RZ 3688 (# 99698) 04/18/07
Computer Science 30 pages

Research Report

Efficient Fork-Linearizable Access to Untrusted Shared Memory

Christian Cachin and abhi shelat

Email: {cca, abs}@zurich.ibm.com

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Alexander Shraer

Email: shralex@cs.technion.ac.il

Department of Electrical Engineering
Technion
Haifa 32000
Israel
shralex@cs.technion.ac.il.

Efficient Fork-Linearizable Access to Untrusted Shared Memory

Christian Cachin* abhi shelat* Alexander Shraer†

April 18, 2007

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.

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 valid write operations to another user.

*IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. {cca, abs}@zurich.ibm.com.

†Department of Electrical Engineering, Technion, Haifa 32000, Israel. shralex@cs.technion.ac.il.

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 [MS02] 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 [MS02] 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* [HW90] 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 [OR06] 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 wait-free [Her91]. This result explains a seeming limitation 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 must assume that clients do not fail. Next, we show 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. Furthermore, we give a "global" definition of fork-linearizability in terms of a single sequential history of operations, of which the clients observe linearizable subsequences.

Second, we provide an efficient protocol for emulating fork-linearizable shared memory on an untrusted server. Our protocol is inspired by the SUNDR protocol [MS02] 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 on the vector timestamps of all clients.

Analogously to the work about SUNDR [MS02], we present two protocols: a protocol that proceeds in "lock-step," because 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 [MS02] 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 [MS02] has been implemented in a practical distributed filesystem called SUNDR [LKMS04], 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 [OR06] 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. They consider a situation where both objects are stored on an untrusted server and the clients want to achieve a given consistency property for the high-level file object. They are able to characterize necessary and sufficient conditions for the consistency notions required of the file-key access and file-data access protocols.

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.* [BEG+94] to investigations on "incremental" hash functions [CDvD+03], motivated by the goal to construct secure processors with untrusted main memory [CSG+05]. Hash trees have also been used in several filesystems, starting with Fu's work [Fu98] and the SFSRO filesystem [FKM02].

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 [MAD02, ACKM06]. 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 fork-sequential-consistency 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 [LT89, Lyn96]. 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" [PSL80] or "NR-arbitrary" faults [JCT98].

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 consist 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 [Lam86], 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 [AW04]: *sequential consistency* and *linearizability*. The third one, *fork-linearizability*, has been introduced by Mazières and Shasha [MS02] 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 [HW90], which guarantees that all operations occur "atomically," i.e., appear to be executed at a single point in time.

Definition 2 (Linearizability [HW90]). 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 observed by all clients together.

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. 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 take place. As long as C_j and C_i do not communicate with each other, none of them can detect this violation, and thus neither definition can be satisfied.

Mazières and Shasha [MS02] called such behavior a *forking attack*. They postulate that *forking two (sets of) clients* by introducing discrepancies between the events observed by two 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* [MS02] 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 [OR06] 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. [JCT98]. 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 the of the 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 [Lyn96, AW04].

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 in on concurrency and correctness and not 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 the invocation contains $x \in L$, 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 the Notion of 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 wait-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 in Fork-Linearizable Byzantine Emulations

It is well-understood that lock-based algorithms for synchronizing concurrent access to shared data are problematic and that *wait-free* [Her91] 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 include *lock-freedom*, *fw-termination* [ACKM06], and *obstruction freedom* [HLM03]. 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.

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 $w_1^1 = \mathit{write}_1^1(X_1, u) \rightarrow \text{OK}$. 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 that, C_2 invokes an operation $r_2^1 = \mathit{read}_2^1(X_1)$ that returns u. Subsequently, C_1 executes a write operation $w_1^2 = \mathit{write}_1^2(X_1, v) \rightarrow \text{OK}$, 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., reads $r_2^1, \ldots, r_2^{z-1}$ 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.

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.

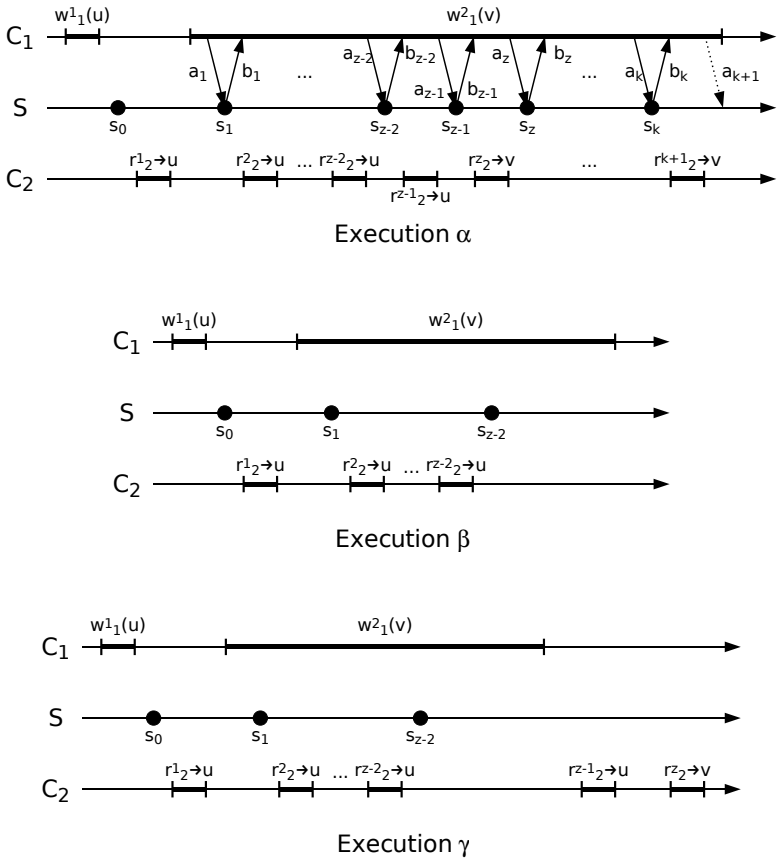

Figure 1: Three executions α, β, and γ with $z > 2$, as described in the text. C_1 cannot distinguish execution γ from β and C_2 cannot distinguish γ from α.

Note that C_1 cannot distinguish execution γ from β and C_2 cannot distinguish γ from a prefix of α, and both are executions that satisfy linearizability 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 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 [MS02] 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 [Her91, JCT98] 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 *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.

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

Proof. Notice that global fork-linearizability trivially implies fork-linearizability. To prove the reverse implication, we first construct a history π such that every π_i according to Definition 4 is a subsequence of π. We start by appending sequential operations to π according to the common prefix of all sequences π_i, for $i = 1, \ldots, n$. When the forking tree branches, we continue simultaneously along both branches. We always take the next event from any of the branches under consideration according to real-time order and append it to π. We continue with this procedure and eventually proceed simultaneously along all n histories π_i, until π contains all events of σ. Note that the only difference between the resulting sequence π and the global history π of Definition 6 is that the latter must be sequential.

We describe a procedure that turns π into a sequential history π' that is equivalent to π with respect to all conditions of fork-linearizability. The difference mentioned above means that there exists some operation in some subsequence π_i that is concurrent to one or more operations that are contained in different subsequences. Let o_i be the first such operation according to the order of π. W.l.o.g. let $\mathcal{C} = \{c_1, \ldots, c_k\}$ be the set of all operations concurrent to o_i that are invoked after o_i in π, where $c_1 \in \pi_{i_1}, \ldots, c_k \in \pi_{i_k}$ such that operation c_j is executed by client C_{i_j}. For the moment, assume that o_i is complete.

Fork-linearizability implies that o_i is not contained in any π_{i_j} for $j = 1, \ldots, k$ because π_{i_j} is sequential. For the same reason, no operation that occurs in π_i after o_i is contained in any π_{i_j} for $j = 1, \ldots, k$ and, vice versa, no operation that occurs in some π_{i_j} after c_j for $j = 1, \ldots, k$ is contained in π_i.

Let $c^* \in \pi_{i^*}$ be the operation from $\mathcal{C}$, whose invocation in π occurs first, i.e., before the invocation of any other operation in $\mathcal{C}$. Hence, the invocation of o_i occurs in π immediately before the invocation of c^* and there is no other event between them. In turn, the invocation of c^* occurs somewhere before the completion of o_i. Modify π to π' by moving the completion of o_i to an earlier point in time, inserting it between the invocation of o_i and the invocation of c^*.

Notice that o_i is now sequential and no longer concurrent with any operation in the resulting history π'. We claim that π', together with the same subsequences $\pi_1, \ldots, \pi_n$ as before, satisfies all conditions of fork-linearizability according to Definition 4. First, π' preserves the real-time order of σ because o_i only terminates earlier and this can only have enlarged the set of operations that are invoked after o_i. For the second condition, note that any subsequence π_j that contains o_i is not affected by the modification because π_j was already sequential and did not contain any events between the invocation of o_i and its completion, i.e., $j \notin \{i_1, \ldots, i_k\}$; hence, all such π_j may remain the same as before and satisfy Definition 4 with respect to π', just as with π. But any other subsequence π_c is not affected either because π_c does not even contain o_i.

If π' is sequential, the theorem holds. Otherwise, repeat this step and use induction on the sequence of modifications to obtain a sequential permutation of all events. Because at least one pair of concurrent operations is eliminated in each step, the induction terminates. □

3.3 Comparing Fork-Linearizability 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 $write_1(X_1, v) \rightarrow \text{OK}$ 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 $write_2(X_2, v) \rightarrow \text{OK}$ 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 and contradicts the assumption that P always produces sequentially consistent executions. □

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 of Fork-Linearizable Shared Memory

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 [MS02], 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.

Description. Our shared memory consists of n SWMR registers $X_1, \ldots, X_n$ 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 [LKMS04].

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 $i = 1, \ldots, n$. 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 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 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 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.

The detailed protocol is shown in Algorithms 1 and 2.

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

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

write(x)
  $\bar{x} \leftarrow x$
  send $\langle$SUBMIT, ***WRITE***, $\perp\rangle$ to S
  wait for a message $\langle$REPLY, $V, \ell, \varphi'\rangle$ from S
  if not $\big([V = (0, \ldots, 0)$ **or** verify$_\ell(\varphi', $COMMIT$\|V)\big]$ **and** $T \leq V$ **and** $T[i] = V[i]\big)$ **then**
    halt
  $T \leftarrow V$; $T[i] \leftarrow T[i] + 1$
  $\varphi \leftarrow$ sign$_i($COMMIT$\|T)$
  $\sigma \leftarrow$ sign$_i($VALUE$\|x\|T[i])$
  send $\langle$COMMIT, $T, \varphi, x, \sigma\rangle$ to S
  return OK

read(j)
  $x \leftarrow \bar{x}$
  send $\langle$SUBMIT, ***READ***, $j\rangle$ to S
  wait for a message $\langle$REPLY, $V, \ell, \varphi', (y, \rho)\rangle$ from S
  if not $\big([V = (0, \ldots, 0)$ **or** verify$_\ell(\varphi', $COMMIT$\|V)\big]$ **and** $T \leq V$ **and** $T[i] = V[i]\big)$ **then**
    halt
  if not $\big(V[j] = 0$ **or** verify$_j(\rho, $VALUE$\|y\|V[j])\big)$ **then**
    halt
  $T \leftarrow V$; $T[i] \leftarrow T[i] + 1$
  $\varphi \leftarrow$ sign$_i($COMMIT$\|T)$
  $\sigma \leftarrow$ sign$_i($VALUE$\|x\|T[i])$
  send $\langle$COMMIT, $T, \varphi, x, \sigma\rangle$ to S
  return y

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

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

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 [MS02] by an order of magnitude; their protocol achieves the same guarantees, but has communication complexity $\Omega(n^2 + n\kappa + |x|)$.

Analysis. The rest of this section is devoted to the analysis of the lock-step protocol. Recall from the protocol that every client stores in T a "current" version that was computed during its most recent operation. At the end of every operation, the client sends T to the server in the COMMIT message; we say that this version is *associated* to the operation.

We first prove two lemmas about the versions associated to operations that causally influence each other. The first lemma shows that the version vectors associated to the operations of any single client are totally ordered. Recall that by the assumption of sequential interaction, every client executes its operations sequentially.

Lemma 6. *Let o and o' be two operations completed by a client C_i with associated versions v and v', respectively, such that o precedes o'. Then $v[i] < v'[i]$ and $v < v'$.*

Proof. Observe that C_i stores its current version in T. At the start of any operation, T is equal to the version associated to the most recently completed operation by C_i. Suppose $T = v_0$ when o' is invoked. During the execution of o', the checks that $T \leq V$ in lines 8 and 19 of Algorithm 1 ensure that T is updated on line 10 and 23 only to a V that is at least as big as v_0. The remaining code modifies the current version T only by incrementing the i-th entry. Since the version v' associated to o' reflects the value of T immediately before o' ends, $v'[i] > v_0[i]$ and v' is bigger than v_0. By induction on C_i's operations, it follows that v' is bigger than the version associated to any o executed by C_i that precedes o'. □

The second lemma shows that the version associated to a write operation is smaller than the version associated to any read operation that returns the written value.

Lemma 7. *Let $r = \mathit{read}_i(j) \rightarrow x$ be a completed read operation of some client C_i with associated version v_r such that $x \neq \perp$. Let w be the write operation by C_j that wrote x, and let v_w be the version associated to it. Then $v_w < v_r$.*

Proof. Note first that w is well-defined because the condition $x \neq \perp$ ensures that C_j has executed some write operation and because each written value is unique.

Let $\langle \text{REPLY}, V, \ell, \varphi, (x, \sigma) \rangle$ be the message received by C_i during r such that σ verifies successfully. Although read operations by C_j occurring after w also update x and σ, it is easy to see from the client code that the written value has not changed since C_j's last write operation. Hence, we may assume in the rest of the proof that C_j did not execute any read operation after w and that it computed σ during a write operation.

According to the code for the writer, it must be $V[j] > 0$. On line 23, C_i assigns V to its current version and later changes it only by incrementing the i-th entry. Therefore, $v_r > V$. By the algorithm and by the integrity of the signature scheme, only C_j may increase the j-th entry in a version vector that passes signature verification, and since C_j increases the entry in every operation, there is exactly one version signed by C_j with the j-th entry equal to $V[j]$. This version is v_w, and we need to prove that $v_w < v_r$.

During execution of r, client C_i receives V from S signed by C_ℓ. Let o be the operation by C_ℓ whose associated version is V. If $c = j$, then because C_j signed the value x together with $V[j]$ and C_j signs only one version with the j-th entry equal to $V[j]$, we get that $V = v_w$. Hence, $v_r > v_w$ as required because $v_r > V$. If $c \neq j$, consider the execution of o. Client C_ℓ has received a signed version v' from S, which is associated to some operation o' by some $C_{\ell'}$. Since $c \neq j$, operation o does not change the j-th entry in v' and therefore $v'[j] = V[j]$. If $c' = j$, then $v' = v_w$ since there is only one operation by C_j with an associated version whose j-th entry is equal to s. It follows that $v_w = v' < V < v_r$ as required. Otherwise, we may continue like this and trace the execution of the operations, producing the sequence of associated version vectors $v_r, V, v', \ldots$ backwards in time, until we encounter an operation by C_j. Since no operation in the sequence considered up to here has changed the j-th entry of the version with respect to v_r, the version signed by C_j must have its j-th entry equal to $V[j]$. And because C_j signs only one such version, this must be v_w. Because the partial order on version vectors is transitive, it follows $v_w < v_r$.

We still have to show that a version vector signed by C_j appears somewhere in the above sequence. Towards a contradiction, assume that there is no such version. Observe that any two vectors in the sequence differ by 1 in exactly in one entry. Since all vectors contain only non-negative values and the sequence starts with the all-zero vector according to the protocol, the number of vectors in the sequence is clearly finite. But since $V[j] > 0$ at the end of the sequence and because only C_j may have increased the j-th entry, there exists a version signed by C_j. □

The main result about the lock-step protocol is the following theorem.

Theorem 8. *The lock-step protocol consisting of Algorithms 1 and 2 emulates n SWMR registers on a Byzantine server with fork-linearizability.*

Proof. Let σ be the sequence of events observed by the clients in the protocol. We construct a *sequential* permutation π of σ and show that it preserves the real-time order of σ and that there exists a suitable subsequence of π for every client satisfying the definition of fork-linearizability, according to Definition 6.

Note that there might be invocations in σ that have no matching response. But because the clients interact sequentially with the shared memory, there can be at most one such incomplete operation per client and n in total. Whatever the order of π, these operations can easily be added at the end of π because they do not influence the clients in any way, since their responses are missing. Hence, we assume in the following that σ is complete.

We construct the sequential execution π from a total order of all events in σ. Observe that π is sequential and the invocation and response events of every operation immediately follow each other.

Hence, we can speak of the order of operations in π. We order σ as follows:

1. Sort the operations in σ by the order on their associated version vectors;
2. Sort any yet unsorted sequential operations by their real-time order in σ ;
3. Sort any yet unsorted operations according to the real-time order of their completion event.

We also construct the subsequences π_i: First, we include in π_i all operations of client C_i. Next, for each $o \in \pi_i$ we add to π_i all operations $o' \in \sigma$ whose associated version is less than or equal to the version associated to o.

Claim 8.1. *Let o and o' be two operations in σ such that o precedes o'. Then, o precedes o' in π.*

Proof of Claim 8.1. Let the versions associated to o and o' be v and v', respectively. Since o precedes o' in σ, note that the first two rules of the sort order imply the claim except in case $v > v'$. We show that this does not occur by constructing a contradiction in that case.

Consider operation o. The client executing o receives a signed version v_1 from S and v_1 differs from v by -1 in exactly one entry. Consider the operation o_1 to which v_1 is associated. Clearly, o does not precede o_1 because o_1 must already have been invoked before o completed. Continuing like this with o_1, we build a sequence of operations, where the version vectors associated to every two neighboring operations differ in one entry by -1. The sequence is finite and ends with an operation in which the client receives the all-zero vector from S. By induction, o completes in σ after the invocation of any operation in the sequence. But if v' were smaller than v, then o' would occur in the sequence, contradicting the assumption that o' is invoked after o completes. □

Claim 8.2. *Let $r = read_i(j) \rightarrow x$ with $x \neq \perp$ be a completed read operation of some client C_i in some π_k, with associated version v_r; let w be the write operation of C_j that wrote x. Then:*

1. *Operation w is in π_k; and*
2. *There is no write operation by C_j subsequent to w in π_k that completes before r is invoked.*

Proof of Claim 8.2. As in the proof of Lemma 7, we assume w.l.o.g. that C_j issued no read operations after w and before the completion of r.

The first statement is easy to see. Lemma 7 implies that the version v_w associated to w is less than v_r. Operation w is therefore included in π_k by construction.

For the proof of the second statement, suppose towards a contradiction that there is another write operation w' by C_j with associated version v'_w in π_k such that w' is invoked after w completes and such that w' precedes r in σ. We have three sequential operations w, w', and r, occurring in that order in σ.

We first prove that v_r and v'_w are not sorted according to the order on version vectors by showing that $v_r[j] < v'_w[j]$ and $v_r[i] > v'_w[i]$. Then we extend this to show that $v'_w \notin \pi_k$.

Suppose C_i received a vector V and a tuple (x, σ) in the REPLY message from S during r. Since $i \neq j$, client C_i does not change the j-th entry in its version and therefore $v_r[j] = V[j]$. Note that by definition of w, we have also $v_w[j] = V[j]$. Moreover, since w and w' are both operations by C_j, and o_w precedes o'_w, we have $v'_w[j] > v_w[j]$ according to Lemma 6, and therefore

$$v_r[j] = v_w[j] < v'_w[j]. \tag{1}$$

On the other hand, observe that C_i is the only client that may increment the i-th entry in version vectors accompanied by valid signatures, and it does that for every operation. Hence, the version vector $\bar{V}$ that C_i receives from S during r satisfies $\bar{V}[i] = v_r[i] - 1$. Moreover, the i-th entry in any version

vector accompanied by a valid signature that was observed by a client before the invocation of r was less than $v_r[i]$. In particular, this applies also to v'_w and we have

$$v_r[i] > v'_w[i]. \tag{2}$$

Since r and w' are included in π_k, by construction of π_k there exist operations o and o' by C_k with minimal associated version vectors v and v', respectively, such that $v_r \leq v$ and $v'_w \leq v'$. Since every two operations by C_k are ordered, either $v \leq v'$ or $v > v'$. W.l.o.g. assume $v \leq v'$. Therefore, $v' \geq v'_w$ and $v' \geq v_r$.

Since C_i is the only client that may increment the i-th entry of a version vector, in order for v' to satisfy $v'[i] \geq v_r[i]$, there must be a sequence of operations and associated version vectors as in the proof of Lemma 7, starting with o_r and ending in o', in which every two adjacent version vectors differ by 1 in one entry.

Since $v_r[j] < v'_w[j]$ by (1), and because no other client than C_j may increment the j-th entry of a version vector, no operations invoked by C_j are in this sequence of operations; otherwise, C_j would notice (by checking $T[i] = V[i]$ on line 8) that the j-th entry in the version sent by the server is $v_r[j] < v'_w[j]$. But C_j expects this to be equal to $T[j] = v'_w[j]$, after having completed o'_w. Because C_j is not in this sequence, all version vectors contained in it, including v', have their j-th entry equal to $v_r[j]$. Hence, $v'[j] = v_r[j] < v'_w[j]$ and we know that $v'[i] \geq v_r[i] > v'_w[i]$ from (2). Therefore, v' is incomparable with v'_w, which contradicts the fact that o' was chosen such that $v' \geq v'_w$ and shows that $v'_w \notin \pi_k$. □

It remains to show that π and all π_i satisfy the properties of fork-linearizability: Claim 8.1 shows that π preserves the real-time order of σ; furthermore, properties 2(a) and 2(c) of fork-linearizability are immediate by construction of π_i, and property 2(b) follows from Claim 8.2. This shows that σ is fork-linearizable with respect to n SWMR registers.

As the last step in the proof of the theorem, we have to establish that the protocol is live when run with a correct server. The key observation is that only C_i updates the i-th entry of the server's version V, but does that for every operation. Hence, after every operation, every register value $X[i] = (x_i, \sigma_i)$ of S contains a valid signature σ_i on VALUE$\|x_i\|V[i]$. From this it is easy to verify that every client operation eventually completes. □

5 A Concurrent Implementation of Fork-Linearizable Shared Memory

In the lock-step protocol, when a correct server executes an operation o submitted by a client, it is not allowed to accept any other request until o is completed. 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, but a read operation depends on the most recent write operation to the same register. Hence, the server blocks a read operation o if it has received a concurrent write operation o' before o until o' completes. As shown in Section 3.1, this is unavoidable for any protocol that emulates shared memory on a Byzantine server with fork-linearizability.

The protocol for clients is presented in Algorithms 3 and 4, and the protocol for the server appears in Algorithm 5. Our protocol is derived from the concurrent version of the SUNDR protocol [MS02], and improves the communication complexity from $\Omega(n^2)$ to $O(n)$, while providing the same guarantees.

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} and a list $\mathcal{M}_{old}$ of *missing proofs* that were created by its most recent operation. This version acts as a vector clock, similarly to the lock-step protocol, but $\mathcal{M}_{old}$ is a new data structure that collects information on the operations that were concurrent to the most recent operation. The client also maintains the latest written value $\bar{w}$.

A client submits an operation with a SUBMIT message, which includes an *announcement* of the operation, 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 greatest 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 $\mathcal{M}_S$.

This announcement is then used by the server to notify 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, a new operation announcement 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 45-49). 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 now augmented by a *list of missing proofs* $\mathcal{M}$. When a client commits an operation o, it includes in $\mathcal{M}_{new}$ a pair $(o', h_{V'})$ for every announced operation o' that the server intends to schedule before o, and whose COMMIT message did not arrive at the server before the REPLY message for o was sent.

When a client C_i executing an operation op encounters an announcement of an operation o' in $\mathcal{C}$, it can calculate the version vector V' that will 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 $\mathcal{M}_{new}$. When o' commits, it sends its real new version vector V_{new} signed 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 44). 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 51), and no client can have more than one unverified operation (line 52).

When a client C_i commits, it sends to the server a list $\mathcal{M}_{new}$ of operations that C_i is aware of that remain unverified. All information committed by the last operation of C_i is stored by the server. Additionally, $\mathcal{M}_{new}$ is saved by C_i in a local variable $\mathcal{M}_{old}$. When a new operation o by a C_i starts, it receives $\mathcal{M}_S$ (the $\mathcal{M}_{new}$ list sent to the server during o_S). If o is a *read* operation of register l, C_i also receives $\mathcal{M}_x$ which is the $\mathcal{M}_{new}$ that was stored by the server for client C_l. The client then initializes $\mathcal{M}_{new}$ for the new operation with $\mathcal{M}_S$ (line 43). It runs the *verify-proofs* procedure, which removes all unverified operations that match a proof received in the $\mathcal{P}$ from $\mathcal{M}_{new}$. If a proof arrives for operation o such that the hash of the real version vector committed by o 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 49).

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 [MS02].

Definition 7 (Order of version-vector/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. $V \leq V'$ (according to the 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) $V[c] < V'[c]$ (i.e., o was scheduled before o'')
   (b) $V[c] = V'[c]$ and $(o'', E) \in \mathcal{M}$ (i.e., o'' was unverified during o)
   (c) $V[c] = V'[c]$ and $E = H(V)$ (i.e., 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 41 and 50 and additionally by line 29 for a *read*. If a faulty server conceals operation o with associated version vector V from a later operation o' whose associated version vector 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 vector 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; it 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 the C_r executing *read* cannot generally know if the server returns the data which corresponds to the latest preceding committed operation by C_l, C_r can make sure that it itself is not aware of any 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 33 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 31. In this case, V_x should be the operation of C_l which immediately preceded the concurrent *read*. This is assured by line 32 which makes sure that $V_r[l] = V_x[l] + 1$.

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 greatest version vector out of all committed operations thus far. The server code consists of two procedures (lines 5–13 and lines 15–17), 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 6, 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 greatest 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 5–13) 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

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

notation
  Strings $= \{0,1\}^* \cup \{\perp\}$; *Clients* $= \{1,\ldots,n\} \cup \{\perp\}$; *Opcodes* $= \{$*READ*, *WRITE*, $\perp\}$
  Operations $=$ *Clients* $\times$ *Opcodes* $\times \mathbb{N} \times$ *Clients* $\times$ *Strings*
  OpHashSets $= 2^{\{(o,h) \mid o \in \mathit{Operations}, h \in \mathit{Strings}\}}$ // lists of tuples from *Operations* and hash values
state
  $V_{old}[i] \in \mathbb{N}$, initially $V_{old}[i] = 0$, for $i \in \{1,\ldots,n\}$ // version vector of last operation
  $\mathcal{M}_{old} \in$ *OpHashSets*, initially empty // list of missing proofs
  $\bar{w} \in$ *Strings*, initially $\perp$ // most recently written value
write(w)
  $\bar{w} \leftarrow w$
  $\tau' \leftarrow \text{sign}_i(\text{SUBMIT} \| \textit{WRITE} \| V_{old}[i] \| i)$
  $op \leftarrow (i, \textit{WRITE}, V_{old}[i], i, \tau')$
  send $\langle \text{SUBMIT}, op \rangle$ to S
  wait for a message $\langle \text{REPLY}, S_{\mathit{info}}, \mathcal{C}, \mathcal{P} \rangle$ from S
    where $S_{\mathit{info}} = (s, h_{o_S}, h_{x_S}, V_S, \mathcal{M}_S, \varphi_S)$
  $(V_{new}, \mathcal{M}_{new}) \leftarrow \textit{common}(op, S_{\mathit{info}}, \mathcal{C}, \mathcal{P})$
  $(V_{old}, \mathcal{M}_{old}) \leftarrow (V_{new}, \mathcal{M}_{new})$
  $\varphi' \leftarrow \text{sign}_i(\text{COMMIT} \| H(op) \| H(w) \| H(V_{new}) \| H(\mathcal{M}_{new}))$
  send $\langle \text{COMMIT}, op, w, V_{new}, \mathcal{M}_{new}, \varphi' \rangle$ to S
  return *OK*
read(l)
  $w \leftarrow \bar{w}$
  $\tau' \leftarrow \text{sign}_i(\text{SUBMIT} \| \textit{READ} \| V_{old}[i] \| l)$
  $op \leftarrow (i, \textit{READ}, V_{old}[i], l, \tau')$
  send $\langle \text{SUBMIT}, op \rangle$ to S
  wait for a message $\langle \text{REPLY}, S_{\mathit{info}}, X_{\mathit{info}}, \mathcal{C}, \mathcal{P} \rangle$ from S
    where $S_{\mathit{info}} = (s, h_{o_S}, h_{x_S}, V_S, \mathcal{M}_S, \varphi_S)$ and $X_{\mathit{info}} = (h_{o_x}, x, V_x, \mathcal{M}_x, \varphi_x)$
  if not $\big(V_x = (0,\ldots,0)$ **or** $\textit{verify}_l(\varphi_x, \text{COMMIT} \| h_{o_x} \| H(x) \| H(V_x) \| H(\mathcal{M}_x))\big)$ **then halt**
  if $(V_S, \mathcal{M}_S) \not\geq (V_x, \mathcal{M}_x)$ **then halt**
  $(V_{new}, \mathcal{M}_{new}) \leftarrow \textit{common}(op, S_{\mathit{info}}, \mathcal{C}, \mathcal{P})$
  if exists $(o, E) \in \mathcal{M}_{new}$ where o is a *WRITE* by client C_l **then halt**
  if exists $(o, E) \in \mathcal{M}_{new}$ where o is a *READ* by client C_l
    then if $V_{new}[l] \neq V_x[l] + 1$ **then halt**
  else if $V_{new}[l] \neq V_x[l]$ **then halt**
  $(V_{old}, \mathcal{M}_{old}) \leftarrow (V_{new}, \mathcal{M}_{new})$
  $\varphi' \leftarrow \text{sign}_i(\text{COMMIT} \| H(op) \| H(w) \| H(V_{new}) \| H(\mathcal{M}_{new}))$
  send $\langle \text{COMMIT}, op, w, V_{new}, \mathcal{M}_{new}, \varphi' \rangle$ to S
  return x

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

common$(op, (s, h_{o_S}, h_{x_S}, V_S, \mathcal{M}_S, \varphi_S), \mathcal{C}, \mathcal{P})$
  if not $\big(V_S = (0, \ldots, 0)$ **or** *verify*$_s(\varphi_S, \text{COMMIT} \| h_{o_S} \| h_{x_S} \| H(V_S) \| H(\mathcal{M}_S))\big)$
    then halt
  if not $\big((V_S, \mathcal{M}_S) \geq (V_{old}, \mathcal{M}_{old})$ **and** $V_S[i] = V_{old}[i]\big)$ **then halt**
  if *op* is not the last operation in $\mathcal{C}$ **then halt**
  $\mathcal{M}_{new} \leftarrow \mathcal{M}_S$
  verify-proofs$(\mathcal{P}, \mathcal{M}_{new})$
  $V_{new} \leftarrow V_S$
  for $o = (c, oc, v, l, \tau)$ in $\mathcal{C}$ **do**
    if not $(v = V_{new}[c]$ and *verify*$_c(\tau, \text{SUBMIT} \| oc \| v \| l))$ **then halt**
    $V_{new}[c] \leftarrow V_{new}[c] + 1$
    if $o \neq op$ **then** add $(o, H(V_{new}))$ to $\mathcal{M}_{new}$
  if $(V_{new}, \mathcal{M}_{new}) \not> (V_S, \mathcal{M}_S)$ **then halt**
  if exists an operation by C_i in $\mathcal{M}_{new}$ **then halt**
  if exists a client C_j with more than one entry in $\mathcal{M}_{new}$ **then halt**
  return $(V_{new}, \mathcal{M}_{new})$
verify-proofs$(\mathcal{P}, \mathcal{M})$
  for $(c, o_c, h_{x_c}, h_{V_c}, h_{\mathcal{M}_c}, \varphi_c) \in \mathcal{P}$ **do**
    if not *verify*$_c(\varphi_c, \text{COMMIT} \| H(o_c) \| h_{x_c} \| h_{V_S} \| h_{\mathcal{M}_c}))$ **then halt**
    if exists E s.t. $(o_c, E) \in \mathcal{M}$
      if $E \neq h_{V_c}$ **then halt**
      remove (o_c, E) from $\mathcal{M}$

Algorithm 5 Concurrent protocol. Code for server.

state
  $\mathcal{C} \in$ *Operations**, initially empty // list of concurrent operations
  $X[i] = (op_i, x_i, V_i, \mathcal{M}_i, \varphi_i) \in$ *Operations* $\times$ *Strings* $\times \mathbb{N}^n \times$ *OpHashSets* $\times$ *Strings*,
    initially $(\perp, \perp, (0, \ldots, 0), \emptyset, \perp)$, for $i \in \{1, \ldots, n\}$ // current state
  $c \in$ *Clients*, initially 1.
upon receiving a message $\langle \text{SUBMIT}, op \rangle$ from C_i, where $op = (i, oc, v, l, \tau)$:
  if ((*oc* = *READ*) **and** (the latest operation op' of C_l s.t. $op' \in \mathcal{C}$
    or exists $(op', E) \in \mathcal{M}_c$, is a *WRITE*)) **then wait until** $op_l = op'$
  append op to the end of $\mathcal{C}$
  $\mathcal{P} \leftarrow \emptyset$
  for each o by client C_j s.t. $(o, *) \in \mathcal{M}_c$ **do**
    if $(op_j = o)$ **then** add $(j, op_j, H(x_j), H(V_j), H(\mathcal{M}_j), \varphi_j)$ to $\mathcal{P}$
  if $oc =$ *READ* **then**
    send $\langle \text{REPLY}, (c, H(op_c), H(x_c), V_c, \mathcal{M}_c, \varphi_c), (H(op_l), x_l, V_l, \mathcal{M}_l, \varphi_l), \mathcal{C}, \mathcal{P} \rangle$ to C_i
  else
    send $\langle \text{REPLY}, (c, H(op_c), H(x_c), V_c, \mathcal{M}_c, \varphi_c), \mathcal{C}, \mathcal{P} \rangle$ to C_i
upon receiving a message $\langle \text{COMMIT}, op, x, V, \mathcal{M}, \varphi \rangle$ from client C_i:
  $X[i] \leftarrow (op, x, V, \mathcal{M}, \varphi)$
  if $(V > V_c)$ **then**
    $c \leftarrow i$
    remove from $\mathcal{C}$ the prefix of operations up to (including) op

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 follows the protocol, 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 greatest 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 15–17), 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 communication complexity of the concurrent SUNDR protocol [MS02] by an order of magnitude; the protocol achieves the same guarantees, in the same model, but has communication complexity $O(n^2 + n(\kappa + |x|))$, which remains its worst-case complexity even after applying the suggested bandwidth optimizations.

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 avoid this for the sake of simplicity of presentation.

Analysis. The rest of this section is devoted to the analysis of the concurrent protocol. It is structured in the same way as the analysis of Mazières and Shasha [MS02].

Lemma 9. *Let o and o' be two succeeding operations completed by the same client C_i with associated version vectors v and v', respectively, such that o precedes o'. Then (a) $v'[i] = v[i] + 1$; and (b) $v < v'$*

Proof. Consider the processing of o' by C_i. By line 41 of procedure *common*, $V_S[i] = V_{old}[i] = v[i]$. On line 45 we have $V_{new} \leftarrow V_S$ and then when o' is encountered in $\mathcal{C}$ (o' must be the last operation in

$\mathcal{C}$ by line 42), the i-th entry is incremented to $V_S[i]+1$. Notice that there are no later operations by C_i in $\mathcal{C}$ (since no such operations exist when o' is processed) and that if o' is encountered more than once in $\mathcal{C}$, the check on line 47 will fail. Thus, at the end of procedure *common*, $v'[i] = V_{new}[i] = V_S[i]+1 = V_{old}[i]+1 = v[i]+1$, which proves (a). The check on line 41 requires that for every index j, $V_S[j] \geq V_{old}[j] = v[j]$. From (a) we have $v'[i] > v[i]$. Thus $v' > v$, as required by (b). □

Lemma 10. *Let $o = (c, oc, v, l, \tau)$ and $o' = (c', oc', v', l', \tau')$ be two operations that commit $(V, \mathcal{M})$ and $(V', \mathcal{M}')$ respectively, s.t. $(V, \mathcal{M}) < (V', \mathcal{M}')$. Then $V[c'] < V'[c']$.*

Proof. Note that since there is a strong inequality, $o' \neq o$. Assume for the purpose of reaching a contradiction that $V[c'] = V'[c']$ (since $V' > V$, we know that $V[c'] \leq V'[c']$). Since $V'[c] \geq V[c]$ then either o' saw o in $\mathcal{C}$, otherwise $V_S[c] \geq V_c[c]$ and then either $o_S = o$ or o_S saw o in its $\mathcal{C}$, or $V'_S[c] \geq V[c]$ where V'_S is the V_S received by o_S from the server, and so on.

Notice that if o' inserts some $(o'', H(V''))$ to $\mathcal{M}$, then $V'[c'] > V''[c']$, since o', which is the last element in $\mathcal{C}$, according to line 42, causes the c'-th entry to be incremented, after $(o'', H(V''))$ was already added to $\mathcal{M}$. Therefore, if o' indeed sees o in $\mathcal{C}$ and insets $(o, E = H(V''))$ to $\mathcal{M}$, then we get $V''[c'] < V'[c']$. Therefore, V'' cannot be V (we assumed $V'[c'] = V[c']$), and $E \neq H(V)$, which contradicts $(V, \mathcal{M}) < (V', \mathcal{M}')$: we assumed that $V[c'] = V'[c']$, and o cannot have an operation by c in its $\mathcal{M}$ by line 51, which leaves only the option of $E = H(V)$, which as we just saw does not hold.

Therefore, it must be that either $o_S = o$ or o_S saw o in its $\mathcal{C}$, or $V'_S[c] \geq V[c]$ where V'_S is the V_S received by o_S from the server, and so on. However, from Lemma 9 we know that $V_S[c'] < V'[c']$, and therefore, $o_S \neq o$ since we assumed $V[c'] = V'[c']$. Furthermore, if V_S inserts (o, E) to $\mathcal{M}_S$, this will also be with a version vector which has less than $V'[c']$ is the c'-th entry, i.e., $E \neq H(V)$ just like before. Similarly we show that some operation, either o_S, or the operation o'_S that o_S received from the server, or the one o'_S received from the server, and so on, had to insert o with the (hash of) wrong expected version vector to $\mathcal{M}$. Moreover, no such operation could have removed this entry from $\mathcal{M}$, since the verification procedure checks that $E = H(V)$, which is not the case. Therefore, o' must have received (o, E) in $\mathcal{M}_S$, inserted it to $\mathcal{M}_{new}$ (and could not remove it from this list, as explained before). But this contradicts $(V, \mathcal{M}) < (V', \mathcal{M}')$ as was explained above. □

Lemma 11. *The order relation on version-vector/list-of-missing-proofs pairs is transitive, i.e., if $(V_1, \mathcal{M}_1) \leq (V_2, \mathcal{M}_2)$ and $(V_2, \mathcal{M}_2) \leq (V_3, \mathcal{M}_3)$ then $(V_1, \mathcal{M}_1) \leq (V_3, \mathcal{M}_3)$*

Proof. If $(V_1, \mathcal{M}_1) = (V_2, \mathcal{M}_2)$ or $(V_2, \mathcal{M}_2) = (V_3, \mathcal{M}_3)$, the proof is trivial. We therefore assume that $(V_1, \mathcal{M}_1) < (V_2, \mathcal{M}_2)$ and $(V_2, \mathcal{M}_2) < (V_3, \mathcal{M}_3)$. First, since $V_1 < V_2 < V_3$, by transitivity of the '$<$' relation on version vectors we have $V_1 < V_3$. Second, for any $(o, E) \in \mathcal{M}_3$ where $o = (c, oc, v, l, \tau)$, (a) if $V_2[c] < V_3[c]$, then since $V_1 < V_2$, we have $V_1[c] < V_3[c]$; otherwise, if $V_2[c] = V_3[c]$ then (b) if $(o, E) \in (V_2, \mathcal{M}_2)$, then either (b-1) $V_1[c] < V_2[c]$ and then we have $V_1[c] < V_3[c]$; otherwise, if $V_1[c] = V_2[c]$ then (b-2) $(o, E) \in (V_1, \mathcal{M}_1)$, or the last option is (b-3) $E = H(V_1)$, which also stands to the requirement. (c) $E = H(V_2)$. This means that $(V_2, \mathcal{M}_2)$ was committed by o (an operation by C_c). Since $(V_1, \mathcal{M}_1) < (V_2, \mathcal{M}_2)$, by Lemma 10 we have $V_1[c] < V_2[c] = V_3[c]$. □

Lemma 12. *(a) All operations of the same client are ordered according to the order on (version vector, list of missing proofs)*
(b) Let o_r be a completed read operation that commits $(V_r, \mathcal{M}_r)$ and receives X_{info} = ($H(o_x)$, x, V_x, $\mathcal{M}_x$, φ_x) from the server, and o_x the operation that wrote this data, then $(V_r, \mathcal{M}_r) \geq (V_x, \mathcal{M}_x)$.
(c) Let o_r be a completed read operation that commits $(V_r, \mathcal{M}_r)$ and received X_{info} = ($H(o_x)$, x, V_x, $\mathcal{M}_x$, φ_x) from the server, and o_w be o_x if o_x is a write *operation or the latest* write *operation that precedes o_x if o_x is a* read *operation, s.t. o_w commits with $(V_w, \mathcal{M}_w)$. Then $(V_r, \mathcal{M}_r) \geq (V_w, \mathcal{M}_w)$.*

Proof. When o_r is executed, the check $(V_{new}, \mathcal{M}_{new}) > (V_S, \mathcal{M}_S)$ on line 50 in the *common* procedure must hold if o_r successfully completes. The check on line 41 in the *common* procedure makes sure that $(V_S, \mathcal{M}_S) \geq (V_{old}, \mathcal{M}_{old})$. The check on line 29 in the *read* procedure makes sure that $(V_S, \mathcal{M}_S) \geq (V_x, \mathcal{M}_x)$. (a) is correct by transitivity of the order on (version vector, list of missing proofs) pairs and the first+second checks mentioned above. (b) is correct by transitivity of the order on (version vector, list of missing proofs) pairs and the first+third checks mentioned above. (c) is correct by transitivity of the order on (version vector, list of missing proofs) pairs and (a)+(b). □

Lemma 13. *Let o be an operation that commits with version vector V_{new} and missing proof list $\mathcal{M}_{new}$. Then for each $(o', E') \in \mathcal{M}_{new}$ where $o' = (c', oc', v', l', \tau')$, it holds that $V_{new}[c'] = v' + 1$.*

Proof. Consider the following sequence of operations, ending with o. If $(o', E') \notin \mathcal{M}_S$, o is the only operation in the sequence. Otherwise, the operation that precedes o in this sequence is o_S, the operation which committed the information sent by the server in the S_{info} structure. If $(o', E') \notin \mathcal{M}'_S$, where $\mathcal{M}'_S$ is the $\mathcal{M}_S$ received by o_S from the server, then o_S is the first in the sequence. Otherwise, the operation that precedes o_S in the sequence is o'_S, which is the operation that committed the information sent by the server in the S'_{info} structure to o_S, and so on. The first operation in the sequence is the one that did not receive (o', E') from the server in $\mathcal{M}_S$, and still committed $\mathcal{M}_{new}$ which included (o', E').

We prove the lemma by induction on the position of o in this sequence. The base case is that o is the first in this sequence, i.e., did not have (o', E') in $\mathcal{M}_S$ it received from the server, and included (o', E') in $\mathcal{M}_{new}$. Thus, o saw o' in $\mathcal{C}$. According to line 47, $V_{new}[c] = v$ when o' was encountered in $\mathcal{C}$. Then $V_{new}[c]$ is incremented once on line 48 (it cannot be incremented more than once because of line 52), and thus $V_{new}[c] = v + 1$.

Suppose that the claim holds for every operation o in a position $i < k$ in the sequence, and let o be the $k > 1$ operation in the sequence. Therefore, o gets $(o', E') \in \mathcal{M}_S$. By induction assumption, $V_S[c'] = v' + 1$. o cannot encounter in $\mathcal{C}$ o' or any other operation by C'_c that precedes o', otherwise the check on line 47 would be violated. Since $(o', E') \in \mathcal{M}_{new}$ signed by o, o cannot encounter later operations by C'_c in $\mathcal{C}$, since the check on line 52 would be violated. Therefore, no operations by C'_c are seen in $\mathcal{C}$ by o and $V_{new}[c'] = V_S[c'] = v' + 1$. □

Lemma 14. *If an operation o by client C_i reads register i and successfully completes, then it cannot receive stale data from the server.*

Proof. The received data was written by C_i, otherwise the signature check on line 28 would fail. Since the latest data by C_i was written with a version vector V_{old}, and we assume that V_x corresponds to a stale operation, we have that $V_x[i] < V_{old}[i]$. By line 51, $\mathcal{M}_{new}$ does not include operations by C_i in $\mathcal{M}_{new}$. Thus, the check on line 33 of the *read* procedure requires that $V_S[i] = V_x[i]$. By the check on line 41 of the *common* procedure, $V_S[i] = V_{old}[i]$ and thus $V_x[i] = V_{old}[i]$, contradicting the assumption that the data is stale. □

The following lemma proves that if the server hides data that was written by operation o_w, then o_w and the read o_r that returns the stale data (i.e., data which was written before o_w) will have incomparable version vectors. Since Lemma 14 proved that stale data cannot be returned if the reader read his own data, we assume that the writer and the reader are different clients.

Lemma 15. *Let o_r be a completed* read *operation by client C_r of register $l \neq r$, which received X_{info} $= (H(o_x), x, V_x, \mathcal{M}_x, \varphi_x)$ from the server, and committed with version vector V_r. If exists an operation o_w by C_l that starts after o_x completes, completes before o_r starts, and commits with version vector V_w, then (a) $V_r[r] > V_w[r]$; and (b) $V_r[l] < V_w[l]$.*

Proof. (a) The r-th entry in a version vector is incremented (on line 48) only when encountering an announcement of an operation by C_r. Until the beginning of o_r no operation could possibly encounter announcements of operations by C_r with a sequence number of $V_r[r] - 1$ or greater (notice that announcements are signed in the *read* and *write* procedure, and the signatures are checked in the *common* procedure). Thus, no operation could set the r-th entry in its associated version vector to $V_r[r]$ or greater. Since o_r starts after o_w ends, we conclude that $V_r[r] > V_w[r]$.

(b) Observe the computation of V_r during the processing of o_r. First, by line 31 in *read* procedure, there could not be an unverified *write* operation by C_l. However, there might be an unverified *read* operation by C_l. Suppose that there indeed exists a *read* operation o_l by C_l that appears in $\mathcal{M}_{new}$ committed by o_r. By line 32 in the *read* procedure, $V_r[l] = V_x[l] + 1$. Since o_l is a *read* operation and o_x is the operation that immediately preceded it, o_w (a *write* operation that comes after o_x) must be a later operation by C_l then o_l, i.e., must have more than $V_x[l] + 1$ in the l-th entry (by Lemma 9), and we get $V_w[l] > V_x[l] + 1 = V_r[l]$. If there is no such *read* operation by C_l in $\mathcal{M}_{new}$, by line 33 in the *read* procedure, $V_r[l] = V_x[l]$. Since we assumed that o_w starts after o_x, we have that $V_w[l] > V_x[l]$ (by Lemma 9), and thus $V_w[l] > V_r[l]$ as required. □

Lemma 16. *After an operation o by C_i adopts V_S on line 45, each entry in the version vector is incremented at most once.*

Proof. From Lemma 9, the i-th entry is incremented exactly once during o. For any other $j \neq i$, an entry in the version vector will be incremented only if an operation by C_j is found in $\mathcal{C}$, and each such operation is inserted into the $\mathcal{M}_{new}$ list. Since the check on line 52 makes sure that not more than a single operation by C_j is in $\mathcal{M}_{new}$ (and the check on line 47 makes sure that no operation appears twice in $\mathcal{C}$), we conclude that the j-th entry could be incremented only once. □

Lemma 17. *If operation o and operation o' sign version vectors V and V' respectively, s.t. $V[i] > V'[i]$ and $V'[j] > V[j]$, then if some operation o'' signs a version vector V'' s.t. $V'' \geq V$ and $V'' \geq V'$ then $\mathcal{M}_{new}$ committed by o'' includes (o, E) s.t. $E \neq H(V)$ or (o', E) s.t. $E \neq H(V')$ (i.e., the expected version vector is incorrect).*

Proof. Observe the following sequence of operations ending with o''. Preceding o'' in this sequence is o_S, the operation that committed the information received by o'' in the S_{info} structure. Before o_S comes o'_S, the operation that committed S'_{info} received during o_S. We continue building the sequence in this way and the first operation in the sequence is an operation o_m s.t. the version vector received by o_m from the server in the S_{info} structure, VS_m, does not have this property, i.e., it does not hold that both $VS_m \geq V$ and $VS_m \geq V'$. We prove the lemma by induction on the position of o in this sequence.

The base case is that o'' is the first operation in the sequence, i.e., $o'' = o_m$. There are three options regarding VS_m: (a) $VS_m[j] \geq V'[j]$ and $VS_m[i] \not\geq V[i]$; (b) $VS_m[i] \geq V[i]$ and $VS_m[j] \not\geq V'[j]$; and (c) $VS_m[i] \not\geq V[i]$ and $VS_m[j] \not\geq V'[j]$. We deal with each case separately.

(a) $VS_m[j] \geq V'[j]$ and $VS_m[i] \not\geq V[i]$. Since $V_m[i] \geq V[i]$, the i-th entry is incremented during o_m. By Lemma 16 the i-th entry cannot be incremented from $VS_m[i]$ more than once during o_m. Thus, $V_m[i] = V[i]$ and $VS_m[i] = V[i] - 1$. According to the check on line 47, the announcement that causes this increment, must have $V[i] - 1$ as the operation sequence number, and thus must be the announcement of o. Therefore, o appears in $\mathcal{C}$ and as any such operation it is inserted by o_m to $\mathcal{M}_{new}$ on line 49. Since VS_m is adopted on line 45, and then the entries can only be incremented on line 48, the version vector that o is expected to sign includes a value in the l-th entry which is at least $VS_m[j]$, and since $VS_m[j] \geq V'[j] > V[j]$, it cannot be V. Thus, (o, E) is inserted to $\mathcal{M}_{new}$ and $E \neq H(V)$.

(b) $VS_m[i] \geq V[i]$ and $VS_m[j] \not\geq V'[j]$. Since $V_m[j] \geq V'[j]$, the l-th entry is incremented during o_m. By Lemma 16 the l-th entry cannot be incremented from $VS_m[j]$ more than once during o_m. Thus,

$V_m[j] = V'[j]$ and $VS_m[j] = V'[j] - 1$. According to the check on line 47, the announcement that causes this increment, must have $V'[j] - 1$ as the operation sequence number, and thus must be the announcement of o'. Therefore, o' appears in $\mathcal{C}$ and as any such operation it is inserted by o_m to $\mathcal{M}_{new}$ on line 49. Since VS_m is adopted on line 45, and then the entries can only be incremented on line 48, the version vector that o' is expected to sign includes a value in the i-th entry which is at least $VS_m[i]$, and since $VS_m[i] \geq V[i] > V'[i]$, it cannot be V'. Thus, (o', E) is inserted to $\mathcal{M}_{new}$ and $E \neq H(V')$.

(c) $VS_m[i] \not\geq V[i]$ and $VS_m[j] \not\geq V'[j]$. Since $V_m[j] \geq V'[j]$ and $V_m[i] \geq V[i]$ and by Lemma 16 o_m can increment each entry of the version vector at most once, it must be that $VS_m[i] = V[i] - 1$, and $VS_m[j] = V'[j] - 1$. Therefore, o_m must see both o and o' in $\mathcal{C}$. One of them appears after the other in $\mathcal{C}$. If o appears before o', then the version vector o' expected to sign will have $V[i]$ in the i-th entry, and the proof continues just as in case (b). If o appears after o', then o will be expected to sign a version vector with $V'[j]$ in the l-th entry, and the proof continues just as in case (a).

Assume that the lemma holds for operations with position up to $k-1$ in the sequence we constructed. Observe the k-th operation o'' in the sequence, which received V_S and $\mathcal{M}_S$ from the previous one in the sequence. Since the version vector upon commit, V_{new} is always greater then V_S, from line 50, the version vector committed by the $k-1$'th operation in the sequence is greater then V_m. According to the induction assumption, this means that this operation includes o' or o with the incorrect expected version vector. o receives $\mathcal{M}_S$ from the $k-1$-th operation in the sequence and then $\mathcal{M}_S$ is included in $\mathcal{M}_{new}$. Next, all received proofs are verified. Since o'' commits, it does not fail. Thus, the server does not send the correct proofs for o and o', and the incorrect entry in $\mathcal{M}_{new}$ must remain there, which completes the proof of the lemma. □

Corollary 18. *If operation o signs $(V, \mathcal{M})$ and operation o' signs $(V', \mathcal{M}')$, s.t. $V[i] > V'[i]$ and $V'[j] > V[j]$, then no operation o_m signs $(V_m, \mathcal{M}_m)$ s.t. $(V_m, \mathcal{M}_m) \geq (V', \mathcal{M}')$ and $(V_m, \mathcal{M}_m) \geq (V, \mathcal{M})$*

Proof. Assume for the purpose of contradiction that there exists such operation o_m. By definition of the order on (version vector, list of missing proofs) pairs, $V_m \geq V$ and $V_m \geq V'$. By Lemma 17, one of the following holds: (a) $(o, E) \in \mathcal{M}_m$ and $E \neq H(V)$; (b) $(o', E) \in \mathcal{M}_m$ and $E \neq H(V')$;

If (a) holds, by Lemma 13, $V_m[i] = V[i]$ (the lemma talks about the sequence number in the announcement of o, which is $V[i]-1$). By definition of the order on (version vector, list of missing proofs) pairs, since $(V_m, \mathcal{M}_m) \geq (V, \mathcal{M})$, either $(o, E) \in \mathcal{M}$ or $E = H(V)$. The later does not hold in (a), and the former would contradict the check on line 51 that does not allow a client to include its own operations in the $\mathcal{M}$.

Similarly, if (b) holds, by Lemma 13, $V_m[j] = V'[j]$. By definition of the order on (version vector, list of missing proofs) pairs, since $(V_m, \mathcal{M}_m) \geq (V', \mathcal{M}')$, either $(o', E) \in \mathcal{M}'$ or $E = H(V')$. The later does not hold in (b), and the former would contradict the check on line 51 that does not allow a client to include its own operations in the $\mathcal{M}$. □

Lemma 19. *If the server follows the protocol, then an operation o by client C_i commits with a version vector V that is equal to the vector committed by the the preceding operation that was appended to $\mathcal{C}$ by the server, in all entries but the i-th entry, where it is greater by 1. Or, if o is the first operation to be appended to $\mathcal{C}$ in the execution, then V has 1 in the i-th entry and 0 in all other entries.*

Proof. If the server executes Algorithm 5, then every operation is appended to $\mathcal{C}$. Observe the order of these insertions. We can think of an operation as having a sequence number according to the number of operations that were appended to $\mathcal{C}$ before it. We call this sequence number k and prove the lemma by induction on k. Base case: $k = 0$. In this case, it receives $\mathcal{C} = \{o\}$ and $V_S = (0, \ldots, 0)$. First, V_{new} becomes V_S, and when o is encountered in $\mathcal{C}$, the i-th entry is incremented. Since it is encountered in

$\mathcal{C}$ exactly once (the server is correct), then the vector is 1 in the i-th entry and 0 in all the rest, like in V_S, the lemma holds. Suppose that the lemma holds for all operations having a sequence number up to $k-1$ and and consider the k-th operation.

The k-th operation o receives $\mathcal{M}_S$, V_S and $\mathcal{C}$ from the server. This information was committed by operation o_S. If o is the only operation in $\mathcal{C}$, then since the server is correct, $\mathcal{C}$ is always pruned up to and including o_S. Thus, the fact that o is the only operation in $\mathcal{C}$ means that o_S immediately precedes o in the sequence, i.e., o_S is the $k-1$-th operation in that sequence. Like in the base case, V_S is adopted and then the i-th entry is incremented exactly once. If o is not the only operation in $\mathcal{C}$, then since the server is correct, the operation that precedes o in $\mathcal{C}$ is the $k-1$-th operation in our sequence, and we denote it o'. By induction assumption, all operations in $\mathcal{C}$ differ one from another by only one entry, where they are greater by 1 from the preceding operation. o_S precedes the first operation in $\mathcal{C}$ when the server is correct, as was explained above. Therefore, o' signs exactly the version vector which o expects it to sign (i.e., o' is inserted to $\mathcal{M}_{new}$ with the hash of the correct version vector). V, the version vector signed by o differs only in the i-th entry from this vector, since o is the last in $\mathcal{C}$ and o' immediately precedes it. Therefore, the lemma holds in this case as well. □

Theorem 20. *The concurrent protocol emulates n SWMR registers on a Byzantine server with fork-linearizability.*

Proof. Let σ be the sequence of events observed by the clients in the protocol. We first exclude from σ the invocations of operations that were not completed. We construct the sets σ_i (for $i = 1 \dots n$) as required by the definition of fork-linearizability. We include in σ_i the last operation of client C_i, o_i, which committed $(V_i, \mathcal{M}_i)$. Then, we include all operations o' that committed with $(V', \mathcal{M}')$ s.t. $(V_i, \mathcal{M}_i) \geq (V', \mathcal{M}')$. We now create the sequences π_i from σ_i by sorting σ_i according to the order relation on (version vector, missing proof list) pairs. Since every operation inserted into π_i has an associated (version vector, missing proof list) pair which is less or equal to $(V_i, \mathcal{M}_i)$, according to Lemma 18 all operations in π_i are totally ordered according to this relation.

Requirement 1 of fork-linearizability is preserved by Lemma 12, the last committed operation of C_i was inserted into π_i, and all previous operations are less than the last according to the order on (version vector, list of missing proofs) pairs (Lemma 12) and thus were also included.

For any two operations o by client C_i and o' by client C_j in π_i s.t. o' started after o has completed in σ, we show that o was ordered before o' in π_i. Let $(V, \mathcal{M})$ and $(V', \mathcal{M}')$ be the data committed by o and o' respectfully. If these are operations by the same client, i.e., $i = j$, then this holds from Lemma 12, which says that $(V, \mathcal{M}) \leq (V', \mathcal{M}')$ and thus o would be ordered before o' in π_i. Otherwise, $i \neq j$, and then since o' starts after o has already completed, it must be that $o'[j] > o[j]$ (o could not see the announcement for o'). Since $(V, \mathcal{M})$ and $(V', \mathcal{M}')$ are ordered, it must be that $(V', \mathcal{M}') \geq (V, \mathcal{M})$, and thus, o' is ordered in π_i after o. Thus, the order of π_i preserves real-time order (requirement 2 of fork-linearizability).

Suppose that o which committed $(V, \mathcal{M})$ was included in groups π_i and π_j. Let o_i be the last operation of C_i in π_i, which committed with $(V_i, \mathcal{M}_i)$. By construction of π_i, $(V_i, \mathcal{M}_i) \geq (V, \mathcal{M})$. All operations o' that committed $(V', \mathcal{M}')$ s.t. $(V', \mathcal{M}') \leq (V, \mathcal{M})$ were also included into π_i due to the transitivity of the order on (version vector, list of missing proofs) pairs. For the same reason, the same group of operations were included into π_j as well. Thus, requirement 4 of fork-linearizability holds.

For requirement 3 of fork-linearizability, we need to show that π_i sequential specification holds. For any π_i, let $o_r \in \pi_i$ be a *read* operation by client C_r from register l which commits with $(V_r, \mathcal{M}_r)$ and receives $X_{info} = (H(o_x), x, V_x, \mathcal{M}_x, \varphi_x)$ from the server, and o_w be the *write* operation by C_l which is o_x if o_x is a *write* operation, or otherwise (if o_x is a *read*) the latest *write* operation that precedes o_x, s.t. o_w commits with $(V_w, \mathcal{M}_w)$. By Lemma 12, it holds that $(V_r, \mathcal{M}_r) \geq (V_w, \mathcal{M}_w)$. Thus, $o_w \in \pi_i$ by

transitivity of the order of (version vector, list of missing proofs) pairs as before. Last, we must show that there is no *write* operation o'_w in π_i, s.t. o'_w starts after o_w ends, ends before o_r starts, and writes to l (i.e., an operation by C_l).

By definition of o_w, it equals o_x if o_x is a *write* operation, or it is the latest *write* operation that precedes o_x if o_x is a *read* operation. In both cases, o'_w, which is a *write* operation that starts after o_w ends, must start after o_x ends. By Lemma 15, $V_r[r] > V'_w[r]$; and (b) $V_r[l] < V'_w[l]$. By Corollary 18, no operation o_m signs $(V_m, \mathcal{M}_m)$ s.t. $(V_m, \mathcal{M}_m) \geq (V'_w, \mathcal{M}'_w)$ and $(V_m, \mathcal{M}_m) \geq (V_r, \mathcal{M}_r)$. Let o_i be the last operation of C_i in π_i, which committed with $(V_i, \mathcal{M}_i)$. By construction of π_i, if both o'_w and o_r are included into π_i, this would mean that $(V_i, \mathcal{M}_i) \geq (V_r, \mathcal{M}_r)$ and $(V_i, \mathcal{M}_i) \geq (V'_w, \mathcal{M}'_w)$, which is impossible according to Corollary 18.

It is left to show that when the server is correct, (a) every admissible execution is complete, i.e., every operation eventually ends, and (b) that the execution is linearizable. (a) follows from the fact that the communication links are reliable, the fact that *write* operations are never blocked by the server, and since the queue of operations waiting to be processed by one of the server procedures, is managed as a FIFO queue. Thus, even if a *read* operation is blocked waiting for a *write* to complete, the *write* will eventually complete, the *read* will be inserted into the FIFO queue, and eventually processed. Since this *read* waits for one specific *write*, the one that was scheduled before it in $\mathcal{C}$ by the server, next time it will get to execute the server code it will not be blocked, and since a message from the server will eventually arrive back to the client, the operation will end (the code of the client never blocks).

Additionally, it can be easily shown that no checks in the client's code are violated if the server follows the protocol. One check worth noticing is the check that makes sure that there at most one operation by each client in $\mathcal{M}_{new}$. It might happen that an operation receives (o, E) in $\mathcal{M}_S$ s.t. o is an operation by C_j, and there a later operation o' by C_j, which appears in $\mathcal{C}$ and thus inserted into $\mathcal{M}_{new}$. Notice, that in this case, o must have committed, since the client executes operations sequentially. When the server constructs set $\mathcal{P}$, it will see that the latest committed operation by C_j, denoted o_j at the server, equals to o, and will include a proof for it in its REPLY message. Thus, before o' is included in $\mathcal{M}_{new}$, o will be removed from there by the *verify-proofs* procedure. Notice that the expected version vectors are correct when the server is correct, as explained in Lemma 19.

To show (b), we will show that a *read* operation always returns the value written by the latest completed operation by the writer. Suppose for the purpose of reaching a contradiction that this is not the case, and the *read* operation o_r by client C_r of register l, returns information written by operation o_x of client C_l, and there is a write operation o_w that is a later operation by C_l that completed before o_r starts. According to Lemma 15, the version vector V_r committed by o_r is incomparable with the version vector V_w committed by o_w. This contradicts Lemma 19 from which follows that every two operations have comparable version vector when the server is correct. □

Acknowledgments

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

References

[ACKM06] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, *Byzantine disk Paxos: Optimal resilience with Byzantine shared memory*, Distributed Computing **18** (2006), no. 5, 387–408.

[AW04] H. Attiya and J. Welch, *Distributed computing: Fundamentals, simulations and advanced topics*, second ed., Wiley, 2004.

[BEG+94] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, *Checking the correctness of memories*, Algorithmica **12** (1994), 225–244.

[CDvD+03] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, *Incremental multiset hash functions and their application to memory integrity checking*, Advances in Cryptology: ASIACRYPT 2003 (C. S. Laih, ed.), Lecture Notes in Computer Science, vol. 2894, Springer, 2003.

[CSG+05] D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas, *Towards constant bandwidth overhead integrity checking of untrusted data*, Proc. 26th IEEE Symposium on Security & Privacy, 2005.

[FKM02] K. Fu, F. Kaashoek, and D. Mazières, *Fast and secure distributed read-only file system*, ACM Transactions on Computer Systems **20** (2002), no. 1, 1–24.

[Fu98] K. E. Fu, *Group sharing and random access in cryptographic storage file systems*, Master Thesis, MIT LCS, 1998.

[Her91] M. Herlihy, *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems **11** (1991), no. 1, 124–149.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir, *Obstruction-free synchronization: Double-ended queues as an example*, Proc. 23rd Intl. Conference on Distributed Computing Systems (ICDCS), 2003, pp. 522–529.

[HW90] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Transactions on Programming Languages and Systems **12** (1990), no. 3, 463–492.

[JCT98] P. Jayanti, T. D. Chandra, and S. Toueg, *Fault-tolerant wait-free shared objects*, Journal of the ACM **45** (1998), no. 3, 451–500.

[Lam86] L. Lamport, *On interprocess communication*, Distributed Computing **1** (1986), no. 2, 77–85, 86–101.

[LKMS04] J. Li, M. Krohn, D. Mazires, and D. Shasha, *Secure untrusted data repository (SUNDR)*, Proc. 6th Symp. Operating Systems Design and Implementation (OSDI), 2004, pp. 121–136.

[LT89] N. A. Lynch and M. R. Tuttle, *An introduction to input/output automata*, CWI Quaterly **2** (1989), no. 3, 219–246.

[Lyn96] N. A. Lynch, *Distributed algorithms*, Morgan Kaufmann, San Francisco, 1996.

[MAD02] J.-P. Martin, L. Alvisi, and M. Dahlin, *Minimal Byzantine storage*, Proc. 16th International Conference on Distributed Computing (DISC 2002) (D. Malkhi, ed.), Lecture Notes in Computer Science, vol. 2508, Springer, 2002, pp. 311–325.

[MS02] D. Mazières and D. Shasha, *Building secure file systems out of Byzantine storage*, Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC), 2002, pp. 108–117.

[OR06] A. Oprea and M. K. Reiter, *On consistency of encrypted files*, Proc. 20th International Conference on Distributed Computing (DISC 2006) (S. Dolev, ed.), Lecture Notes in Computer Science, vol. 4167, 2006, pp. 254–268.

[PSL80] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, Journal of the ACM **27** (1980), no. 2, 228–234.