarrow \perp$ and S also pretends to C_2 that it is the only client executing any operation. Such "split-brain" behavior is obviously possible when S is faulty: it can act as if the write operations to X_1 and X_2 have completed, as far as the writing client is concerned, but still return the old values of X_1 and X_2 in the read operations. Since the only interaction of the clients is with S, neither client can distinguish execution λ from a sequentially consistent execution where it executes alone.

Notice λ is not sequentially consistent: There is no permutation of the operations in λ in which the sequential specification of both X_1 and X_2 is preserved and, at the same time, the order of operations occurring at each client is the same as their real-time order in λ. Specifically, in any possible permutation of λ, the operation $read_1(X_2) \rightarrow \perp$ cannot be positioned after $write_2(X_2, v)$, since the read would have to return $v \neq \perp$ according to the sequential specification of X_2. However, $read_1(X_2) \rightarrow \perp$ may neither occur before $write_2(X_2, v)$ as we now argue. Since the local order of operations has to be the same as in λ in this case, $write_1(X_1, u)$ must occur before $read_1(X_2) \rightarrow \perp$ and hence also before $write_2(X_2, v)$. But since the latter operation precedes $read_2(X_1) \rightarrow \perp$ in the local order seen by C_2, we conclude that $write_1(X_1, u)$ precedes $read_2(X_1) \rightarrow \perp$, which contradicts the sequential specification of X_1. Thus, λ is not sequentially consistent. □

Note that execution λ constructed in the proof above is fork-linearizable but not sequentially consistent. On the other hand, execution γ exhibited in the proof of Theorem 1 and shown in Figure 1 is sequentially consistent but not fork-linearizable. Hence, we obtain the following result.

Corollary 5. *Fork-linearizability is neither stronger nor weaker than sequential consistency.*

4. A SIMPLE IMPLEMENTATION

In this section, we present a simple protocol that implements a shared memory on a Byzantine server S and guarantees fork-linearizability. It is called the *lock-step protocol* and is derived from the *bare-bones protocol* of Mazières and Shasha [18], but achieves the same task more efficiently. Whereas their bare-bones protocol requires messages of size $\Omega(n^2)$, the size of the messages in our lock-step protocol is $O(n)$.

In the lock-step protocol, a client sends a SUBMIT message containing a request to server S. Upon accepting the request, S sends a REPLY message with current state information to the client and stops accepting further requests, until it receives a final COMMIT message from the client. Hence, the name lock-step protocol. The detailed protocol is shown in Algorithms 1 and 2. The main result about the lock-step protocol is the following theorem, proved in the full version [4].

Theorem 6. *The lock-step protocol described in Algorithms 1 and 2 emulates n SWMR registers on a Byzantine server with fork linearizability.*

Algorithm 1 Lock-step protocol, algorithm for client C_i.

1: **state**
2: $T[j] \in \mathbb{N}$, initially 0, for $j = 1, \dots, n$ // current version of C_i
3: $\bar{x} \in \{0,1\}^*$ // most recently written value

4: *write*(x)
5: $\bar{x} \leftarrow x$
6: send $\langle \text{SUBMIT}, \textit{WRITE}, \perp \rangle$ to S
7: **wait for** a message $\langle \text{REPLY}, V, \ell, \varphi' \rangle$ from S
8: **if not** $\big([V = (0, \dots, 0)$ **or** $\text{verify}_\ell(\varphi', \text{COMMIT}\|V)]$
9: **and** $T \leq V$ **and** $T[i] = V[i]\big)$ **then halt**
10: $T \leftarrow V; T[i] \leftarrow T[i] + 1$
11: $\varphi \leftarrow \text{sign}_i(\text{COMMIT}\|T)$
12: $\sigma \leftarrow \text{sign}_i(\text{VALUE}\|x\|T[i])$
13: send $\langle \text{COMMIT}, T, \varphi, x, \sigma \rangle$ to S
14: **return** OK

15: *read*(j)
16: $x \leftarrow \bar{x}$
17: send $\langle \text{SUBMIT}, \textit{READ}, j \rangle$ to S
18: **wait for** a message $\langle \text{REPLY}, V, \ell, \varphi', (y, \rho) \rangle$ from S
19: **if not** $\big([V = (0, \dots, 0)$ **or** $\text{verify}_\ell(\varphi', \text{COMMIT}\|V)]$
20: **and** $T \leq V$ **and** $T[i] = V[i]\big)$ **then halt**
21: **if not** $\big(V[j] = 0$ **or** $\text{verify}_j(\rho, \text{VALUE}\|y\|V[j])\big)$ **then halt**
22: $T \leftarrow V; T[i] \leftarrow T[i] + 1$
23: $\varphi \leftarrow \text{sign}_i(\text{COMMIT}\|T)$
24: $\sigma \leftarrow \text{sign}_i(\text{VALUE}\|x\|T[i])$
25: send $\langle \text{COMMIT}, T, \varphi, x, \sigma \rangle$ to S
26: **return** y

Algorithm 2 Lock-step protocol, algorithm for server S.

1: **state**
2: $X[i] \in \{0,1\}^* \times \{0,1\}^*$, // current state
3: initially $(\perp, \perp)$, for $i = 1, \dots, n$
4: $V[i] \in \mathbb{N}$, // current version
5: initially 0, for $i = 1, \dots, n$
6: $\ell \in \{1, \dots, n\}$, // client that completed the last operation
7: initially 1
8: $\omega \in \{0,1\}^*$, // sig. by C_ℓ for last operation
9: initially the empty string
10: **loop**
11: **wait for** receiving a message $\langle \text{SUBMIT}, o, j \rangle$ from some client C_i
12: **if** $o = \textit{READ}$ **then**
13: send $\langle \text{REPLY}, V, \ell, \omega, X[j] \rangle$ to C_i
14: **else**
15: send $\langle \text{REPLY}, V, \ell, \omega \rangle$ to C_i
16: **wait for** receiving a message $\langle \text{COMMIT}, T, \varphi, x, \sigma \rangle$ from C_i
17: $(V, \ell, \omega) \leftarrow (T, i, \varphi)$
18: $X[i] \leftarrow (x, \sigma)$

Description. Our shared memory consists of n SWMR registers $X_1, \dots, X_n$ with domain $\mathcal{V}$; client C_i may write only to X_i but read from any register. The domain $\mathcal{V}$ of a register is arbitrary. In practice, however, an array of fixed-size registers can provide consistent access to an array of arbitrarily large data sets through using a hash tree [14].

Every client locally maintains a timestamp that it increments during every operation. We call a vector of n such timestamps a *version vector* or simply a *version*; it acts as a vector clock for ordering operations. We define a partial order on version vectors. For two version vectors u and v, we say that u is *smaller than or equal to* v, denoted $u \leq v$, whenever $u[i] \leq v[i]$ for $i = 1, \dots, n$. We say that u is *smaller* than v, denoted $u < v$, if and only if $u \leq v$ and $u[i] < v[i]$ for some i.

The state of the client consists of a version vector T representing its most recently completed operation, together with a copy of its own data value $\bar{x}$. For simplicity of the protocol description, the client stores $\bar{x}$ and writes it back during every read operation.

The server S maintains an array X, representing the register values, where entry $X[i]$ represents X_i and is a pair of the form $(x_i, \sigma_i) \in \{0,1\}^* \times \{0,1\}^*$. The string x_i contains the actual value, and σ_i is a digital signature by C_i on the string $\text{VALUE}\|x_i\|t_i$, where t_i is a timestamp equal to C_i's own timestamp $T[i]$ at the time of completing the operation that wrote x_i. Furthermore, S keeps information related to the most recently executed operation: the version vector V of the operation, the identity c of the client performing the operation, and a digital signature φ by C_c on V.

When a client C_i invokes an operation, it sends the request to the server in a SUBMIT message. The server sends a REPLY message, containing the version vector V and the accompanying signature ω from the most recently completed operation. In a read operation for register j, the server also sends (x_j, σ_j), representing the current value of the register. The server then waits for another message from C_i and does not process any messages from other clients.

The client verifies that the reply contains valid data: the version V must be at least as big as its own version T, the i-th entry of V must correspond to the i-th entry of T, and the signature on $\text{COMMIT}\|V$ must be valid. In a read operation, the client also verifies the signature on the data value and the associated timestamp. When a client detects any inconsistency in the reply, it considers the server to be Byzantine and stops the execution. In practice, the client might generate an alarm in this situation and alert an operator to invoke a recovery procedure.

After C_i has successfully verified the reply, it adopts the received version V as its own version T, increments its timestamp $T[i]$, and signs the new version T, resulting in a signature φ. It also signs its own value together with $T[i]$. Then it sends a COMMIT message to S, containing the T, φ, its value x and the signature on $\text{VALUE}\|x\|T[i]$.

The server then stores T and φ as its version V and signature ω from the most recent operation, and updates $X[i]$ with the received value and signature.

Complexity. All messages sent in the protocol have size $O(n + |x| + \kappa)$, where $|x|$ denotes an upper bound on the length of the register values and κ denotes the length of a digital signature. Hence, the protocol uses network bandwidth economically. In particular, this improves the communication complexity of the bare-bones protocol of Mazières and Shasha [18] by an order of magnitude; their protocol achieves the same guarantees, but has communication complexity $\Omega(n^2 + n\kappa + |x|)$.

5. A CONCURRENT IMPLEMENTATION

In the lock-step protocol, when a correct server executes an operation o submitted by a client, it is not allowed to start processing any other request until o completes. This section extends the protocol to allow concurrent processing of independent operations, while maintaining $O(n)$ communication complexity. When the server follows the protocol and the execution is admissible, every client may complete its operations independently of the speed of other clients, unless the two operations depend on each other. A write operation never depends on another operation, whereas a read operation depends on the most recent write operation to the same register. Hence, if the server receives a write operation o and later a read operation o' from the same register, it blocks o' until o completes. As shown in Section 3.1, blocking is unavoidable for any protocol that emulates shared memory on a Byzantine server with fork-linearizability.

Algorithm 3 Concurrent protocol, algorithm for client C_i, part 1.

```
1: notation
2:    Strings = {0,1}* ∪ {⊥}; Clients = {1,...,n} ∪ {⊥}
3:    Opcodes = {READ, WRITE, ⊥}
4:    Operations = Clients × Opcodes × ℕ × Clients × Strings
         // lists of tuples from Operations and hash values
5:    OpHashSets = 2^{(o,h) | o∈Operations, h∈Strings}
6: state
7:    V_old[i] ∈ ℕ,                       // version vector of last operation
8:        initially 0, for i ∈ {1,...,n}
9:    M_old ∈ OpHashSets, initially empty       // list of missing proofs
10:   w̄ ∈ Strings, initially ⊥                  // most recently written value
11: write(w)
12:   w̄ ← w
13:   τ' ← sign_i(SUBMIT‖WRITE‖V_old[i]‖i)
14:   op ← (i, WRITE, V_old[i], i, τ')
15:   send ⟨SUBMIT, op⟩ to S
16:   wait for a message ⟨REPLY, S_info, C, P⟩ from S
17:       where S_info = (s, h_oS, h_xS, V_S, M_S, φ_S)
18:   (V_new, M_new) ← common(op, S_info, C, P)
19:   (V_old, M_old) ← (V_new, M_new)
20:   φ' ← sign_i(COMMIT‖H(op)‖H(w)‖H(V_new)‖H(M_new))
21:   send ⟨COMMIT, op, w, V_new, M_new, φ'⟩ to S
22:   return OK
23: read(l)
24:   w ← w̄
25:   τ' ← sign_i(SUBMIT‖READ‖V_old[i]‖l)
26:   op ← (i, READ, V_old[i], l, τ')
27:   send ⟨SUBMIT, op⟩ to S
28:   wait for a message ⟨REPLY, S_info, X_info, C, P⟩ from S
29:       where S_info = (s, h_oS, h_xS, V_S, M_S, φ_S)
30:        and X_info = (h_ox, x, V_x, M_x, φ_x)
31:   if not (verify_l(φ_x, COMMIT‖h_ox‖H(x)‖H(V_x)‖H(M_x))
32:           or V_x = (0,...,0)) then halt
33:   if (V_S, M_S) ≱ (V_x, M_x) then halt
34:   (V_new, M_new) ← common(op, S_info, C, P)
35:   if exists (o, E) ∈ M_new where o is a WRITE by client C_l then
36:       halt
37:   if exists (o, E) ∈ M_new where o is a READ by client C_l
38:       then if V_new[l] ≠ V_x[l] + 1 then halt
39:   else if V_new[l] ≠ V_x[l] then halt
40:   (V_old, M_old) ← (V_new, M_new)
41:   φ' ← sign_i(COMMIT‖H(op)‖H(w)‖H(V_new)‖H(M_new))
42:   send ⟨COMMIT, op, w, V_new, M_new, φ'⟩ to S
43:   return x
```

Algorithm 4 Concurrent protocol, algorithm for client C_i, part 2.

```
44: common(op, (s, h_oS, h_xS, V_S, M_S, φ_S), C, P)
45:   if not (verify_s(φ_S, COMMIT‖h_oS‖h_xS‖H(V_S)‖H(M_S))
46:           or V_S = (0,...,0)) then halt
47:   if not ((V_S, M_S) ≥ (V_old, M_old)
48:           and V_S[i] = V_old[i]) then halt
49:   if op is not the last operation in C then halt
50:   M_new ← M_S
51:   verify-proofs(P, M_new)
52:   V_new ← V_S
53:   for o = (c, oc, v, l, τ) in C do
54:       if not (verify_c(τ, SUBMIT‖oc‖v‖l)
55:               and v = V_new[c]) then halt
56:       V_new[c] ← V_new[c] + 1
57:       if o ≠ op then add (o, H(V_new)) to M_new
58:   if (V_new, M_new) ≯ (V_S, M_S) then halt
59:   if exists an operation by C_i in M_new then halt
60:   if exists a client C_j with more than one entry in M_new then halt
61:   return (V_new, M_new)
62: verify-proofs(P, M)
63:   for (c, o_c, h_xc, h_Vc, h_Mc, φ_c) ∈ P do
64:       if not verify_c(φ_c, COMMIT‖H(o_c)‖h_xc‖h_VS‖h_Mc) then
65:           halt
66:       if exists E s.t. (o_c, E) ∈ M
67:           if E ≠ h_Vc then halt
68:           remove (o_c, E) from M
```

The protocol for clients is presented in Algorithms 3 and 4, and the protocol for the server appears in Algorithm 5. Our protocol follows the general lines of the concurrent version of the SUNDR protocol [18], and improves the communication complexity from $\Omega(n^2)$ to $O(n)$, while providing the same guarantees. In the full version [4], we prove the following theorem.

Theorem 7. *The concurrent protocol emulates n SWMR registers on a Byzantine server with fork-linearizability.*

Description. We first describe the algorithm of the clients (all line numbers refer to Algorithms 3 and 4). Every client locally maintains a version vector V_{old}, which acts as a vector clock similarly to the lock-step protocol, and a list $\mathcal{M}_{old}$ of *missing proofs*, which holds information about operations that were concurrent to the client's previous operation. The client also maintains the latest written value $\bar{w}$.

A client invokes an operation by sending a SUBMIT message to the server, which includes an *announcement*, consisting of a tuple of the form (c, oc, v, l, τ), where c is the index of the client submitting the operation, oc is the operation code (READ or WRITE), v is the sequence number of this operation, l is the register being read (relevant only for *read* operations), and τ is a signature. The server sends back a REPLY message, containing a data structure S_{info} representing the operation which committed with the largest version vector thus far. In the client code, this operation is denoted by o_S, the associated version vector by V_S, and the associated list of missing proofs by $\mathcal{M}_S$.

Announcements are used by the server to notify a client invoking a new operation *op* about all uncommitted operations that started after o_S had committed and that were scheduled by the server before *op*. Specifically, the announcement of a new operation is appended to a list $\mathcal{C}$ of *concurrent operations* that is maintained by the server, and this list is sent in the REPLY message to clients. The last operation in $\mathcal{C}$ received from the server during operation *op* is always *op* itself. A client C_i, executing *op* computes its new version vector by taking V_S as a base, and incrementing the entries corresponding to every client C_c having an operation in $\mathcal{C}$ (lines 52-57). Since *op* is in $\mathcal{C}$ as well, the i-th entry of the vector is always incremented as in the lock-step protocol. When encountering an operation by client C_c in $\mathcal{C}$, C_i first checks that V_S reflects all previous operations of that client. Otherwise the server must be faulty and the execution is halted.

In the algorithm as described thus far, a client cannot know whether the operations in $\mathcal{C}$ are presented in the same order to other clients, and whether these operations are really concurrent, i.e., that the server did not just retransmit some old announcements. To solve this problem, every version vector is augmented by a *list of missing proofs* $\mathcal{M}$, as described next.

When a client C_i executing an operation o encounters an announcement of an operation o' in $\mathcal{C}$, it is able to predict the version vector V' that should be committed by o', if o and o' receive consistent information about concurrent operations from the server. Operation o then includes the pair $(o', H(V'))$ in its $\mathcal{M}_{new}$, which it sends to the server in its COMMIT message. When o' commits, it signs and sends its real new version vector V_{new} to the server. The server then uses the pair $(o', H(V_{new}))$ as a proof to other clients, which they use to remove $(o', h_{V'})$ from $\mathcal{M}$, by comparing the "proof," $H(V_{new})$, to the expected version vector hash, $h_{V'}$,

in *verify-proofs* (line 51). We thus call operations in $\mathcal{M}$ *unverified*. Proofs are transmitted in a set $\mathcal{P}$ in the REPLY message. Since clients execute operations sequentially, no operation by client C_i could be unverified during *op* (line 59), and no client can have more than one unverified operation (line 60).

As mentioned above, when a client C_i commits, it sends to the server $\mathcal{M}_{new}$, a list that includes all operations known to C_i that remain unverified. The server stores all information included in C_i's latest COMMIT message. Additionally, $\mathcal{M}_{new}$ is saved by C_i in the local variable $\mathcal{M}_{old}$. When a new operation o by C_i starts, it receives $\mathcal{M}_S$ (the $\mathcal{M}_{new}$ list sent to the server as o_S committed). If o is a *read* operation of register l, C_i additionally receives $\mathcal{M}_x$ which is the $\mathcal{M}_{new}$ list included in C_l's latest COMMIT message. The client then initializes $\mathcal{M}_{new}$ for the new operation with $\mathcal{M}_S$ (line 50). Next, using *verify-proofs* procedure, the client verifies (i.e., removes from $\mathcal{M}_{new}$) all operations that match a proof included in $\mathcal{P}$. If there is a proof in $\mathcal{P}$ for some operation o such that the hash of the real (signed) version vector committed by o (as appears in $\mathcal{P}$) does not match the one saved with o in $\mathcal{M}_{new}$, the server must be Byzantine and the execution is halted. Next, as was already mentioned, all operations that appear in $\mathcal{C}$ are added to $\mathcal{M}_{new}$ together with a hash of the version vector the operation is expected to sign upon completion (line 57).

In order to simplify the presentation of the protocol and the proof, we define an order on version-vector/list-of-missing-proof pairs. A similar order, but for different data structures, was used by Mazières and Shasha [18].

Definition 7 (Order of version/list-of-missing-proofs pairs). Consider an operation o which commits with version vector V and a list of missing proofs $\mathcal{M}$, and an operation o' which commits with V' and $\mathcal{M}'$. We say that $(V, \mathcal{M}) \leq (V', \mathcal{M}')$ if the following conditions hold:

1. $V \leq V'$ (order on version vectors from Section 4); and
2. For each $(o'', E) \in \mathcal{M}'$ where $o'' = (c, oc, v, l, \tau)$, one of the following holds:
   (a) $V[c] < V'[c]$
   (o was scheduled before o'')
   (b) $V[c] = V'[c]$ and $(o'', E) \in \mathcal{M}$
   (o'' was unverified during o)
   (c) $V[c] = V'[c]$ and $E = H(V)$
   (o and o'' are the same operation).

The protocol makes sure that all operations of the same client are ordered according to this relation, and that a *read* operation is ordered with the *write* operation that wrote the value returned by the *read*. This is achieved by lines 47 and 58 and additionally by line 33 for a *read*. If a faulty server conceals operation o with associated version V from a later operation o' whose associated version is V', then it can be shown that $V \not\geq V'$ and $V' \not\geq V$. The concurrent protocol guarantees that no later operation o'' can sign a version V'' and a missing proof list $\mathcal{M}''$ s.t. $(V'', \mathcal{M}'') \geq (V, \mathcal{M})$ and $(V'', \mathcal{M}'') \geq (V', \mathcal{M}')$ (where $\mathcal{M}$ and $\mathcal{M}'$ are the missing proof lists committed by o and o' respectively).

A *read* operation by client C_r from register l receives additional information in the REPLY message from the server. Specifically, it receives X_{info} committed by C_l (the client that writes to register l), which includes the written data, x, the version vector of the operation that wrote the data, V_x, and the missing proof list $\mathcal{M}_x$. Although client C_r executing the *read* cannot generally know if the server returns data corresponding to the latest preceding committed operation by C_l, C_r can make sure that it itself is not aware of a later operation by C_l. If there are no unverified operations by C_l, then the data returned by C_r must have been written by the last operation of C_l as known to C_r. Specifically, the check on line 39 makes sure that $V_x[l] = V_r[l]$, where V_r is the version vector committed by the reader. On the other hand, if there is an unverified operation by C_l, then this operation can only be a *read* if the server follows the protocol, which is checked by line 35. In this case, V_x should be the operation of C_l which immediately preceded the concurrent *read*. This is assured by line 38 which makes sure that $V_r[l] = V_x[l] + 1$.

Algorithm 5 Concurrent protocol, algorithm for server S.

```
1:  state
2:     C ∈ Operations*, initially empty    // list of concurrent operations
3:     X[i] = (op_i, x_i, V_i, M_i, φ_i)                      // current state
4:              ∈ Operations × Strings × ℕ^n × OpHashSets × Strings,
5:        initially (⊥, ⊥, (0, ..., 0), ∅, ⊥), for i ∈ {1, ..., n}
6:     c ∈ Clients, initially 1.
7:  upon receiving a message ⟨SUBMIT, op⟩ from C_i,
8:                      where op = (i, oc, v, l, τ):
9:     if ((oc = READ) and exists op', a WRITE operation by C_l,
10:        s.t. [op' ∈ C or (op', *) ∈ M_c]) then
11:          wait until op_l = op'
12:    append op to the end of C
13:    P ← ∅
14:    for each o by client C_j s.t. (o, *) ∈ M_c do
15:       if (op_j = o) then
16:          add (j, op_j, H(x_j), H(V_j), H(M_j), φ_j) to P
17:    if oc = READ then
18:       send ⟨REPLY, (c, H(op_c), H(x_c), V_c, M_c, φ_c),
19:             (H(op_l), x_l, V_l, M_l, φ_l), C, P⟩ to C_i
20:    else
21:       send ⟨REPLY, (c, H(op_c), H(x_c), V_c, M_c, φ_c), C, P⟩ to C_i
22: upon receiving a message ⟨COMMIT, op, x, V, M, φ⟩ from C_i:
23:    X[i] ← (op, x, V, M, φ)
24:    if (V > V_c) then
25:       c ← i
26:       remove from C the prefix of operations up to (including) op
```

We now describe the server code (all line numbers refer to Algorithm 5). The server maintains in $X[i]$ all information committed by the last operation of client C_i. It additionally maintains the list $\mathcal{C}$ of concurrent operations. The server stores in c the identifier of the client that committed with the largest version vector out of all committed operations thus far. The server code consists of two procedures (lines 7–21 and lines 22–26), which operate on common variables. Only one of the procedures is allowed to run at any given time, but if the processing of an operation is blocked on line 11, another operation can be processed meanwhile, and when the blocking condition is satisfied, processing of the blocked operation will be able to resume when no process executes either procedure. The queue of pending operations, i.e., which wait for a permission to enter one of the procedures, is implicitly managed as a FIFO queue. When a REPLY message is sent to a client C_i for its operation o, the list $\mathcal{C}$ always includes o as its last operation. When an operation o commits and has the largest version vector out of all operations that committed thus far, the server updates c. Furthermore, S may delete the prefix of $\mathcal{C}$ up to o. Intuitively, this is done since all important information about this prefix can be deducted from information committed by o.

The first procedure (lines 7–21) deals with the receipt of a SUBMIT message from a client C_i. If the submitted operation is a *read* from register l, and there is a *write* operation by client C_l (the only client that writes to l) that was received by the server before the *read* but which COMMIT message was not yet received, then C_i's *read* is blocked until this operation by C_l completes. Note that if the server is correct, the operation must be either in $\mathcal{C}$ or in $\mathcal{M}_c$.

The server then examines the list of missing proofs $\mathcal{M}_c$ that it sends to C_i, and collects in $\mathcal{P}$ all necessary "proofs" it has for operations in this list. These are tuples containing an operation and the hash of the actual version vector committed by the operation. Then, S sends a REPLY message, which includes the data committed by operation o_c which had the largest version vector thus far (was scheduled later than any other operation that committed), including its associated list of missing proofs $\mathcal{M}_c$. The REPLY message also contains the data requested by the client, the list $\mathcal{C}$ of concurrent operations, and $\mathcal{P}$.

When a COMMIT message is received from client C_i (lines 22–26), the received information is saved in $X[i]$, and if the committed operation has a greater version vector then the operation committed by c, then c becomes i and the prefix of $\mathcal{C}$ up to the newly committed operation is deleted. The server's code does not include any checks for correctness of the client responses because we assume that clients are correct. Such checks could be added easily (the server can calculate precisely with what version vector and missing proof list a client is supposed to commit).

Complexity. During an execution of operation *op* by client C_i, all operations in $\mathcal{C}$ (other than *op* itself) are inserted into $\mathcal{M}_{new}$ and then C_i checks that there is at most one operation by each client in $\mathcal{M}_{new}$, and no operations by C_i. Thus, the size of $\mathcal{C}$ is bounded by the size of $n-1$ operations, i.e. $O(n\kappa)$, where κ denotes the maximal length of digital signatures and hashes. For the same reason, any list of missing proofs can contain at most $n-1$ operation/hash pairs, thus its size is $O(n\kappa)$ as well. Furthermore, $\mathcal{P}$ has at most $n-1$ entries because $\mathcal{M}$ has at most $n-1$ entries, and the size of $\mathcal{P}$ is also $O(n\kappa)$. Using $|x|$ to denote an upper bound on the length of register values, the communication complexity of the protocol when run with a correct server is therefore $O(n\kappa + |x|)$.

A faulty server could obviously send larger messages. Because most data sent by S must successfully verify a signature issued by a client, together with including some additional checks, we can actually guarantee a stronger notion: whenever a client completes an operation, its communication complexity was no more than $O(n\kappa + |x|)$. The additional checks in the *verify-proofs* procedure should check that no operation is included more than once in $\mathcal{P}$ and that every operation o_c with a "proof" in $\mathcal{P}$ is actually needed because it is in $\mathcal{M}$.

This improves the complexity of the concurrent SUNDR protocol [18] by an order of magnitude; the protocol provides the same guarantees, in the same model, but has communication complexity $O(n^2 + n(\kappa + |x|))$, which remains its worst-case complexity even with the bandwidth optimizations suggested therein.

Observe that writing during *read* operations is not necessary, however if *read* operations do not update the version of the data, a reader of the data must be able to distinguish stale data from data that was not updated by *read* operations. This can be solved by maintaining two version vectors instead of one, one for the number of *write* operations only, while the other counts the total number of operations. We have avoided this for simplifying the presentation.

Acknowledgments

We thank Idit Keidar, Gregory Chockler, Eshcar Hillel, Petr Kouznetsov, and Marko Vukolić for many helpful discussions and valuable comments.

6. REFERENCES

[1] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley, second edition, 2004.

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[4] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. TR RZ3688, IBM Research, Apr. 2007.

[5] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT 2003*.

[6] D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *Proc. 26th IEEE Symposium on Security & Privacy*, 2005.

[7] K. Fu, F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, Feb. 2002.

[8] K. E. Fu. Group sharing and random access in cryptographic storage file systems. Master Thesis, MIT LCS, 1998.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS 2003*, pages 522–529.

[11] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998.

[13] L. Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 86–101, 1986.

[14] J. Li, M. Krohn, D. Mazires, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136, 2004.

[15] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

[16] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, Sept. 1989.

[17] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *DISC 2002*, pages 311–325.

[18] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *PODC*, pages 108–117, 2002.

[19] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *DISC*, pages 254–268, 2006.

[20] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.