Fail-Aware Untrusted Storage[§]

Christian Cachin[*] Idit Keidar[†] Alexander Shraer[‡]

January 31, 2011

In diesem Sinne kannst du's wagen.
Verbinde dich; du sollst, in diesen Tagen,
Mit Freuden meine Künste sehn,
Ich gebe dir was noch kein Mensch gesehn.[1]

— Mephistopheles in *Faust I*, by J. W. Goethe

Abstract

We consider a set of clients collaborating through an online service provider that is subject to attacks, and hence not fully trusted by the clients. We introduce the abstraction of a *fail-aware untrusted service*, with meaningful semantics even when the provider is faulty. In the common case, when the provider is correct, such a service guarantees consistency (linearizability) and liveness (wait-freedom) of all operations. In addition, the service always provides accurate and complete consistency and failure detection.

We illustrate our new abstraction by presenting a *Fail-Aware Untrusted STorage service (FAUST)*. Existing storage protocols in this model guarantee so-called *forking* semantics. We observe, however, that none of the previously suggested protocols suffice for implementing fail-aware untrusted storage with the desired liveness and consistency properties (at least wait-freedom and linearizability when the server is correct). We present a new storage protocol, which does not suffer from this limitation, and implements a new consistency notion, called *weak fork-linearizability*. We show how to extend this protocol to provide eventual consistency and failure awareness in FAUST.

1 Introduction

Nowadays it is common for users to keep data at remote online service providers. Such services allow clients that reside in different domains to collaborate with each other through acting on shared data. Examples include distributed filesystems, versioning repositories for source code, Web 2.0 collaboration tools like Wikis and discussion forums, and "cloud computing" services, whereby shared resources, software, and information are provided on demand. Clients access the provider over an asynchronous network in day-to-day operations, and occasionally communicate directly with each other. Because the provider is subject to attacks, or simply because the clients do not fully trust it, the clients are interested in a meaningful semantics of the service, even when the provider misbehaves.

[*]IBM Research - Zurich, CH-8803 Rüschlikon, Switzerland. cca@zurich.ibm.com

[†]Department of Electrical Engineering, Technion, Haifa 32000, Israel. idish@ee.technion.ac.il

[‡]Yahoo! Research, 4401 Great America Parkway, Santa Clara, CA 95054, USA. shralex@yahoo-inc.com

[§]A preliminary version of this paper appeared in the proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009).

[1]In this mood you can dare to go my ways. / Commit yourself; you shall in these next days / Behold my arts and with great pleasure too. / What no man yet has seen, I'll give to you.

The service allows clients to invoke operations and should guarantee both consistency and liveness of these operations whenever the provider is correct. More precisely, the service considered here should ensure *linearizability* [12], which provides the illusion of atomic operations. As a liveness condition, the service ought to be *wait-free*, meaning that every operation of a correct client eventually completes, independently of other clients. When the provider is faulty, it may deviate arbitrarily from the protocol, exhibiting so-called Byzantine faults. Hence, some malicious actions cannot be prevented. In particular, it is impossible to guarantee that every operation is live, as the server can simply ignore client requests. Linearizability cannot be ensured either, since the server may respond with an outdated return value to a client, omitting more recent update operations that affected its state.

In this paper, we tackle the challenge of providing meaningful service semantics in such a setting and define a class of *fail-aware untrusted services*. We also present *FAUST*, a *Fail-Aware Untrusted STorage service*, which demonstrates our new notion for *online storage*. We do this by reinterpreting in our model, with an untrusted provider, two established notions: eventual consistency and fail-awareness.

Eventual consistency [24] allows an operation to complete before it is consistent in the sense of linearizability, and later notifies the client when linearizability is established and the operation becomes *stable*. Upon completion, only a weaker notion holds, which should include at least causal consistency [13], a basic condition that has proven to be important in various applications [1, 25]. Whereas the client invokes operations *synchronously*, stability notifications occur *asynchronously*; the client can invoke more operations while waiting for a notification on a previous operation.

Fail-awareness [9] additionally introduces a notification to the clients in case the service cannot provide its specified semantics. This gives the clients a chance to take appropriate recovery actions. Fail-awareness has previously been used with respect to timing failures; here we extend this concept to alert clients of Byzantine server faults whenever the execution is not consistent.

Our new abstraction of a *fail-aware untrusted service*, introduced in Section 3, models a data storage functionality. It requires the service to be linearizable and wait-free when the provider is correct, and to be always causally consistent, even when the provider is faulty. Furthermore, the service provides *accurate* consistency information in the sense that every stable operation is guaranteed to be consistent at all clients and that when the provider is accused to be faulty, it has actually violated its specification. Furthermore, the stability and failure notifications are *complete* in the sense that every operation eventually either becomes stable or the service alerts the clients that the provider has failed. For expressing the stability of operations, the service assigns a timestamp to every operation.

The main building block we use to implement our fail-aware untrusted storage service is an untrusted storage protocol. Such protocols guarantee linearizability when the server is correct, and weaker, so-called *forking* consistency semantics when the server is faulty [20, 16, 4]. Forking semantics ensure that if certain clients' perception of the execution is not consistent, and the server causes their views to diverge by mounting a *forking attack*, they eventually cease to see each other's updates or expose the server as faulty. The first protocol of this kind, realizing *fork-linearizable* storage, was implemented by SUNDR [20, 16].

Although we are the first to define a fail-aware service, the existing untrusted storage protocols come close to supporting fail-awareness, and it has been implied that they can be extended to provide such a storage service [16, 17]. However, none of the existing forking consistency semantics allow for *wait-free* implementations; in previous protocols [16, 4] concurrent operations by different clients may block each other, even if the provider is correct. In fact, no fork-linearizable storage protocol can be wait-free in all executions where the server is correct [4].

A weaker notion called *fork-*-linearizability* has been proposed recently [17]. But as we show in Section 7, the notion (when adapted to our model with only one server) cannot provide wait-free client operations either. Fork-*-linearizability also permits a faulty server to violate causal consistency, as we show in Section 8. Thus, no existing semantics for untrusted storage protocols is suitable for realizing our notion of fail-aware storage.

In Section 4, we define a new consistency notion, called *weak fork-linearizability*, which circumvents the above impossibility and has all necessary features for building a fail-aware untrusted storage service. We present a weak fork-linearizable storage protocol in Section 5 and show that it never causes clients to block, even if some clients crash. The protocol is efficient, requiring a single round of message exchange between a client and the server for every operation, and a communication overhead of $O(n)$ bits per request, where n is the number of clients.

Starting from the weak fork-linearizable storage protocol, we introduce our fail-aware untrusted storage service (FAUST) in Section 6. FAUST adds mechanisms for consistency and failure detection, issues eventual stability notifications whenever the views of correct clients are consistent with each other, and detects all violations of consistency caused by a faulty server. The FAUST protocol lets the clients exchange messages infrequently.

In summary, the contributions of this paper are:

1. The new abstraction of a fail-aware untrusted service, which guarantees linearizability and wait-freedom when the server is correct, eventually provides either consistency or failure notifications, and ensures causal consistency (Sections 2–3);
2. The insight that no existing forking consistency notion can be used for realizing fail-aware untrusted storage, because they inherently rule out wait-free implementations (Sections 4, 7, and 8);
3. An efficient wait-free protocol giving a Byzantine emulation of untrusted storage, relying on the novel notion of weak fork-linearizability (Section 5); and
4. The implementation of FAUST, our fail-aware untrusted storage service, from a wait-free untrusted storage protocol (Section 6).

Although this paper focuses on fail-aware untrusted services that provide a data storage functionality, we believe that the notion can be generalized to a large variety of services.

Related work. In order to provide wait-freedom when linearizability cannot be ensured, numerous real-world systems guarantee notions of eventual consistency, for example, Coda [14], Bayou [24], Tempest [19], and Dynamo [8]. As in many of these systems, the clients in our model are not simultaneously present and may be disconnected temporarily. Thus, eventual consistency is a natural choice for the semantics of our online storage application. Eventual consistency can be expressed through many different semantics [22].

FAUST adds timestamps to operation responses for consistency notifications, similar to some of the systems just mentioned. The stability notion of a fail-aware untrusted service resembles the one of Bayou [24] and other weakly consistent replicated systems [22], where an operation becomes stable when its position in the order of operations has been permanently determined. Stability is also used in multicast communication protocols [2, 5], where a message becomes stable if it has reached all its destinations.

The notion of fail-awareness [9] is exploited by many systems in the timed asynchronous model, where nodes are subject to crash failures [7]. Note that unlike in previous work, detecting an inconsistency in our model constitutes evidence that the server has violated its specification, and that it should no longer be used.

The pioneering work of Mazières and Shasha [20] introduces untrusted storage protocols and the notion of fork-linearizability (under the name of *fork consistency*). SUNDR [16] and later work [4] implement storage systems respecting this notion. The weaker notion of *fork-sequential consistency* has been suggested by Oprea and Reiter [21]. Neither fork-linearizability nor fork-sequential consistency can guarantee wait-freedom for client operations in all executions where the server is correct [4, 3].

Fork-*-linearizability [17] has been introduced recently (under the name of *fork-* consistency*), with the goal of allowing wait-free implementations of a service constructed using replication, when more than a third of the replicas may be faulty. In our context, we consider only the special case of a non-replicated service.

The CATS system [26] adds *accountability* to a storage service. Similar to our fail-aware approach, CATS makes misbehavior of the storage server detectable by providing auditing operations. However, it relies on a much stronger assumption in its architecture, namely, a trusted external publication medium accessible to all clients, like an append-only bulletin board with immutable write operations. The server periodically publishes a digest of its state there and the clients rely on it for audits. When the server in FAUST additionally signs all its responses to clients using digital signatures, then we obtain the same level of strong accountability as CATS (i.e., that any misbehavior leaves around cryptographically strong non-repudiable evidence and that no false accusations are possible).

Exploring a similar direction, the *A2M-Storage service* [6] guarantees linearizability, even when the server is faulty. It relies on the strong assumption of a trusted module with an immutable log that prevents equivocation by a malicious server. A2M-Storage provides two protocols: in the pessimistic protocol, a client first reserves a sequence number for an operation and then submits the actual operation with that sequence number; in the optimistic protocol, the client submits an operation right away, assuming that it knows the latest sequence number, and then restarts when the predicted sequence number was outdated. Both protocols guarantee weaker notions of liveness than FAUST when the server is correct. In fact, if a client fails just after reserving a sequence number in the pessimistic protocol, it prevents all other clients from progressing. The optimistic protocol is lock-free in the sense that *some* client always makes progress, but progress is not guaranteed for all clients. On the other hand, FAUST guarantees wait-freedom when the server is correct, that is, *all* correct clients complete every operation, regardless of failures or concurrent operations by other clients.

The idea of monitoring applications to detect consistency violations due to Byzantine behavior was considered in previous work in peer-to-peer settings, for example in PeerReview [10]. Eventual consistency has recently been used in the context of Byzantine faults by Zeno [23]; Zeno uses replication to tolerate server faults and always requires some servers to be correct. Zeno relaxes linearizable semantics to eventual consistency for gaining liveness, as does FAUST, but provides a slightly different notion of eventual consistency to clients than FAUST. In particular, Zeno may temporarily violate linearizability even when all servers are correct, which means inconsistencies are reconciled at a later point in time, whereas in FAUST linearizability is only violated if the server is Byzantine, but the application might be notified of operation stability (consistency) after the operation completes.

2 System Model

We consider an asynchronous distributed system consisting of n clients $C_1, \ldots, C_n$ and a server S. Every client is connected to S through an asynchronous reliable channel that delivers messages in first-in/first-out (FIFO) order. In addition, there is a low-bandwidth communication channel among every pair of clients, which is also reliable and FIFO-ordered. We call this an *offline* communication method because it stands for a method that exchanges messages reliably even if the clients are not simultaneously connected. The system is illustrated in Figure 1. The clients and the server are collectively called *parties*. System components are modeled as deterministic I/O Automata [18]. An automaton has a state, which changes according to *transitions* that are triggered by *actions*. A *protocol* P specifies the behaviors of all parties. An execution of P is a sequence of alternating states and actions, such that state transitions occur according to the specification of system components. The occurrence of an action in an execution is called an *event*.

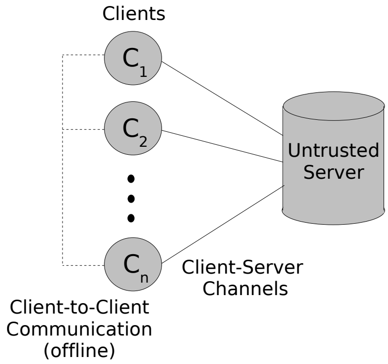

Figure 1: System architecture. Client-to-client communication may use offline message exchange.

All clients follow the protocol, and any number of clients can *fail* by crashing. The server might be faulty and deviate arbitrarily from the protocol. A party that does not fail in an execution is *correct*.

2.1 Preliminaries

Operations and histories. Our goal is to emulate a *shared functionality* F, i.e., a shared object, to the clients. Clients interact with F via *operations* provided by F. As operations take time, they are represented by two events occurring at the client, an *invocation* and a *response*. A *history* of an execution σ consists of the sequence of invocations and responses of F occurring in σ. An operation is *complete* in a history if it has a matching response. For a sequence of events σ, *complete*(σ) is the maximal subsequence of σ consisting only of complete operations.

An operation o *precedes* another operation o' in a sequence of events σ, denoted $o <_\sigma o'$, whenever o completes before o' is invoked in σ. A sequence of events π *preserves the real-time order* of a history σ if for every two operations o and o' in π, if $o <_\sigma o'$ then $o <_\pi o'$. Two operations are *concurrent* if neither one of them precedes the other. A sequence of events is *sequential* if it does not contain concurrent operations. For a sequence of events σ, the subsequence of σ consisting only of events occurring at client C_i is denoted by $\sigma|_{C_i}$ (we use the symbol $|$ as a projection operator). For some operation o, the prefix of σ that ends with the last event of o is denoted by $\sigma|^o$.

An operation o is said to be *contained in* a sequence of events σ, denoted $o \in \sigma$, whenever at least one event of o is in σ. Thus, every *sequential* sequence of events corresponds naturally to a sequence of operations. Analogously, every sequence of operations corresponds naturally to a sequential sequence of events.

An execution is *well-formed* if the sequence of events at each client consists of alternating invocations and matching responses, starting with an invocation. An execution is *fair*, informally, if it does not halt prematurely when there are still steps to be taken or messages to be delivered (see the standard literature for a formal definition [18]).

Read/write registers. A functionality F is defined via a *sequential specification*, which indicates the behavior of F in sequential executions.

The functionality considered in this paper is a storage service composed of *registers*. Each register X stores a value x from a domain $\mathcal{X}$ and offers *read* and *write* operations. Initially, a register holds a special value $\bot \notin \mathcal{X}$. When a client C_i invokes a read operation, the register responds with a value x, denoted *read*$_i(X) \to x$; when C_i invokes a write operation with value x, denoted *write*$_i(X, x)$, the response of

X is OK. By convention, an operation with subscript i is executed by C_i. The sequential specification requires that each read operation returns the value written by the most recent preceding write operation, if there is one, and the initial value otherwise. We assume that all values that are ever written to a register in the system are unique, i.e., no value is written more than once. This can easily be implemented by including the identity of the writer and a sequence number together with the stored value.

Specifically, the functionality F is composed of n single-writer/multi-reader (SWMR) registers $X_1, \ldots, X_n$, where every client may read from every register, but only client C_i can write to register X_i for $i = 1, \ldots, n$. The registers are accessed independently of each other. In other words, the operations provided by F to C_i are $\mathit{write}_i(X_i, x)$ and $\mathit{read}_i(X_j)$ for $j = 1, \ldots, n$.

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

A hash function maps a bit string of arbitrary length to a short, unique representation. The functionality provides only a single operation H; its invocation takes a bit string x as parameter and returns an integer h with the response. The implementation maintains a list L of all x that have been queried so far. When 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 $\mathit{verify}(i, s, m) \to \text{TRUE}$ for all $i \in \{1, \ldots, n\}$ and $m \in \{0,1\}^*$ if and only if C_i has executed $\mathit{sign}(i, m) \to s$ before, and $\mathit{verify}(i, s, m) \to \text{FALSE}$ otherwise. Only C_i may invoke $\mathit{sign}(i, \cdot)$ and S cannot invoke *sign*. Every party may invoke *verify*.

Traditional consistency and liveness properties. Our definitions rely on the notion of a possible *view* of a client, defined as follows.

Definition 1 (View). A sequence of events π is called a *view* of a history σ at a client C_i w.r.t. a functionality F if σ can be extended (by appending zero or more responses) to a history σ' such that:

1. π is a sequential permutation of some subsequence of $\mathit{complete}(\sigma')$;
2. $\pi|_{C_i} = \mathit{complete}(\sigma')|_{C_i}$; and
3. π satisfies the sequential specification of F.

Intuitively, a view π of σ at C_i contains at least all those operations that either occur at C_i or are apparent from C_i's interaction with F. Note there are usually multiple views possible at a client. If two clients C_i and C_j do not have a common view of a history σ w.r.t. a functionality F, we say that their views of σ are *inconsistent* with each other.

One of the most important consistency conditions for concurrent operations is linearizability, which guarantees that all operations occur atomically.

Definition 2 (Linearizability [12]). A history σ is *linearizable* w.r.t. a functionality F if there exists a sequence of events π such that:

1. π is a view of σ at all clients w.r.t. F; and
2. π preserves the real-time order of σ.

The notion of *causal consistency* for shared memory [13] weakens linearizability and allows clients to observe different orders of those write operations that do not influence each other. It is based on the notion of *potential causality* [15]. Recall that F consists of registers. For two operations o and o' in a history σ, we say that o *causally precedes* o', denoted $o \rightarrow_\sigma o'$, whenever one of the following conditions holds:

1. Operations o and o' are both invoked by the same client and $o <_\sigma o'$;
2. Operation o is a write operation of a value x to some register X and o' is a read operation from X returning x; or
3. There exists an operation $o'' \in \sigma$ such that $o \rightarrow_\sigma o''$ and $o'' \rightarrow_\sigma o'$.

In the literature, there are several variants of causal consistency. Here, we formalize the intuitive definition of causal consistency by Hutto and Ahamad [13].

Definition 3 (Causal consistency). A history σ is *causally consistent* w.r.t. a functionality F if for each client C_i there exists a sequence of events π_i such that:

1. π_i is a view of σ at C_i w.r.t. F;
2. For each operation $o \in \pi_i$, all write operations that causally precede o in σ are also in π_i; and
3. For all operations $o, o' \in \pi_i$ such that $o \rightarrow_\sigma o'$, it holds that $o <_{\pi_i} o'$.

Finally, a shared functionality needs to ensure liveness. A desirable requirement is that clients should be able to make progress independently of the actions or failures of other clients. A notion that formally captures this idea is *wait-freedom* [11].

Definition 4 (Wait-freedom). A history is *wait-free* if every operation by a correct client is complete.

By slight abuse of terminology, we say that an execution satisfies a notion such as linearizability, causal consistency, wait-freedom, etc., if its history satisfies the respective condition.

3 Fail-Aware Untrusted Services

Consider a shared functionality F that allows clients to invoke operations and returns a response for each invocation. Our goal is to implement F with the help of server S, which may be faulty.

We define a *fail-aware untrusted service* O^F from F as follows. When S is correct, then it should emulate F and ensure linearizability and wait-freedom. When S is faulty, then the service should always ensure causal consistency and eventually provide either consistency or failure notifications. For defining these properties, we extend F in two ways.

First, we include with the response of every operation of F an additional parameter t, called the *timestamp* of the operation. We say that an operation of O^F *returns a timestamp* t when the operation completes and its response contains timestamp t. The timestamps returned by the operations of a client increase monotonically. Timestamps are used as local operation identifiers, so that additional information can be provided to the application by the service regarding a particular operation, after that operation has already completed (using the *stable* notifications as defined below).

Second, we add two new output actions at client C_i, called *stable*$_i$ and *fail*$_i$, which occur asynchronously. (Note that the subscript i denotes an action at client C_i.) The action *stable*$_i$ includes a vector of timestamps W as a parameter and informs C_i about the stability of its operations with respect to the other clients.

Definition 5 (Operation stability). Let o be a complete operation of C_i that returns a timestamp t. We say that o is *stable w.r.t. a client* C_j, for $j = 1, \ldots, n$, after some event $stable_i(W)$ has occurred at C_i with $W[j] \geq t$. An operation o of C_i is *stable w.r.t. a set of clients* $\mathcal{C}$, where $\mathcal{C}$ includes C_i, when o is stable w.r.t. all $C_j \in \mathcal{C}$. Operations that are stable w.r.t. all clients are simply called *stable*.

Informally, $stable_i$ defines a *stability cut* among the operations of C_i with respect to the other clients, in the sense that if an operation o of client C_i is stable w.r.t. C_j, then C_i and C_j are guaranteed to have the same view of the execution up to o. If o is stable, then the prefix of the execution up to o is linearizable. The service should guarantee that every operation eventually becomes stable, but this may only be possible if S is correct. Otherwise, the service should notify the users about the failure.

Failure detection should be accurate in the sense that it should never output false suspicions. When the action $fail_i$ occurs, it indicates that the server is demonstrably faulty, has violated its specification, and has caused inconsistent views among the clients. According to the stability guarantees, the client application does not have to worry about stable operations, but might invoke a recovery procedure for other operations.

When considering an execution σ of O^F, we sometimes focus only on the actions corresponding to F, without the added timestamps, and without the *stable* and *fail* actions. We refer to this as the *restriction of* σ *to* F and denote it by $\sigma|_F$ (similar notation is also used for restricting a sequence of events to those occurring at a particular client).

Definition 6 (Fail-aware untrusted service). A shared functionality O^F is a *fail-aware untrusted service with functionality* F, if O^F implements the invocations and responses of F and extends it with timestamps in responses and with *stable* and *fail* output actions, and where the history σ of every fair execution such that $\sigma|_F$ is well-formed satisfies the following conditions:

1. *(Linearizability with correct server)* If S is correct, then $\sigma|_F$ is linearizable w.r.t. F;
2. *(Wait-freedom with correct server)* If S is correct, then $\sigma|_F$ is wait-free;
3. *(Causality)* $\sigma|_F$ is causally consistent w.r.t. F;
4. *(Integrity)* When an operation o of C_i returns a timestamp t, then t is bigger than any timestamp returned by an operation of C_i that precedes o;
5. *(Failure-detection accuracy)* If $fail_i$ occurs, then S is faulty;
6. *(Stability-detection accuracy)* If o is an operation of C_i that is stable w.r.t. some set of clients $\mathcal{C}$ then there exists a sequence of events π that includes o and a prefix τ of $\sigma|_F$ such that π is a view of τ at all clients in $\mathcal{C}$ w.r.t. F. If $\mathcal{C}$ includes all clients, then τ is linearizable w.r.t. F;
7. *(Detection completeness)* For every two correct clients C_i and C_j and for every timestamp t returned by an operation of C_i, eventually either *fail* occurs at all correct clients, or $stable_i(W)$ occurs at C_i with $W[j] \geq t$.

We now illustrate how a fail-aware service can be used by clients who collaborate from across the world by editing a file. Suppose that the server S is correct and three correct clients access it: Alice and Bob from Europe, and Carlos from America. Since S is correct, linearizability is preserved. However, the clients do not know this, and rely on *stable* notifications for detecting consistency. Suppose that it is daytime in Europe, Alice and Bob use the service, and they see the effects of each other's updates. However, they do not observe any operations of Carlos because he is asleep.

Suppose Alice completes an operation that returns timestamp 10, and subsequently receives a notification $stable_{Alice}([10, 8, 3])$, indicating that she is consistent with Bob up to her operation with timestamp 8, consistent with Carlos up to her operation with timestamp 3, and trivially consistent with herself up to her last operation (see Figure 2). At this point, it is unclear to Alice (and to Bob) whether Carlos is only temporarily disconnected and has a consistent state, or if the server is faulty and hides operations of Carlos from Alice (and from Bob). If Alice and Bob continue to execute operations while Carlos is offline, Alice will continue to see vectors with increasing timestamps in the entries corresponding to Alice and Bob. When Carlos goes back online, since the server is correct, all operations issued by Alice, Bob, and Carlos will eventually become stable at all clients.

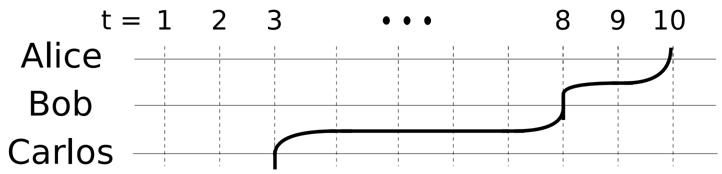

Figure 2: The stability cut of Alice indicated by the notification $stable_{Alice}([10, 8, 3])$. The values of t are the timestamps returned by the operations of Alice.

In order to implement a fail-aware untrusted service, we proceed in two steps. The first step consists of defining and implementing a weak fork-linearizable Byzantine emulation of a storage service. This notion is formulated in the next section and implemented in Section 5. The second step consists of extending the Byzantine emulation to a fail-aware storage protocol, as presented in Section 6.

4 Forking Consistency Conditions

This section introduces the notion of a *weak fork-linearizable Byzantine emulation* of a storage service. Section 4.1 first recalls existing forking semantics that are relevant here. Afterwards, in Section 4.2, the new notion of weak fork-linearizability is introduced. Section 4.3 defines Byzantine emulations with forking conditions.

4.1 Previously Defined Conditions

The notion of fork-linearizability [20] (originally called *fork consistency*) requires that when an operation is observed by multiple clients, the history of events occurring before the operation is the same. For instance, when a client reads a value written by another client, the reader is assured to be consistent with the writer up to the write operation.

Definition 7 (Fork-linearizability). A history σ is *fork-linearizable* w.r.t. a functionality F if for each client C_i there exists a sequence of events π_i such that:

1. π_i is a view of σ at C_i w.r.t. F;
2. π_i preserves the real-time order of σ;
3. (*No-join*) For every client C_j and every operation $o \in \pi_i \cap \pi_j$, it holds that $\pi_i|^o = \pi_j|^o$.

Li and Mazières [17] relax this notion and define *fork-*-linearizability* (under the name of *fork-* consistency*) by replacing the no-join condition of fork-linearizability with:

4. (*At-most-one-join*) For every client C_j and every two operations $o, o' \in \pi_i \cap \pi_j$ by the same client such that o precedes o', it holds that $\pi_i|^o = \pi_j|^o$.

The at-most-one-join condition of fork-*-linearizability guarantees to a client C_i that its view is identical to the view of any other client C_j up to the penultimate operation of C_j that is also in the view of C_i. Hence, if a client reads values written by *two* operations of another client, the reader is assured to be consistent with the writer up to the *first* of these writes.

But oddly, fork-*-linearizability still requires that the real-time order of *all* operations in the view is preserved, including the last operation of every other client. Furthermore, fork-*-linearizability does not preserve linearizability when the server is correct and permit wait-free client operations at the same time, as we show in Section 7.

4.2 Weak Fork-Linearizability

We introduce a new consistency notion, called *weak fork-linearizability*, which permits wait-free protocols and is therefore suitable for implementing fail-aware untrusted services. It is based on the notion of *weak* real-time order that removes the above anomaly and allows the last operation of every client to violate real-time order. Let π be a sequence of events and let *lastops*(π) be a function of π returning the set containing the last operation from every client in π (if it exists), that is,

$$\textit{lastops}(\pi) \triangleq \bigcup_{i=1,\ldots,n} \Big\{ o \in \pi|_{C_i} \;\Big|\; \text{there is no operation } o' \in \pi|_{C_i} \text{ such that } o \text{ precedes } o' \text{ in } \pi \Big\}.$$

We say that π *preserves the weak real-time order* of a sequence of operations σ whenever π excluding all events belonging to operations in *lastops*(π) preserves the real-time order of σ. With these notions, we are now ready to state weak fork-linearizability.

Definition 8 (Weak fork-linearizability). A history σ is *weakly fork-linearizable* w.r.t. a functionality F if for each client C_i there exists a sequence of events π_i such that:

1. π_i is a view of σ at C_i w.r.t. F;
2. π_i preserves the weak real-time order of σ;
3. For every operation $o \in \pi_i$ and every write operation $o' \in \sigma$ such that $o' \rightarrow_\sigma o$, it holds that $o' \in \pi_i$ and that $o' <_{\pi_i} o$; and
4. (*At-most-one-join*) For every client C_j and every two operations $o, o' \in \pi_i \cap \pi_j$ by the same client such that o precedes o', it holds that $\pi_i|^o = \pi_j|^o$.

Compared to fork-linearizability, weak fork-linearizability only preserves the *weak* real-time order in the second condition. The third condition in Definition 8 explicitly requires causal consistency; this is implied by fork-linearizability, as shown in Section 8. The fourth condition allows again an inconsistency for the last operation of every client in a view, through the at-most-one-join property from fork-*-linearizability. Hence, every fork-linearizable history is also weakly fork-linearizable.

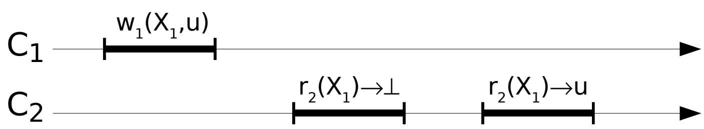

Figure 3: A weak fork-linearizable history that is not fork-linearizable.

Consider the following history, shown in Figure 3: Initially, X_1 contains $\bot$. Client C_1 executes *write*$_1(X_1, u)$, then client C_2 executes *read*$_2(X_1) \rightarrow \bot$ and *read*$_2(X_1) \rightarrow u$. During the execution of the first read operation of C_2, the server pretends that the write operation of C_1 did not occur. This history is weak fork-linearizable. The sequences:

$$\pi_1: \quad \textit{write}_1(X_1, u)$$
$$\pi_2: \quad \textit{read}_2(X_1) \rightarrow \bot,\ \textit{write}_1(X_1, u),\ \textit{read}_2(X_1) \rightarrow u$$

are a view of the history at C_1 and C_2, respectively. They preserve the weak real-time order of the history because the write operation in π_2 is exempt from the requirement. However, there is no way to construct a view of the execution at C_2 that preserves the real-time order of the history, as required by fork-linearizability. Intuitively, every protocol that guarantees fork-linearizability prevents this example because the server is supposed to reply to C_2 in a read operation with evidence for the completion of a concurrent or preceding write operation to the same register. But this implies that a reader should wait for a concurrent write operation to finish.

Weak fork-linearizability and fork-*-linearizability are not comparable in the sense that neither notion implies the other one. This is illustrated in Section 7 and follows, intuitively, because the real-time order condition of weak fork-linearizability is less restricting than the corresponding condition of fork-*-linearizability, but on the other hand, weak fork-linearizability requires causal consistency, whereas fork-*-linearizability does not.

4.3 Byzantine Emulation

We are now ready to define the requirements on our service. When the server is correct, it should guarantee the standard notion of linearizability. Otherwise, one of the three forking consistency conditions mentioned above must hold. In the following, let Γ be one of *fork*, *fork-**, or *weak fork*.

Definition 9 (Γ-linearizable Byzantine emulation). A protocol P *emulates* a functionality F *on a Byzantine server* S *with* Γ*-linearizability* whenever the following conditions hold:

1. If S is correct, the history of every fair and well-formed execution of P is linearizable w.r.t. F; and
2. The history of every fair and well-formed execution of P is Γ-linearizable w.r.t. F.

Furthermore, we say that such an emulation is *wait-free* when every fair and well-formed execution of the protocol with a correct server is wait-free.

A *storage service* in this paper is the functionality of an array of n SWMR registers, and a *storage protocol* provides a storage service. As mentioned before, we are especially interested in storage protocols that have only wait-free executions when the server is correct. In Section 7 we show that wait-free fork-*-linearizable Byzantine emulations of a storage service do not exist; this was already shown for fork-linearizability and fork sequential consistency [3].

5 A Weak Fork-Linearizable Untrusted Storage Protocol

We present a wait-free weak fork-linearizable emulation of n SWMR registers $X_1, \ldots, X_n$, where client C_i writes to register X_i.

At a high level, our untrusted storage protocol (USTOR) works as follows. When a client invokes a read or write operation, it sends a SUBMIT message to the server S. The server processes arriving SUBMIT messages in FIFO order; when the server receives multiple messages concurrently, it processes each message atomically. The client waits for a REPLY message from S. When this message arrives, C_i verifies its content and halts if it detects any inconsistency. Otherwise, C_i sends a COMMIT message to the server and returns without waiting for a response, returning OK for a write and the register value for a read. Sending a COMMIT message is simply an optimization to expedite garbage collection at S; this message can be eliminated by piggybacking its contents on the SUBMIT message of the next operation. The bulk of the protocol logic is devoted to dealing with a faulty server.

The USTOR protocol for clients is presented in Algorithm 1, and the USTOR protocol for the server appears in Algorithm 2. The notation uses *operations*, *upon-clauses*, and *procedures*. Operations correspond to the invocation events of the corresponding operations in the functionality, upon-clauses denote a condition and are actions that may be triggered whenever their condition is satisfied, and procedures are subroutines called from an operation or from an upon-condition. In the face of concurrency, operations and upon-conditions act like monitors: only one thread of control can execute any of them at a time. By invoking **wait for** *condition*, the thread releases control until *condition* is satisfied. The statement **return** *args* at the end of an operation means that it executes **output** *response*(*args*), which triggers the response event of the operation (denoted by *response* with parameters *args*).

We augment the protocol so that C_i may output an asynchronous event fail_i, in addition to the responses of the storage functionality. It signals that the client has detected an inconsistency caused by S; the signal will be picked up by a higher-layer protocol.

We describe the protocol logic in two steps: first in terms of its data structures and then by the flow of an operation.

Data structures. The variables representing the state of client C_i are denoted with the subscript i. Every client locally maintains a *timestamp* t that it increments during every operation (lines 113 and 126). Client C_i also stores a hash $\bar{x}_i$ of the value most recently written to X_i (line 107).

A SUBMIT message sent by C_i includes t and a DATA-signature δ by C_i on t and $\bar{x}_i$; for write operations, the message also contains the new register value x. The *timestamp of an operation* o is the value t contained in the SUBMIT message of o.

The operation is represented by an *invocation tuple* of the form (i, oc, j, σ), where *oc* is either READ or WRITE, j is the index of the register being read or written, and σ is a SUBMIT-signature by C_i on *oc*, j, and t. In summary, the SUBMIT message is

$$\langle \text{SUBMIT}, t, (i, oc, j, \sigma), x, \delta \rangle.$$

Client C_i holds a *timestamp vector* V_i, so that when C_i completes an operation o, entry $V_i[j]$ holds the timestamp of the last operation by C_j scheduled before o and $V_i[i] = t$. In order for C_i to maintain V_i, the server includes in the REPLY message of o information about the operations that precede o in the schedule. Although this prefix could be represented succinctly as a vector of timestamps, clients cannot rely on such a vector maintained by S. Instead, clients rely on digitally signed timestamp vectors sent by other clients. To this end, C_i signs V_i and includes V_i and the signature φ in the COMMIT message. The COMMIT message has the form

$$\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle,$$

where M_i and ψ are introduced later.

The server stores the register value, the timestamp, and the DATA-signature most recently received in a SUBMIT message from every client in an array *MEM* (line 202), and stores the timestamp vector and the signature of the last COMMIT message received from every client in an array *SVER* (line 204).

At the point when S sends the REPLY message of operation o, however, the COMMIT messages of some operations that precede o in the schedule may not yet have arrived at S. Hence, S includes explicit information in the REPLY message about the invocations of such submitted and not yet completed operations. Consider the schedule at the point when S receives the SUBMIT message of o, and let o^* be the most recent operation in the schedule for which S has received a COMMIT message. The schedule ends with a sequence $o^*, o^1, \ldots, o^\ell, o$ for $\ell \geq 0$. We call the operations $o^1, \ldots, o^\ell$ *concurrent* to o; the server stores the corresponding sequence of invocation tuples in L (line 205). Furthermore, S stores the index of the client that executed o^* in c (lines 203 and 219). The REPLY message from S to C_i contains c, L, and the timestamp vector V^c from the COMMIT message of o^* together with a signature φ^c by C_c. We use client index c as superscript to denote data in a message constructed by S, such that if S is correct, the data was sent by the indicated client C_c. Hence, the REPLY message for a write operation consists of

$$\langle \text{REPLY}, c, (V^c, M^c, \varphi^c), L, P \rangle,$$

where M^c and P are introduced later; the REPLY message for a read operation additionally contains the value to be returned.

We now define the *view history* $\mathcal{VH}(o)$ of an operation o to be a sequence of operations, as will be explained shortly. Client C_i executing o receives a REPLY message from S that contains a timestamp vector V^c, which is either 0^n or accompanied by a COMMIT-signature φ^c by C_c, corresponding to some operation o_c of C_c. The REPLY message also contains the list of invocation tuples L, representing a sequence of operations $\omega^1, \ldots, \omega^m$. Then we set

$$\mathcal{VH}(o) \triangleq \begin{cases} \omega^1, \ldots, \omega^m, o & \text{if } V^c = 0^n \\ \mathcal{VH}(o_c), \omega^1, \ldots, \omega^m, o & \text{otherwise,} \end{cases}$$

where the commas stand for appending operations to sequences of operations. Note that if S is correct, it holds that $o_c = o^*$ and $o^1, \ldots, o^\ell = \omega^1, \ldots, \omega^m$. View histories will be important in the protocol analysis.

After receiving the REPLY message (lines 117 and 129), C_i updates its vector of timestamps to reflect the position of o according to the view history. It does that by starting from V^c (line 138), incrementing one entry in the vector for every operation represented in L (line 143), and finally incrementing its own entry (line 147).

During this computation, the client also derives its own estimate of the view history of all concurrent operations represented in L. For representing these estimates compactly, we introduce the notion of a *digest* of a sequence of operations $\omega^1, \ldots, \omega^m$. In our context, it is sufficient to represent every operation ω^μ in the sequence by the index i^μ of the client that executes it. The *digest* $D(\omega^1, \ldots, \omega^m)$ of a sequence of operations is defined recursively using a hash function H as

$$D(\omega^1, \ldots, \omega^m) \triangleq \begin{cases} \bot & \textit{if } m = 0 \\ H\big(D(\omega^1, \ldots, \omega^{m-1}) \| i^m\big) & \textit{otherwise.} \end{cases}$$

The collision resistance of the hash function implies that the digest can serve a unique representation for a sequence of operations in the sense that no two distinct sequences that occur in an execution have the same digest.

Client C_i maintains a *vector of digests* M_i together with V_i, computed as follows during the execution of o. For every operation o_k by a client C_k corresponding to an invocation tuple in L, the client computes the digest d of $\mathcal{VH}(o)|^{o_k}$, i.e., the digest of C_i's expectation of C_k's view history of o_k, and stores d in $M_i[k]$ (lines 139, 146, and 148).

The pair (V_i, M_i) is called a *version*; client C_i includes its version in the COMMIT message, together with a so-called COMMIT-signature on the version. We say that *an operation o or a client C_i commits a version* (V_i, M_i) when C_i sends a COMMIT message containing (V_i, M_i) during the execution of o.

Definition 10 (Order on versions). We say that a version (V_i, M_i) is *smaller than or equal to* a version (V_j, M_j), denoted $(V_i, M_i) \dot{\leq} (V_j, M_j)$, whenever the following conditions hold:

1. $V_i \leq V_j$, i.e., for every $k = 1, \ldots, n$, it holds that $V_i[k] \leq V_j[k]$; and
2. For every k such that $V_i[k] = V_j[k]$, it holds that $M_i[k] = M_j[k]$.

Furthermore, we say that (V_i, M_i) is *smaller* than (V_j, M_j), and denote it by $(V_i, M_i) \dot{<} (V_j, M_j)$, whenever $(V_i, M_i) \dot{\leq} (V_j, M_j)$ and $(V_i, M_i) \neq (V_j, M_j)$. We say that two versions are *comparable* when one of them is smaller than or equal to the other.

Suppose that an operation o_i of client C_i commits (V_i, M_i) and an operation o_j of client C_j commits (V_j, M_j) and consider their order. The first condition orders the operations according to their timestamp vectors. The second condition checks the consistency of the view histories of C_i and C_j for operations that may not yet have committed. The precondition $V_i[k] = V_j[k]$ means that some operation o_k of C_k is the last operation of C_k in the view histories of o_i and of o_j. In this case, the prefixes of the two view histories up to o_k should be equal, i.e., $\mathcal{VH}(o_i)|^{o_k} = \mathcal{VH}(o_j)|^{o_k}$; since $M_i[k]$ and $M_j[k]$ represent these prefixes in the form of their digests, the condition $M_i[k] = M_j[k]$ verifies this. Clearly, if S is correct, then the version committed by an operation is bigger than the versions committed by all operations that were scheduled before. In the analysis, we show that this order is transitive, and that for all versions committed by the protocol, $(V_i, M_i) \dot{\leq} (V_j, M_j)$ if and only if $\mathcal{VH}(o_i)$ is a prefix of $\mathcal{VH}(o_j)$.

The COMMIT message from the client also includes a PROOF-signature ψ by C_i on $M_i[i]$ that will be used by other clients. The server stores the PROOF-signatures in an array P (line 206) and includes P in every REPLY message.

Algorithm flow. In order to support its extension to FAUST in Section 6, protocol USTOR not only implements read and write operations, but also provides *extended* read and write operations. They serve exactly the same function as standard counterparts, but additionally return the relevant version(s) from the operation.

Client C_i starts executing an operation by incrementing the timestamp and sending the SUBMIT message (lines 116 and 128). When S receives this message, it updates the timestamp and the DATA-signature in *MEM*$[i]$ with the received values for every operation, but updates the register value in *MEM*$[i]$ only for a write operation (lines 209–210 and 213). Subsequently, S retrieves c, the index of the client that committed the last operation in the schedule, and sends a REPLY message containing c and *SVER*$[c] = (V^c, M^c, \varphi^c)$. For a read operation from X_j, the reply also includes *MEM*$[j]$ and *SVER*$[j]$, representing the register value and the largest version committed by C_j, respectively. Finally, the server appends the invocation tuple to L (line 215).

After receiving the REPLY message, C_i invokes a procedure *updateVersion*. It first verifies the COMMIT-signature φ^c on the version (V^c, M^c) (line 136). Then it checks that (V^c, M^c) is at least as large as its own version (V_i, M_i), and that $V^c[i]$ has not changed compared to its own version (line 137). These conditions always hold when S is correct, since the channels are reliable with FIFO order and therefore, S receives and processes the COMMIT message of an operation before the SUBMIT message of the next operation by the same client.

Next, C_i starts to update its version (V_i, M_i) according to the concurrent operations represented in L. It starts from (V^c, M^c). For every invocation tuple in L, representing an operation by C_k, it checks the following (lines 140–146): first, that S received the COMMIT message of C_k's previous operation and included the corresponding PROOF-signature in $P[k]$ (line 142); second, that $k \neq i$, i.e., that C_i has no concurrent operation with itself (line 144); and third, after incrementing $V_i[k]$, that the SUBMIT-signature of the operation is valid and contains the expected timestamp $V_i[k]$ (line 144). Again, these conditions always hold when S is correct. During this computation, C_i also incrementally updates the digest d and assigns d to $M_i[k]$ for every operation. As the last step of *updateVersion*, C_i increments its own timestamp $V_i[i]$, computes the new digest, and assigns it to $M_i[i]$ (lines 147–148). If any of the checks fail, then *updateVersion* outputs fail_i and halts.

For read operations, C_i also invokes a procedure *checkData*. It first verifies the COMMIT-signature φ^j by the writer C_j on the version (V^j, M^j) (line 150). If S is correct, this is the largest version

Algorithm 1 Untrusted storage protocol (USTOR). Code for client C_i, part 1.

101: **notation**
102:     $\mathit{Strings} = \{0,1\}^* \cup \{\bot\}$
103:     $\mathit{Clients} = \{1, \dots, n\}$
104:     $\mathit{Opcodes} = \{\text{READ}, \text{WRITE}, \bot\}$
105:     $\mathit{Invocations} = \mathit{Clients} \times \mathit{Opcodes} \times \mathit{Clients} \times \mathit{Strings}$

106: **state**
107:     $\bar{x}_i \in \mathit{Strings}$, initially $\bot$ // hash of most recently written value
108:     $(V_i, M_i) \in \mathbb{N}_0^n \times \mathit{Strings}^n$, initially $(0^n, \bot^n)$ // last version committed by C_i

109: **operation** $\mathit{write}_i(x)$ // write x to register X_i
110:     $(\cdots) \leftarrow \mathit{writex}_i(x)$
111:     **return** OK

112: **operation** $\mathit{writex}_i(x)$ // extended write x to register X_i
113:     $t \leftarrow V_i[i] + 1$ // timestamp of the operation
114:     $\bar{x}_i \leftarrow H(x)$
115:     $\tau \leftarrow \text{sign}(i, \text{SUBMIT}\|\text{WRITE}\|i\|t)$; $\delta \leftarrow \text{sign}(i, \text{DATA}\|t\|\bar{x}_i)$
116:     send message $\langle \text{SUBMIT}, t, (i, \text{WRITE}, i, \tau), x, \delta \rangle$ to S
117:     **wait for** receiving a message $\langle \text{REPLY}, c, (V^c, M^c, \varphi^c), L, P \rangle$ from S
118:     $\mathit{updateVersion}(i, (c, V^c, M^c, \varphi^c), L, P)$
119:     $\varphi \leftarrow \text{sign}(i, \text{COMMIT}\|V_i\|M_i)$; $\psi \leftarrow \text{sign}(i, \text{PROOF}\|M_i[i])$
120:     send message $\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle$ to S
121:     **return** (V_i, M_i)

122: **operation** $\mathit{read}_i(X_j)$ // read from register X_j
123:     $(x^j, \cdots) \leftarrow \mathit{readx}_i(X_j)$
124:     **return** x^j

125: **operation** $\mathit{readx}_i(X_j)$ // extended read from register X_j
126:     $t \leftarrow V_i[i] + 1$ // timestamp of the operation
127:     $\tau \leftarrow \text{sign}(i, \text{SUBMIT}\|\text{READ}\|j\|t)$; $\delta \leftarrow \text{sign}(i, \text{DATA}\|t\|\bar{x}_i))$
128:     send message $\langle \text{SUBMIT}, t, (i, \text{READ}, j, \tau), \bot, \delta \rangle$ to S
129:     **wait for** a message $\langle \text{REPLY}, c, (V^c, M^c, \varphi^c), (V^j, M^j, \varphi^j), (t^j, x^j, \delta^j), L, P \rangle$ from S
130:     $\mathit{updateVersion}(j, (c, V^c, M^c, \varphi^c), L, P)$
131:     $\mathit{checkData}(c, (V^c, M^c, \varphi^c), j, (V^j, M^j, \varphi^j), (t^j, x^j, \delta^j))$
132:     $\varphi \leftarrow \text{sign}(i, \text{COMMIT}\|V_i\|M_i)$; $\psi \leftarrow \text{sign}(i, \text{PROOF}\|M_i[i])$
133:     send message $\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle$ to S
134:     **return** $(x^j, V_i, M_i, V^j, M^j)$

committed by C_j and received by S before it replied to C_i's read request. The client also checks the integrity of the returned value x^j by verifying the DATA-signature δ^j on t^j and on the hash of x^j (line 151). Furthermore, it checks that the version (V^j, M^j) is smaller than or equal to (V^c, M^c) (line 152). Although C_i cannot know if S returned data from the most recently submitted operation of C_j, it can check that C_j issued the DATA-signature during the most recent operation o_j of C_j represented in the version of C_i by checking that $t^j = V_i[j]$ (line 152). If S is correct and has already received the COMMIT message of o_j, then it must be $V^j[j] = t^j$, and if S has not received this message, it must be $V^j[j] = t^j - 1$ (line 153).

Finally, C_i sends a COMMIT message containing its version (V_i, M_i), a COMMIT-signature φ on the version, and a PROOF-signature ψ on $M_i[i]$ (lines 120 and 133).

When the server receives the COMMIT message from C_i containing a version (V_i, M_i), it stores the version and the PROOF-signature in *SVER*$[i]$ and stores the COMMIT-signature in $P[i]$ (lines 221 and

Algorithm 1 (cont.) Untrusted storage protocol (USTOR). Code for client C_i, part 2.

procedure *updateVersion*$(j, (c, V^c, M^c, \varphi^c), L, P)$
  if not $\big((V^c, M^c) = (0^n, \perp^n)$ **or** *verify*$(c, \varphi^c, \text{COMMIT}\|V^c\|M^c)\big)$ **then output** *fail*$_i$; **halt**
  if not $\big((V_i, M_i) \dot{\leq} (V^c, M^c)$ **and** $V^c[i] = V_i[i]\big)$ **then output** *fail*$_i$; **halt**
  $(V_i, M_i) \leftarrow (V^c, M^c)$
  $d \leftarrow M^c[c]$
  for $q = 1, \ldots, |L|$ **do**
    $(k, oc, l, \tau) \leftarrow L[q]$
    if not $\big(M_i[k] = \perp$ **or** *verify*$(k, P[k], \text{PROOF}\|M_i[k])\big)$ **then output** *fail*$_i$; **halt**
    $V_i[k] \leftarrow V_i[k] + 1$
    if $k = i$ **or not** *verify*$(k, \tau, \text{SUBMIT}\|oc\|l\|V_i[k])$ **then output** *fail*$_i$; **halt**
    $d \leftarrow H(d\|k)$
    $M_i[k] \leftarrow d$
  $V_i[i] = V_i[i] + 1$
  $M_i[i] \leftarrow H(d\|i)$

procedure *checkData*$(c, (V^c, M^c, \varphi^c), j, (V^j, M^j, \varphi^j), (t^j, x^j, \delta^j))$
  if not $\big((V^j, M^j) = (0^n, \perp^n)$ **or** *verify*$(j, \varphi^j, \text{COMMIT}\|V^j\|M^j)\big)$ **then output** *fail*$_i$; **halt**
  if not $\big(t^j = 0$ **or** *verify*$(j, \delta^j, \text{DATA}\|t^j\|H(x^j))\big)$ **then output** *fail*$_i$; **halt**
  if not $\big((V^j, M^j) \dot{\leq} (V^c, M^c)$ **and** $t^j = V_i[j]\big)$ **then output** *fail*$_i$; **halt**
  if not $\big(V^j[j] = t^j$ **or** $V^j[j] = t^j - 1\big)$ **then output** *fail*$_i$; **halt**

Algorithm 2 Untrusted storage protocol (USTOR). Code for server.

state
  MEM$[i] \in \mathbb{N}_0 \times \mathcal{X} \times$ *Strings*, initially $(0, \perp, \perp)$, for $i = 1, \ldots, n$ // last timestamp, value, and DATA-sig. received from C_i
  $c \in$ *Clients*, initially 1 // client who committed last operation in schedule
  SVER$[i] \in \mathbb{N}_0^n \times$ *Strings*$^n \times$ *Strings*, initially $(0^n, \perp^n, \perp)$, for $i = 1, \ldots, n$ // last version and COMMIT-signature received from C_i
  $L \in$ *Invocations**, initially empty // invocation tuples of concurrent operations
  $P \in$ *Strings*^{n}, initially $\perp^n$ // PROOF-signatures

upon receiving a message $\langle \text{SUBMIT}, t, (i, oc, j, \tau), x, \delta \rangle$ from C_i:
  if $oc = \text{READ}$ **then**
    $(t', x', \delta') \leftarrow$ *MEM*$[i]$
    MEM$[i] \leftarrow (t, x', \delta)$
    send message $\langle \text{REPLY}, c, \mathit{SVER}[c], \mathit{SVER}[j], \mathit{MEM}[j], L, P \rangle$ to C_i
  else
    MEM$[i] \leftarrow (t, x, \delta)$
    send message $\langle \text{REPLY}, c, \mathit{SVER}[c], L, P \rangle$ to C_i
  append (i, oc, j, τ) to L

upon receiving a message $\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle$ from C_i:
  $(V^c, M^c, \varphi^c) \leftarrow$ *SVER*$[c]$
  if $V_i > V^c$ **then**
    $c \leftarrow i$
    remove the last tuple of the form $(i, \cdots)$ and all preceding tuples from L
  SVER$[i] \leftarrow (V_i, M_i, \varphi)$
  $P[i] \leftarrow \psi$

222). Last but not least, the server checks if this operation is now the last committed operation in the schedule by testing $V_i > V^c$; if this is the case, the server stores i in c and removes from L the tuples representing this operation and all operations scheduled before. Note that L has at most n elements because at any time there is at most one operation per client that has not committed.

The following result summarizes the main properties of the protocol. As responding with a fail_i event is not foreseen by the specification of registers, we ignore those outputs in the theorem.

Theorem 1. *Protocol USTOR in Algorithms 1 and 2 emulates n SWMR registers on a Byzantine server with weak fork-linearizability; furthermore, the emulation is wait-free in all executions where the server is correct.*

Proof overview. A formal proof of the theorem appears in Appendix A. Here we explain intuitively why the protocol is wait-free, how the views of the weak fork-linearizable Byzantine emulation are constructed, and why the at-most-one-join property is preserved.

To see why the protocol is wait-free when the server is correct, recall that the server processes arriving SUBMIT messages atomically and in FIFO order. The order in which SUBMIT messages are received therefore defines the schedule of the corresponding operations, which is the linearization order when S is correct. Since communication channels are reliable and the event handler for SUBMIT messages sends a REPLY message to the client, the protocol is wait-free in executions where S is correct.

We now explain the construction of views as required by weak fork-linearizability. It is easy to see that whenever an inconsistency occurs, there are two operations o_i and o_j by clients C_i and C_j respectively, such that neither one of $\mathcal{VH}(o_i)$ and $\mathcal{VH}(o_j)$ is a prefix of the other. This means that if o_i and o_j commit versions (V_i, M_i) and (V_j, M_j), respectively, these versions are incomparable. By Lemma 16 in Appendix A, it is not possible then that any operation commits a version greater than both (V_i, M_i) and (V_j, M_j). Yet the protocol does not ensure that all operations appear in the view of a client ordered according to the versions that they commit. Specifically, a client may execute a read operation o_r and return a value that is written by a concurrent operation o_w; in this case, the reader compares its version only to the version committed by the operation of the writer that precedes o_w (line 152). Hence, o_w may commit a version incomparable to the one committed by o_r, although o_w must appear before o_r in the view of the reader.

In the analysis, we construct the view π_i of client C_i as follows. Let o_i be the last complete operation of C_i and suppose it commits version (V_i, M_i). We construct π_i in two steps. First, we consider all operations that commit a version smaller than or equal to (V_i, M_i), and order them by their versions. As explained above, these versions are totally ordered since they are smaller than (V_i, M_i). We denote this sequence of operations by ρ_i. Second, we extend ρ_i to π_i as follows: for every operation $o_r = \mathit{read}_j(X_k) \to v$ in ρ_i such that the corresponding write operation $o_w = \mathit{write}_k(X_k, v)$ is not in ρ_i, we add o_w immediately before the first read operation in ρ_i that returns v. We will show that if a write operation of client C_k is added at this stage, no subsequent operation of C_k appears in π_i. Thus, if two operations o and o' of C_k are both contained in two different views π_i and π_j and o precedes o', then $o \in \rho_i$ and $o \in \rho_j$. Because the order on versions is transitive and because the versions of the operations in ρ_i and ρ_j are totally ordered, we have that $\rho_i|^o = \rho_j|^o$. This sequence consists of all operations that commit a version smaller than the version committed by o. It is now easy to verify that also $\pi_i|^o = \pi_j|^o$ by construction of π_i and π_j. This establishes the at-most-one-join property.

Complexity. Each operation entails sending exactly three protocol messages (SUBMIT, REPLY, and COMMIT). Every message includes a constant number of components of the following types: timestamps, indices, register values, hash values, digital signatures, and versions. Additionally, the COMMIT message contains a list L of invocation tuples and a vector P of digital signatures. Although in theory, timestamps, hash values, and digital signatures may grow without bound, they grow very slowly. In practice, they are typically implemented by constant-size fields, e.g., 64 bits for a timestamp or 256 bits for a hash value. Let κ denote the maximal number of bits needed to represent a timestamp, hash value, or digital signature. For the sake of the analysis, we will assume that the number of steps taken by all parties of the protocol together is bounded by 2^κ. Register values in $\mathcal{X}$ require at most $\log|\mathcal{X}|$ bits. Indices are represented using $O(\kappa)$ bits. Versions consist of n timestamps and n hash values, and thus require $O(n\kappa)$ bits. For each client, at most one invocation tuple appears in L and at most one PROOF-signature in P. Hence, the sizes of L and P are also $O(n\kappa)$ bits. All in all, the bit complexity associated with an operation is $O(\log|\mathcal{X}| + n\kappa)$. Note that if S is faulty and sends longer messages, then some check by a client fails. Therefore, in all cases, each completed operation incurs at most $O(\log|\mathcal{X}|+n\kappa)$ communication complexity.

6 Fail-Aware Untrusted Storage Protocol

In this section, we extend the USTOR protocol of the previous section to a fail-aware untrusted storage protocol (FAUST). The new component at the client side calls the USTOR protocol and uses the offline client-to-client communication channels; its purpose is to detect the stability of operations and server failures. For both goals, FAUST needs access to the version of every operation, as maintained by the USTOR protocol; FAUST therefore calls the extended read and write operations of USTOR.

For *stability detection*, the protocol performs extra *dummy* operations periodically, for confirming the consistency of the preceding operations with respect to other clients. A client maintains the maximal version committed by the operations of every other client. When the client determines that a version received from another client is consistent with the version committed by an operation of its own, then it notifies the application that the operation has become stable w.r.t. the other client.

Our approach to *failure detection* takes up the intuition used for detecting forking attacks in previous fork-linearizable storage systems [20, 16, 4]. When a client ceases to obtain new versions from another client via the server, it contacts the other client directly with a PROBE message via offline communication and asks for the maximal version that it knows. The other client replies with this information in a VERSION message, and the first client verifies that all versions are consistent. If any check fails, the client reports the failure and notifies the other clients about this with a FAILURE message. The maximal version received from another client may also cause some operations to become stable; this combination of stability detection and failure detection is a novel feature of FAUST.

Figure 4 illustrates the architecture of the FAUST protocol. Below we describe at a high level how FAUST achieves its goals, and refer to Algorithm 3 for the details. For FAUST, we extend our pseudo-code by two elements. The notation **periodically** is an abbreviation for **upon** TRUE. The condition *completion of o with return value args* in an upon-clause stands for receiving the response of some operation o with parameters *args*.

Protocol overview. For every invocation of a read or write operation, the FAUST protocol at client C_i directly invokes the corresponding extended operation of the USTOR protocol. For every response received from the USTOR protocol that belongs to such an operation, FAUST adds the timestamp of the operation to the response and then outputs the modified response. FAUST retains the version committed by every operation of the USTOR protocol and takes the timestamp from the i-th entry in the timestamp vector (lines 316 and 325). More precisely, client C_i stores an array VER_i containing the maximal version that it has received from every other client. It sets $VER_i[i]$ to the version committed by the most recent operation of its own and updates the value of $VER_i[j]$ when a $readx_i(X_j)$ operation of the USTOR

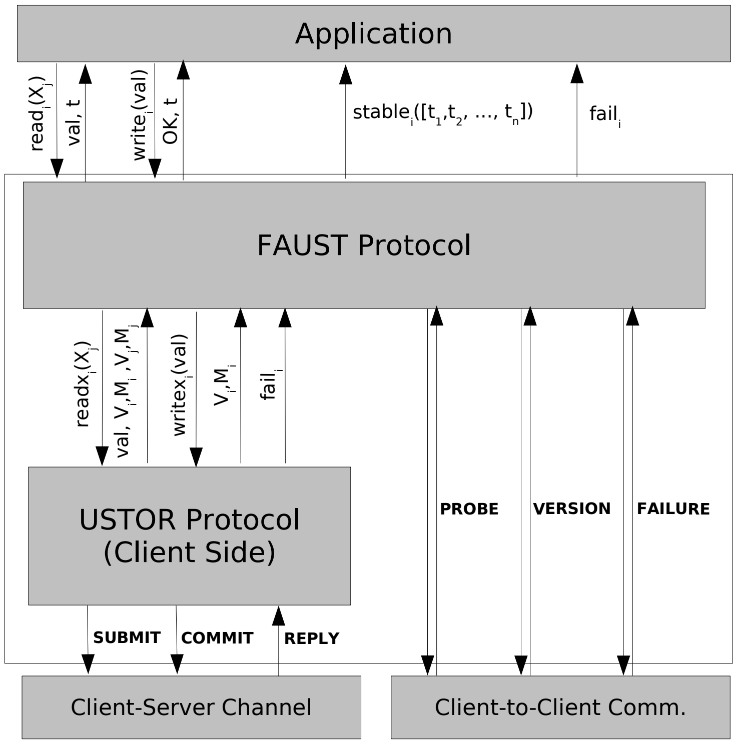

Figure 4: Architecture of the fail-aware untrusted storage protocol (FAUST).

protocol returns a version (V_j, M_j) committed by C_j. Let max_i denote the index of the maximum of all versions in VER_i.

To implement stability detection, C_i periodically issues a *dummy read* operation for the register of every client in a round-robin fashion (lines 331-332). In order to preserve a well-formed interaction with the USTOR protocol, FAUST ensures that it invokes at most one operation of USTOR at a time, either a read or a write operation from the application or a dummy read. We assume that the application invokes read and write operations in a well-formed manner and that these operations are queued such that they are executed only if no dummy read executes concurrently (this is omitted from the presentation for simplicity). The flags $execop_i$ and $execdummy_i$ indicate whether an application-triggered operation or a dummy operation is currently executing at USTOR, respectively. The protocol invokes a dummy read only if $execx_i$ and $dummyexec_i$ are FALSE.

However, dummy read operations alone do not guarantee stability-detection completeness according to Definition 6 because a faulty server, even when it only crashes, may not respond to the client messages in protocol USTOR. This prevents two clients that are consistent with each other from ever discovering that. To solve this problem, the clients communicate directly with each other and exchange their versions, as explained next.

For every entry $VER_i[j]$, the protocol stores in $T_i[j]$ the time when the entry was most recently updated. If a periodic check of these times reveals that more than δ time units have passed without an update from C_j, then C_i sends a PROBE message with no parameters directly to C_j (lines 329–330). Upon receiving a PROBE message, C_j replies with a message $\langle \text{VERSION}, (V, M)\rangle$, where $(V, M) = VER_j[max_j]$ is the maximal version that C_j knows. Client C_i also updates the value of $VER_i[j]$ when it receives a bigger version from C_j in a VERSION message. In this way, the stability detection mechanism eventually propagates the maximal version to all clients. Note that a VERSION message sent by C_i does

Algorithm 3 Fail-aware untrusted storage protocol (FAUST). Code for client C_i.

301: **state**
302: $k_i \in$ *Clients*, initially 0
303: $\mathit{VER}_i[j] \in \mathbb{N}_0^n \times \mathit{Strings}^n$, initially $(0^n, \bot^n)$, for $j = 1, \ldots, n$ // biggest received from C_j
304: $\mathit{max}_i \in$ *Clients*, initially 1 // index of client with maximal version
305: $W_i \in \mathbb{N}_0^n$, initially 0^n // maximal timestamps of C_i's operations observed by different clients
306: $\mathit{wchange}_i \in \{\text{FALSE}, \text{TRUE}\}$, initially TRUE // indicates that W_i changed since last $\mathit{stable}_i(W_i)$
307: $\mathit{execop}_i \in \{\text{FALSE}, \text{TRUE}\}$, initially FALSE // indicates that a non-dummy operation is executing
308: $\mathit{execdummy}_i \in \{\text{FALSE}, \text{TRUE}\}$, initially FALSE // indicates that a dummy operation is executing
309: $T_i \in \mathbb{N}^n$, initially 0^n // time when last updated version was received from C_j

310: **operation** $\mathit{write}_i(x)$:
311: $\mathit{execop}_i \leftarrow \text{TRUE}$
312: invoke $\mathit{USTOR.writex}_i(x)$

313: **upon** completion of $\mathit{USTOR.writex}_i$ with return value (V_i, M_i):
314: $\mathit{execop}_i \leftarrow \text{FALSE}$
315: $\mathit{update}(i, (V_i, M_i))$
316: **output** $(\text{OK}, V_i[i])$

317: **operation** $\mathit{read}_i(X_j)$:
318: $\mathit{execop}_i \leftarrow \text{TRUE}$
319: invoke $\mathit{USTOR.readx}_i(X_j)$

320: **upon** completion of $\mathit{USTOR.readx}_i$ with return value (x, V_i, M_i, V_j, M_j):
321: $\mathit{update}(i, (V_i, M_i))$
322: $\mathit{update}(j, (V_j, M_j))$
323: **if** execop_i **then**
324: $\mathit{execop}_i \leftarrow \text{FALSE}$
325: **output** $(x, V_i[i])$
326: **else**
327: $\mathit{execdummy}_i \leftarrow \text{FALSE}$

328: **periodically**:
329: $D \leftarrow \{C_j \mid \mathit{time}() - T_i[j] > \delta\}$
330: send message $\langle\text{PROBE}\rangle$ to all $C_j \in D$
331: **if not** execop_i **and not** $\mathit{execdummy}_i$ **then**
332: $k_i \leftarrow k_i \mod n + 1$
333: $\mathit{execdummy}_i \leftarrow \text{TRUE}$
334: invoke $\mathit{USTOR.readx}_i(k_i)$

335: **procedure** $\mathit{update}(j, (V, M))$:
336: **if not** $\big((V, M) \dot{\leq} \mathit{VER}_i[\mathit{max}_i]$ **or** $\mathit{VER}_i[\mathit{max}_i] \dot{\leq} (V, M)\big)$ **then**
337: $\mathit{fail}()$
338: **if** $\mathit{VER}_i[j] \dot{<} (V, M)$ **then**
339: $\mathit{VER}_i[j] \leftarrow (V, M)$
340: $T_i[j] \leftarrow \mathit{time}()$
341: **if** $\mathit{VER}_i[\mathit{max}_i] \dot{<} (V, M)$ **then**
342: $\mathit{max}_i \leftarrow j$
343: **if** $W_i[j] < V[i]$ **then**
344: $W_i[j] \leftarrow V[i]$
345: $\mathit{wchange}_i \leftarrow \text{TRUE}$

346: **upon** $\mathit{wchange}_i$:
347: $\mathit{wchange}_i \leftarrow \text{FALSE}$
348: **output** $\mathit{stable}_i(W_i)$

349: **upon** receiving msg. $\langle\text{PROBE}\rangle$ from C_j:
350: send message $\langle\text{VERSION}, \mathit{VER}_i[i]\rangle$ to C_j

351: **upon** receiving msg. $\langle\text{VERSION}, (V, M)\rangle$ from C_j:
352: $\mathit{update}(j, (V, M))$

353: **procedure** $\mathit{fail}()$:
354: send message $\langle\text{FAILURE}\rangle$ to all clients
355: **output** fail_i
356: **halt**

357: **upon** receiving $\mathit{USTOR.fail}_i$ **or** receiving a message $\langle\text{FAILURE}\rangle$ from C_j:
358: $\mathit{fail}()$

not necessarily contain a version committed by an operation of C_i.

Whenever C_i receives a version (V, M) from C_j, either in a response of the USTOR protocol or in a VERSION message, it calls a procedure *update* that checks (V, M) for consistency with the versions that it already knows. It suffices to verify that (V, M) is comparable to $\mathit{VER}_i[\mathit{max}_i]$ (line 336). Furthermore, when $\mathit{VER}_i[j] \dot{\leq} (V, M)$, then C_i updates $\mathit{VER}_i[j]$ to the bigger version (V, M).

The vector W_i in $\mathit{stable}_i(W_i)$ notifications contains the i-th entries of the timestamp vectors in VER_i, i.e., $W_i[j] = V_j[i]$, where $(V_j, M_j) = \mathit{VER}_i[j]$ for $j = 1, \ldots, n$. Hence, whenever the i-th entry in a timestamp vector in $\mathit{VER}_i[j]$ is larger than $W_i[j]$ after an update to $\mathit{VER}_i[j]$, then C_i updates $W_i[j]$ accordingly and issues a notification $\mathit{stable}_i(W_i)$. This means that all operations of FAUST at C_i that returned a timestamp $t \leq W[j]$ are stable w.r.t. C_j.

Note that C_i may receive a new maximal version from C_j by reading from X_j or by receiving a VERSION message directly from C_j. Although using client-to-client communication has been suggested before to detect server failures [20, 16], FAUST is the first algorithm in the context of untrusted storage to employ offline communication explicitly for detecting stability and for aiding progress when no inconsistency occurs.

The client detects server failures in one of three ways: first, the USTOR protocol may output *USTOR.fail*$_i$ if it detects any inconsistency in the messages from the server; second, procedure *update* checks that all versions received from other clients are comparable to the maximum of the versions in *VER*$_i$; and last, another client that has detected a server failure sends a FAILURE message via offline communication. When one of these conditions occurs, the client enters procedure *fail*, sends a FAILURE message to alert all other clients, outputs *fail*$_i$, and halts.

The following result summarizes the properties of the FAUST protocol.

Theorem 2. *Protocol FAUST in Algorithm 3 implements a fail-aware untrusted storage service consisting of n SWMR registers.*

Proof overview. A proof of the theorem appears in Appendix B; here we sketch its main ideas. Note that properties 1, 2, and 3 of Definition 6 immediately follow from the properties of the USTOR protocol: it is linearizable and wait-free whenever the server is correct, and weak fork-linearizable at all times. Property 4 (integrity) holds because subsequent operations of a client always commit versions with monotonically increasing timestamp vectors. Furthermore, the USTOR protocol never detects a failure when the server is correct, even when the server is arbitrarily slow, and the versions committed by its operations are monotonically increasing; this ensures property 5 (failure-detection accuracy).

We next explain why FAUST ensures property 6 of a fail-aware untrusted service (stability-detection accuracy). It is easy to see that any version returned by an extended operation of USTOR at C_i which is subsequently stored in $VER_i[i]$ is comparable to all other versions stored in VER_i. Additionally, we show (Lemma 22 in Appendix B) that every complete operation of the USTOR protocol at a client C_j that does not cause FAUST to output *fail*$_j$, commits a version that is comparable to $VER_i[j]$.

When combined, these two properties imply that when C_i receives a version from C_j that is larger than the version (V_i, M_i) committed by some operation o_i of C_i, then all versions committed by operations of C_j that do not fail are comparable to (V_i, M_i). Hence, when $(V_i, M_i) \dot{<} VER_i[j]$ and o_i becomes stable w.r.t. C_j, then C_j has promised, intuitively, to C_i that they have a common view of the execution up to o_i.

For property 7 (detection completeness), we show that every complete operation of FAUST at C_i eventually becomes stable with respect to every correct client C_j, unless a server failure is detected. Suppose that C_i and C_j are correct and that some operation o_i of C_i returned timestamp t. Under good conditions, when the server is correct and the network delivers messages in a timely manner, the FAUST protocol eventually causes C_j to read from X_i. Every subsequent operation of C_j then commits a version (V_j, M_j) such that $V_j[i] \geq t$. Since C_i also periodically reads all values, C_i eventually reads from X_j and receives such a version committed by C_j, and this causes o_i to become stable w.r.t. C_j.

However, it is possible that C_i does not receive a suitable version committed by C_j, which makes o_i stable w.r.t. C_j. This may be caused by network delays, which are indistinguishable to the clients from a server crash. At some point, C_i simply stops to receive new versions from C_j and, conversely, C_j receives no new versions from C_i. But at most δ time units later, C_j sends a PROBE message to C_i and eventually receives a VERSION message from C_i with a version (V_i, M_i) such that $V_i[i] \geq t$. Analogously, C_i eventually sends a PROBE message to C_j and receives a VERSION message containing some (V_j, M_j) from C_j with $V_j[i] \geq t$. This means that o_i becomes stable w.r.t. C_j.

7 Impossibility of Wait-Free Fork-*-Linearizable Byzantine Emulations

This section shows that fork-*-linearizable Byzantine emulations cannot be wait-free in all executions where the server is correct. This result implies the corresponding impossibility for fork-linearizable Byzantine emulations established before [4]. A similar result about fork-sequentially-consistent Byzantine emulations has been shown in a companion paper [3].

Theorem 3. *There is no protocol that emulates the functionality of $n \geq 1$ SWMR registers on a Byzantine server S with fork-*-linearizability that is wait-free in every execution with a correct S.*

Proof. Towards a contradiction, assume that there exists such an emulation protocol P. Then in any fair and well-formed execution of P with a correct server, every operation of a correct client completes. We next construct three executions of P, called α, β, and γ, with two clients, C_1 and C_2, accessing a single SWMR register X_1. All executions considered here are fair and well-formed, as can easily be verified. The clients are always correct.

We note that protocol P describes the asynchronous interaction of the clients with S. This interaction is depicted in the figures only when necessary.

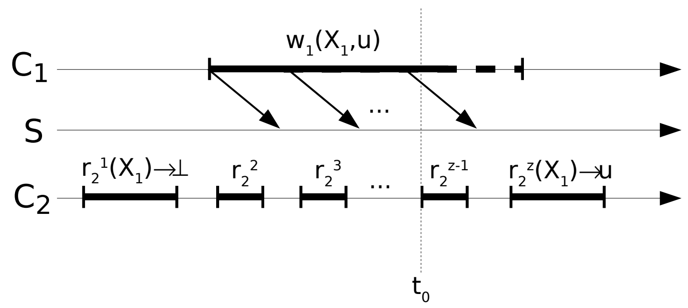

Figure 5: Execution α: S is correct.

Execution α. We construct an execution α, shown in Figure 5, in which S is correct. Client C_1 executes a write operation $\mathit{write}_1(X_1, u)$ and C_2 executes multiple read operations from X_1, denoted r_2^i for $i = 1, \dots, z$, as explained next.

The execution begins with C_2 invoking the first read operation r_2^1. Since S and C_2 are correct and we assume that P is wait-free in all executions when the server is correct, r_2^1 completes. Since C_1 did not yet invoke any operations, it must return the initial value $\perp$.

Next, C_1 invokes $w_1 = \mathit{write}_1(X_1, u)$. This is the only operation invoked by C_1 in α. Every time a message is sent from C_1 to S during w_1, if a non-$\perp$ value was not yet read by C_2 from X_1, then the following things happen in order: (a) the message from C_1 is delayed by the asynchronous network; (b) C_2 executes operation r_2^i reading from X_1, which completes by our wait-freedom assumption; (c) the message from C_1 to S is delivered. The operation w_1 eventually completes (and returns OK) by our wait-freedom assumption. After that point in time, C_2 invokes one more read operation from X_1 if and only if all its previous read operations returned $\perp$. According to the first property of fork-*-linearizable Byzantine emulations, since S is correct, this last read must return $u \neq \perp$ because it was invoked after w_1 completed. We denote the first read in α that returns a non-$\perp$ value by r_2^z (note that $z \geq 2$ since r_2^1 necessarily returns $\perp$ as explained above). By construction, r_2^z is the last operation of C_2 in α. We note that if messages are sent from C_1 to S after the completion of r_2^z, they are not delayed.

We denote by t_0 the invocation point of r_2^{z-1} in α. This point is marked by a vertical dashed line in Figures 5-7.

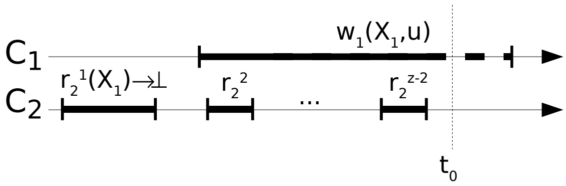

Figure 6: Execution β: S is correct.

Execution β. We next define execution β, in which S is also correct. The execution is shown in Figure 6. It is identical to α until the end of r_2^{z-2}, i.e., until just before point t_0 (as defined in α and marked by the dashed vertical line). In other words, execution β results from α by removing the last two read operations. If $z = 2$, this means that there are no reads in β, and otherwise r_2^{z-2} is the last operation of C_2 in β. Operation w_1 is invoked in β like in α; if β does not include r_2^1, then w_1 begins at the start of β, and otherwise, it begins after the completion of r_2^1. Since the server and C_1 are correct, by our wait-freedom assumption w_1 completes.

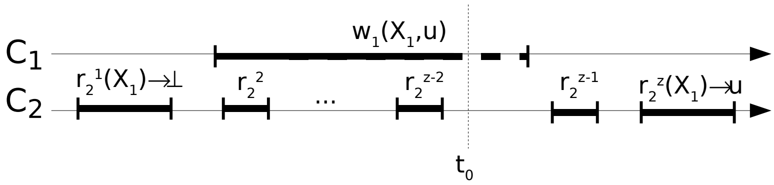

Figure 7: Execution γ: S is faulty. It is indistinguishable from α to C_2 and indistinguishable from β to C_1.

Execution γ. Our final execution is γ, shown in Figure 7, in which S is faulty. Execution γ begins just like the common prefix of α and β until immediately before point t_0, and w_1 begins in the same way as it does in β. In γ, the server simulates β to C_1 by hiding all operations of C_2, starting with r_2^{z-1}. Since C_1 cannot distinguish these two executions, w_1 completes in γ just like in β. After w_1 completes, the server simulates α for the two remaining reads r_2^{z-1} and r_2^z by C_2. We next explain how this is done. Notice that in α, the server receives at most one message from C_1 between t_0 and the completion of r_2^z, and this message is sent before time t_0 by our construction of α. In γ, which is identical to α until just before t_0, the same message (if any) is sent by C_1 and therefore the server has all needed information in order to simulate α for C_2 until the end of r_2^z. Hence, the output of r_2^{z-1} and r_2^z is the same as in α since it depends only on the state of C_2 before these operations and on the messages received from the server during their execution.

Thus, γ is indistinguishable from α to C_2 and indistinguishable from β to C_1. However, we next show that γ is not fork-*-linearizable. Observe the sequential permutation π_2 required by the definition of fork-*-linearizability (i.e., the view of C_2). As the sequential specification of X_1 must be preserved in π_2, and since r_2^z returns u, we conclude that w_1 must appear in π_2. Since the real-time order must be preserved as well, the write appears before r_2^{z-1} in the view. However, this violates the sequential specification of X_1, since r_2^{z-1} returns $\perp$ and not the most recently written value $u \neq \perp$. This contradicts the definition of P as a protocol that guarantees fork-*-linearizability in all executions. $\square$

8 Comparing Forking Consistency Conditions and Causal Consistency

The purpose of this section is to explore the relation between causal consistency and the forking consistency notions introduced in Section 4.1. First, we show that fork-linearizability implies causal consistency.

Theorem 4. *Every fork-linearizable history w.r.t. a functionality F of composed of registers is also causally consistent w.r.t. F.*

Proof. Consider a fork-linearizable execution σ. We will show that the views of the clients satisfying the definition of fork-linearizability also preserve the requirement of causal consistency, which is that for each operation in every client's view, all write operations that causally precede it appear in the view before the particular operation. More formally, let π_i be some view of σ at a client C_i according to fork-linearizability and let o be an operation in π_i. We need to prove that any write operation o' that causally precedes o appears in π_i before o. According to the definition of causal order, this can be proved by repeatedly applying the following two arguments.

First, assume that both o and o' are operations by the same client C_j and consider a view π_j at C_j. Since π_j includes all operations by C_j, also o and o' appear in π_j. Since o' precedes o and since π_j preserves the real-time order of σ according to fork-linearizability, operation o' precedes o also in π_j. By the no-join condition, we have that $\pi_i^o = \pi_j^o$ and, therefore, o' also appears before o in π_i.

Second, assume that o' is of the form $\mathit{write}_j(X, v)$ and o is of the form $\mathit{read}_k(X) \to v$. In this case, operation o' is contained in π_i and precedes o because π_i is a view of σ at C_i; in particular, the third property of a view guarantees that π_i satisfies the sequential specification of a register. □

The next two theorems establish that causal-consistency and fork-*-linearizability are incomparable, in the sense that neither notion implies the other one if we consider a storage service with multiple SWMR registers.

The definition of weak fork-linearizability implies trivially that every weakly fork-linearizable history is also causally consistent. The next theorem shows that a fork-*-linearizable history may not be causally consistent with respect to functionalities with multiple registers.

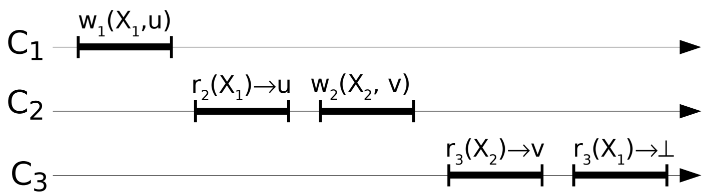

Figure 8: A fork-*-linearizable history that is not causally consistent.

Theorem 5. *There exist histories that are fork-*-linearizable but not causally consistent w.r.t. a functionality containing two or more registers.*

Proof. Consider the following execution, shown in Figure 8: Client C_1 executes $\mathit{write}_1(X_1, u)$, then client C_2 executes $\mathit{read}_2(X_1) \to u$, $\mathit{write}_2(X_2, v)$, and finally, client C_3 executes $\mathit{read}_3(X_2) \to v$, $\mathit{read}_3(X_1) \to \bot$. Define the client views according to the definition of fork-*-linearizability as

$$\begin{aligned}
\pi_1 :&\quad \mathit{write}_1(X_1, u).\\
\pi_2 :&\quad \mathit{write}_1(X_1, u),\ \mathit{read}_2(X_1) \to u,\ \mathit{write}_2(X_2, v).\\
\pi_3 :&\quad \mathit{write}_2(X_2, v),\ \mathit{read}_3(X_2) \to v,\ \mathit{read}_3(X_1) \to \bot.
\end{aligned}$$

It is easy to see that π_1, π_2, and π_3 satisfy the conditions of fork-*-linearizability. In particular, since no two operations of any client appear in two views, the at-most-one-joint condition holds trivially. But clearly, α is not causally consistent: $\mathit{write}_1(X_1, u)$ causally precedes $\mathit{write}_2(X_2, v)$ which itself causally precedes $\mathit{read}_3(X_1) \rightarrow \perp$; thus, returning $\perp$ violates the sequential specification of a read/write register. □

Conversely, a causally consistent history may not be fork-*-linearizable with respect to even one register.

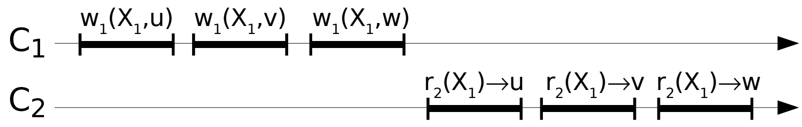

Figure 9: A causally consistent execution that is not fork-*-linearizable.

Theorem 6. *There exist histories that are causally consistent but not fork-*-linearizable with respect to a functionality with one register.*

Proof. Consider the following execution, shown in Figure 9: Client C_1 executes three write operations, $\mathit{write}_1(X_1, u)$, $\mathit{write}_1(X_1, v)$, and $\mathit{write}_1(X_1, w)$. After the last one completes, client C_2 executes three read operations, $\mathit{read}_2(X_1) \rightarrow u$, $\mathit{read}_2(X_1) \rightarrow v$, and $\mathit{read}_2(X_1) \rightarrow w$. We claim that this execution is causally consistent. Intuitively, the causally dependent write operations are seen in the same order by both clients. More formally, the view of C_1 according to the definition of causal consistency contains only operations of C_1, and the view of C_2 contains all operations, with the write and read operations interleaved so that they satisfy the sequential specification; this is consistent with the causal order of the execution.

However, the execution is not fork-*-linearizable, as we explain next. The view π_2 of C_2, as required by the definition of fork-*-linearizability, must be the sequence:

$$\mathit{write}_1(X_1, u),\ \mathit{read}_2(X_1) \rightarrow u,\ \mathit{write}_1(X_1, v),\ \mathit{read}_2(X_1) \rightarrow v,\ \mathit{write}_1(X_1, w),\ \mathit{read}_2(X_1) \rightarrow w.$$

But the operations $\mathit{read}_2(X_1) \rightarrow u$ and $\mathit{write}_1(X_1, v)$ violate the real-time order requirement of fork-*-linearizability. □

9 Conclusion

We tackled the problem of providing meaningful semantics for a service implemented by an untrusted provider. As clients increasingly use online services provided by third parties, such as in cloud computing, the importance of addressing this problem becomes more prominent. For such environments, we presented the new abstraction of a fail-aware untrusted service. This notion generalizes the concepts of eventual consistency and fail-awareness to account for Byzantine faults. We realize this new abstraction in the context of an online storage service with so-called forking semantics. Our service guarantees linearizability and wait-freedom when the server is correct, provides accurate and complete consistency and failure notifications, and ensures causal consistency at all times. We observed that no previous forking consistency notion can be used for building fail-aware untrusted storage, because these notions inherently rule out wait-free implementations. We then presented a new forking consistency condition called weak fork-linearizability, which does not suffer from this limitation. We developed an efficient wait-free protocol for implementing fail-aware untrusted storage with weak fork-linearizability. Finally, we used this untrusted storage protocol to implement fail-aware untrusted storage.

Two problems are left open by this work. First, we did not consider Byzantine client faults. However, the USTOR and FAUST protocols can be extended to deal with such behavior with known methods [20]. Most problems can be avoided by having the clients verify that their peers provide consistent information about past operations. Such methods are orthogonal to our contributions, however, and a precise formulation of the semantics that can be achieved are beyond this work. Second, our protocol require a communication complexity proportional to the number of clients. It remains open to determine if this is an inherent limitation of the model and, potentially, to find more scalable solutions.

Acknowledgments

We thank Alessia Milani, Dani Shaket, Marko Vukolić, and the anonymous reviewers for their valuable comments.

This work is partially supported by the European Commission through the IST Programme under Contract IST-4-026764-NOE ReSIST.

References

[1] R. Baldoni, A. Milani, and S. T. Piergiovanni. Optimal propagation-based protocols implementing causal memories. *Distributed Computing*, 18(6):461–474, 2006.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1987.

[3] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *Inf. Process. Lett.*, 109(7):360–364, 2009.

[4] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2007.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[6] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 189–204, 2007.

[7] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 205–220, 2007.

[9] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 314–321, 1996.

[10] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. 21st ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, 2007.

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

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

[13] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *Proc. 10th Intl. Conference on Distributed Computing Systems (ICDCS)*, pages 302–309, 1990.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[16] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

[17] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

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

[19] T. Marian, M. Balakrishnan, K. Birman, and R. van Renesse. Tempest: Soft state replication in the service tier. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 227–236, 2008.

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

[21] A. Oprea and M. K. Reiter. On consistency of encrypted files. In S. Dolev, editor, *Proc. 20th Intl. Conference on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 254–268, 2006.

[22] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005.

[23] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, , and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *Proc. 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[24] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, pages 172–182, 1995.

[25] J. Yang, H. Wang, N. Gu, Y. Liu, C. Wang, and Q. Zhang. Lock-free consistency control for Web 2.0 applications. In *Proc. 17th Intl. Conference on World Wide Web (WWW)*, pages 725–734, 2008.

[26] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3), 2007.

A Analysis of the Weak Fork-Linearizable Untrusted Storage Protocol

This section is devoted to the proof of Theorem 1. We start with some lemmas that explain how the versions committed by clients should monotonically increase during the protocol execution.

Lemma 7 (Transitivity of order on versions). *Consider three versions (V_i, M_i), (V_j, M_j), and (V_k, M_k). If $(V_i, M_i) \dot{\leq} (V_j, M_j)$ and $(V_j, M_j) \dot{\leq} (V_k, M_k)$, then $(V_i, M_i) \dot{\leq} (V_k, M_k)$.*

Proof. First, $V_i \leq V_j$ and $V_j \leq V_k$ implies $V_i \leq V_k$ because the order on timestamp vectors is transitive. Second, let c be any index such that $V_i[c] = V_k[c]$. Since $V_i[c] \leq V_j[c]$ and $V_j[c] \leq V_k[c]$, but $V_i[c] = V_k[c]$, we have $V_j[c] = V_k[c]$. From $(V_i, M_i) \dot{\leq} (V_j, M_j)$ it follows that $M_i[c] = M_j[c]$. Analogously, it follows that $M_j[c] = M_k[c]$, and hence $M_i[c] = M_k[c]$. This means that $(V_i, M_i) \dot{\leq} (V_k, M_k)$. □

Lemma 8. *Let o_i be an operation of C_i that commits a version (V_i, M_i) and suppose that during its execution, C_i receives a* REPLY *message containing a version (V^c, M^c). Then $(V^c, M^c) \dot{<} (V_i, M_i)$.*

Proof. We first prove that $(V^c, M^c) \dot{\leq} (V_i, M_i)$. According to the order on versions, we have to show that for all $k = 1, \ldots, n$, we have either $V^c[k] < V_i[k]$ or $V^c[k] = V_i[k]$ and $M^c[k] = M_i[k]$. Note how the computation of (V_i, M_i) starts from $(V_i, M_i) = (V^c, M^c)$ (line 138); later, an entry $V_i[k]$ is either incremented (lines 143 and 147), hence $V^c[k] < V_i[k]$, or not modified, and then $M^c[k] = M_i[k]$. Moreover, $V_i[i]$ is incremented exactly once, and therefore $(V^c, M^c) \neq (V_i, M_i)$ □

Lemma 9. *Let o'_i and o_i be two operations of C_i that commit versions (V'_i, M'_i) and (V_i, M_i), respectively, such that o'_i precedes o_i. Then:*

1. *o'_i and o_i are consecutive operations of C_i if and only if $V'_i[i] + 1 = V_i[i]$; and*
2. *$(V'_i, M'_i) \dot{<} (V_i, M_i)$.*

Proof. At the start of o_i, client C_i remembers the most recent version (V'_i, M'_i) that it committed. During the execution of o'_i, C_i receives from S a version (V^c, M^c) and verifies that $V'_i[i] = V^c[i]$ (line 137) and sets $V_i = V'_i$. Afterwards, C_i increments $V_i[i]$ (line 147) exactly once (as guarded by the check on line 144). This establishes the first claim of the lemma. The second claim follows from the check $(V'_i, M'_i) \dot{\leq} (V^c, M^c)$ (line 137) and from Lemma 8 by transitivity of the order on versions. □

The next lemma addresses the situation where a client executes a read operation that returns a value written by a preceding operation or a concurrent operation.

Lemma 10. *Suppose o_i is a read operation of C_i that reads a value x from register X_j and commits version (V_i, M_i). Then the version (V_0^j, M_0^j) that C_i receives with x in the* REPLY *message satisfies $(V_0^j, M_0^j) \dot{<} (V_i, M_i)$. Moreover, suppose o_j is the operation of C_j that writes x. Then all operations of C_j that* precede *o_j commit a version smaller than (V_i, M_i).*

Proof. Let (V^c, M^c) be the version that C_i receives during o_i in the REPLY message, together with (V_0^j, M_0^j), which was committed by an operation o_0^j of C_j (line 150). In procedure *checkData*, C_i verifies that $(V_0^j, M_0^j) \dot{\leq} (V^c, M^c)$; Lemma 8 shows that $(V^c, M^c) \dot{<} (V_i, M_i)$; hence, we have that $(V_0^j, M_0^j) \dot{<} (V_i, M_i)$ from the transitivity of the order on versions. Because the timestamp t^j that was signed together with x under the DATA-signature (line 151) is equal to $V_0^j[j]$ or to $V_0^j[j] + 1$ (line 153), it follows from Lemma 9 that either o_j precedes o_0^j, or o_j is equal to o_0^j, or o_0^j immediately precedes o_j. In either case, the claim follows. □

We now establish the connection between the view history of an operation and the digest vector in the version committed by that operation.

Lemma 11. *Let o_i be an operation invoked by C_i that commits version (V_i, M_i). Furthermore, if $V_i[j] > 0$, let ω denote the operation of C_j with timestamp $V_i[j]$; otherwise, let ω denote an imaginary initial operation $o_\perp$. Then $M_i[j]$ is equal to the digest of the prefix of $\mathcal{VH}(o_i)$ up to ω, i.e.,*

$$M_i[j] = D\big(\mathcal{VH}(o_i)|^{\omega}\big).$$

Proof. We prove the lemma by induction on the construction of the view history of o_i. Consider operation o_i executed by C_i and the REPLY message from S that C_i receives, which contains a version (V^c, M^c). The base case of the induction is when $(V^c, M^c) = (0^n, \perp^n)$. The induction step is the case when (V^c, M^c) was committed by some operation o_c of client C_c.

For the base case, note that for any j, it holds $M^c[j] = \perp$, and this is equal to the digest of an empty sequence. During the execution of o_i in *updateVersion*, the version (V_i, M_i) is first set to (V^c, M^c) (line 138) and the digest d is set to $M^c[c]$. Let us investigate how V_i and M_i change subsequently.

If $j \neq i$, then $V_i[j]$ and $M_i[j]$ change only when an operation by C_j is represented in L. If there is such an operation, C_i computes $d = D\big(\mathcal{VH}(o_i)|^{\omega}\big)$ and sets $M_i[j]$ to d by the end of the loop (lines 140–146). In other words, the loop starts at the same position and cycles through the same sequence of operations $\omega^1, \ldots \omega^m$ as the one used to define the view history. This establishes the claim when ω is the operation of C_j with timestamp $V_i[j]$.

If $i = j$, then the test in line 144 ensures that there is no operation by C_j represented in L. After the execution of the loop, $V_i[i]$ is incremented (line 147), the invocation tuple of o_i is included into the digest at the position corresponding to the definition of the view history, and the result stored in $M_i[i]$. Hence, $M_i[i] = D\big(\mathcal{VH}(o_i)\big)$ and the claim follows also for $\omega = o_i$.

For the induction step, note that $M^c[c] = D\big(\mathcal{VH}(o_c)\big)$ by the induction assumption. For any j such that $V^c[j] = V_i[j]$, the claim holds trivially from the induction assumption. During the execution of o_i in *updateVersion*, the reasoning for the base case above applies analogously. Hence, the claim holds also for the induction step, and the lemma follows. □

Lemma 12. *Let o_i be an operation that commits version (V_i, M_i) such that $V_i[j] > 0$ for some $j \in \{1, \ldots, n\}$. Then the operation of C_j with timestamp $V_i[j]$ is contained in $\mathcal{VH}(o_i)$.*

Proof. Consider the first operation $\tilde{o} \in \mathcal{VH}(o_i)$ that committed a version $(\tilde{V}, \tilde{M})$ such that $\tilde{V}[j] = V_i[j]$. According to the test on line 144, the operation of C_j with timestamp $V_i[j]$ is concurrent to $\tilde{o}$ and therefore is contained in $\mathcal{VH}(o_i)$ by construction. □

Lemma 13. *Consider two operations o_i and o_j that commit versions (V_i, M_i) and (V_j, M_j), respectively, such that $V_i[k] = V_j[k] > 0$ for some $k \in \{1, \ldots, n\}$, and let o_k be the operation of C_k with timestamp $V_i[k]$. Then $M_i[k] = M_j[k]$ if and only if $\mathcal{VH}(o_i)|^{o_k} = \mathcal{VH}(o_j)|^{o_k}$.*

Proof. By Lemma 12, o_k is contained in the view histories of o_i and o_j. Applying Lemma 11 to both sides of the equation $M_i[k] = M_j[k]$ gives

$$D\big(\mathcal{VH}(o_i)|^{o_k}\big) = M_i[k] = M_j[k] = D\big(\mathcal{VH}(o_j)|^{o_k}\big).$$

Because of the collision resistance of the hash function in the digest function, two outputs of D are only equal if the respective inputs are equal. The claim follows. □

We introduce another data structure for the analysis. The *commit history* $\mathcal{CH}(o)$ of an operation o is a sequence of operations, defined as follows. Client C_i executing o receives a REPLY message from

S that contains a timestamp vector V^c, which is either equal to 0^n or comes together with a COMMIT-signature φ^c by C_c, corresponding to some operation o_c of C_c. Then we set

$$\mathcal{CH}(o) \triangleq \begin{cases} o & \text{if } V^c = 0^n \\ \mathcal{CH}(o_c), o & \text{otherwise.} \end{cases}$$

Clearly, $\mathcal{CH}(o)$ is a subsequence of $\mathcal{VH}(o)$; the latter also includes all concurrent operations.

Lemma 14. *Consider two consecutive operations o^μ and $o^{\mu+1}$ in a commit history and the versions (V^μ, M^μ) and $(V^{\mu+1}, M^{\mu+1})$ committed by o^μ and $o^{\mu+1}$, respectively. For $k = 1, \ldots, n$, it holds $V^{\mu+1}[k] \leq V^\mu[k] + 1$.*

Proof. The lemma follows easily from the definition of a commit history and from the statements in procedure *updateVersion* during the execution of $o^{\mu+1}$, because $V^{\mu+1}$ is initially set to V^μ (line 138) and $V^{\mu+1}[k]$ is incremented (line 143) at most once for every k. □

The purpose of the versions in the protocol is to order the operations if the server is faulty. When a client executes an operation, the view history of the operation represents the impression of the past operations that the server provided to the client. But if an operation o_j that committed (V_j, M_j) is contained in $\mathcal{VH}(o_i)$, where o_i committed (V_i, M_i), this does not mean that $(V_j, M_j) \dot{\leq} (V_i, M_i)$. Such a relation holds only when $\mathcal{VH}(o_j)$ is also a prefix of $\mathcal{VH}(o_i)$, as the next lemma shows.

Lemma 15. *Let o_i and o_j be two operations that commit versions (V_i, M_i) and (V_j, M_j), respectively. Then $(V_j, M_j) \dot{\leq} (V_i, M_i)$ if and only if $\mathcal{VH}(o_j)$ is a prefix of $\mathcal{VH}(o_i)$.*

Proof. To show the forward direction, suppose that $(V_j, M_j) \dot{\leq} (V_i, M_i)$. Clearly, $V_j[j] > 0$ because C_j completed o_j and $V_j[j] \leq V_i[j]$ according to the order on versions. In the case that $V_j[j] = V_i[j]$, the assumption of the lemma implies that $M_j[j] = M_i[j]$ by the order on versions. The claim now follows directly from Lemma 13.

It is left to show the case $V_j[j] < V_i[j]$. Let o_m be the first operation in $\mathcal{CH}(o_i)$ that commits a version (V_m, M_m) such that $V_m[j] > V_j[j]$; let o_c be the operation that precedes o_m in its commit history and suppose o_c commits (V^c, M^c). Note that $V^c[j] \leq V_j[j]$. According to Lemma 14, we have $V^c[j] = V_j[j] = V_m[j] - 1$.

Let o'_j be the operation of C_j with timestamp $V_j[j] + 1$. Note that o_j and o'_j are two consecutive operations of C_j according to Lemma 9. There are two possibilities for the relation between o'_j and o_m:

Case 1: If $o'_j = o_m$, then we observe from the definitions of view histories and commit histories that $\mathcal{VH}(o'_j)$ is a prefix of $\mathcal{VH}(o_i)$. We only have to prove that $\mathcal{VH}(o_j)$ is a prefix of $\mathcal{VH}(o'_j)$.

According to the protocol, C_j verifies that $V^c[j] = V_j[j] > 0$ and that $(V_j, M_j) \dot{\leq} (V^c, M^c)$ (line 137). By the definition of the order on versions, we get $M^c[j] = M_j[j]$. Lemma 13 now implies that $\mathcal{VH}(o_j)$ is a prefix of $\mathcal{VH}(o_c)$, which, in turn, is a prefix of $\mathcal{VH}(o'_j)$ according to the definition of view histories, and the claim follows.

Case 2: If o'_j was a concurrent operation to o_m, then the invocation tuple of o'_j was contained in L received by the client executing o_m, and the client verified the PROOF-signature by C_j in $P[j]$ from operation o_j on $M^c[j]$. If the verification succeeds, we know that $M^c[j] = D\big(\mathcal{VH}(o_j)\big)$ according to Lemma 11. According to the verification of the SUBMIT-signature from C_j on $V^c[j]$, we have $V_j[j] = V^c[j] > 0$ (line 144); hence, Lemma 13 implies that $\mathcal{VH}(o_j)$ is a prefix of $\mathcal{VH}(o_c)$ and the claim follows because $\mathcal{VH}(o_c)$ is a prefix of $\mathcal{VH}(o_i)$ by the definition of view histories.

To prove the backward direction, suppose that $(V_j, M_j) \dot{\not\leq} (V_i, M_i)$. There are two possibilities for this comparison to fail: there exists a k such that either $V_j[k] > V_i[k]$ or that $V_i[k] = V_j[k]$ and $M_i[k] \neq M_j[k]$.

In the first case, Lemma 12 shows that there exists an operation o_k by client C_k in $\mathcal{VH}(o_j)$ that is not contained in $\mathcal{VH}(o_i)$. Thus, $\mathcal{VH}(o_j)$ is not a prefix of $\mathcal{VH}(o_i)$.

In the second case, Lemma 13 implies that $\mathcal{VH}(o_i)|^{o_k}$ is different from $\mathcal{VH}(o_j)|^{o_k}$, and, again, $\mathcal{VH}(o_j)$ is not a prefix of $\mathcal{VH}(o_i)$. This concludes the proof. □

This result connects the versions committed by two operations to their view histories and shows that the order relation on committed versions is isomorphic to the prefix relation on the corresponding view histories. The next lemma contains a useful formulation of this property.

Lemma 16 (No-join). *Let o_i and o_j be two operations that commit versions (V_i, M_i) and (V_j, M_j), respectively. Suppose that (V_i, M_i) and (V_j, M_j) are incomparable, i.e., $(V_i, M_i) \dot{\not\leq} (V_j, M_j)$ and $(V_j, M_j) \dot{\not\leq} (V_i, M_i)$. Then there is no operation o_k that commits a version (V_k, M_k) that satisfies $(V_i, M_i) \dot{\leq} (V_k, M_k)$ and $(V_j, M_j) \dot{\leq} (V_k, M_k)$.*

Proof. Suppose for the purpose of reaching a contradiction that there exists such an operation o_k. From Lemma 15, we know that $\mathcal{VH}(o_i)$ and $\mathcal{VH}(o_j)$ are not prefixes of each other. But the same lemma also implies that $\mathcal{VH}(o_i)$ is a prefix of $\mathcal{VH}(o_k)$ and that $\mathcal{VH}(o_j)$ is a prefix of $\mathcal{VH}(o_k)$. This is only possible if one of $\mathcal{VH}(o_i)$ and $\mathcal{VH}(o_j)$ is a prefix of the other, and this contradicts the previous statement. □

We are now ready to prove that our algorithm emulates a storage service of n SWMR registers on an untrusted server with weak fork linearizability. We do this in two steps. The first theorem below shows that the protocol execution with a correct server is linearizable and wait-free. The second theorem below shows that the protocol preserves weak fork-linearizability even with a faulty server. Together they imply Theorem 1.

Theorem 17. *In every fair and well-formed execution with a correct server:*

1. *Every operation of a correct client is complete; and*
2. *The history is linearizable w.r.t. n SWMR registers.*

Proof. Consider a fair and well-formed execution σ of protocol USTOR where S is correct. We first show that every operation of a correct client is complete. According to the protocol for S, every client that sends a SUBMIT message eventually receives a REPLY message from S. This follows because the parties use reliable FIFO channels to communicate, the server processes arriving messages atomically and in FIFO order, and at the end of processing a SUBMIT message, the server sends a REPLY message to the client.

It remains to show that a correct client does not halt upon receiving the REPLY message and therefore satisfies the specification of the functionality. We now examine all checks by C_i in Algorithm 1 and explain why they succeed when S is correct.

The COMMIT-signature on the version (V^c, M^c) received from S is valid because S sends it together with the version that it received from the signer (line 136). For the same reason, also the COMMIT-signature on (V^j, M^j) (line 150) and the DATA-signature on t^j and $H(x^j)$ (line 151) are valid.

Suppose C_i executes operation o_i. In order to see that $(V_i, M_i) \dot{\leq} (V^c, M^c)$ and $V_i[i] = V^c[i]$ (line 137), consider the schedule constructed by S: The schedule at the point in time when S receives the SUBMIT message corresponding to o_i is equal to the view history of o_i. Moreover, the version committed by any operation scheduled before o_i is smaller than the version committed by o_i.

According to Algorithm 2, S keeps track of the last operation in the schedule for which it has received a COMMIT message and stores the index of the client who executed this operation in c (line 203).

Note that *SVER*$[c]$ holds the version (M^c, V^c) committed by this operation. Therefore, when C_i receives a REPLY message from S containing (M^c, V^c), the check $(V_i, M_i) \dot{\leq} (V^c, M^c)$ succeeds since the preceding operation of C_i already committed (V_i, M_i). This preceding operation is in $\mathcal{VH}(o_i)$ by Lemma 12; moreover, it is the last operation of C_i in the schedule, and therefore, $V_i[i] = V^c[i]$.

Next, we examine the verifications in the loop that runs through the concurrent operations represented in L (lines 140–146). Suppose C_i is verifying an invocation tuple representing an operation o_k of C_k. It is easy to see that the PROOF-signature of C_k in $P[k]$ was created during the most recent operation o'_k of C_k that precedes o_k, because C_k and S communicate using a reliable FIFO channel and, therefore, the COMMIT message of o'_k has been processed by S before the SUBMIT message of o_k. It remains to show that the value $M_i[k]$, on which the signature is verified (line 142), is equal to $M'_k[k]$, where (M'_k, V'_k) is the version committed by o'_k. Since o'_k is the last operation by C_k in the schedule before o_c, it holds $V'_k[k] = V^c[k]$. Furthermore, it holds $(V'_k, M'_k) \dot{\leq} (V^c, M^c)$ and this means that $M'_k[k] = M^c[k]$ by the order on versions. Since M_i is set to M^c before the loop (line 138), we have that $M_i[k] = M^c[k] = M'_k[k]$ and the verification of the PROOF-signature succeeds.

Extending this argument, since $V^c[k]$ holds the timestamp of o'_k, the timestamp of o_k is $V^c[k] + 1$, and thus the SUBMIT-signature of o_k is valid (line 144). Since no operation of C_i that precedes o_i occurs in the schedule after o_c, and since L includes only operations that occur in the schedule after o_c (according to line 220), no operation by C_i is represented in L. Therefore, the check that $k \neq i$ succeeds (line 144).

For a read operation from X_j, client C_i receives the timestamp t^j and the value x^j, together with a version (V^j, M^j) committed some operation o_j of C_j. Consider the operation o_w of C_j that writes x^j. It may be that $o_w = o_j$ if S has received its COMMIT message before the read operation. But since C_j sends the timestamp and the value with the SUBMIT message to S, it may also be that o_j precedes o_w. C_i first verifies that $(V^j, M^j) \dot{\leq} (V^c, M^c)$, and this holds because (V^c, M^c) was committed by the last operation in the schedule (line 152). Furthermore, C_i checks that $t^j = V_i[j]$ (line 152); because both values correspond to the timestamp of the last operation by C_j scheduled before o_i, the check succeeds. Finally, C_i verifies that (V^j, M^j) is consistent with t^j: if $o_w = o_j$, then $V^j[j] = t^j$; otherwise, o_w is the subsequent operation of C_j after o_j, and $V^j[j] = t^j - 1$ (line 153).

For the proof of the second claim, we have to show that the schedule constructed by S satisfies the two conditions of linearizability. First, the schedule preserves the real-time order of σ because any operation o that precedes some operation o' is also scheduled before o', according to the instructions for S. Second, every read operation from X_j returns the value written either by the most recent completed write operation of C_j or by a concurrent write operation of C_j. □

Let σ be the history of a fair and well-formed execution of the protocol. The definition of weak fork-linearizability postulates the existence of sequences of events π_i for $i = 1, \ldots, n$ such that π_i is a view of σ at client C_i. We construct π_i in three steps:

1. Let o_i be the last complete operation of C_i in σ and suppose it committed version (V_i, M_i). Define α_i to be the set of all operations in σ that committed a version smaller than or equal to (V_i, M_i).
2. Define β_i to be the set of all operations o_j of the form $\textit{write}_j(X_j, x)$ from $\sigma \setminus \alpha_i$ for any x such that α_i contains a read operation returning x. (Recall that written values are unique.)
3. Construct a sequence ρ_i from α_i by ordering all operations in α_i according to the versions that these operations commit, in ascending order. This works because all versions are smaller than (V_i, M_i) by construction of α_i, and, hence, totally ordered by Lemma 16. Next, we extend ρ_i to π_i by adding the operations in β_i as follows. For every $o_j \in \beta_i$, let x be the value that it writes; insert o_j into π_i immediately before the first read operation that returns x.

Theorem 18. *The history of every fair and well-formed execution of the protocol is weakly fork-linearizable w.r.t. n SWMR registers.*

Proof. We use $\alpha_i, \beta_i, \rho_i$, and π_i as defined above.

Claim 18.1. *Consider some π_i and let $o_j, o'_j \in \sigma$ be two operations of client C_j such that $o'_j \in \pi_i$. Then $o_j <_\sigma o'_j$ if and only if $o_j \in \alpha_i$ and $o_j <_{\pi_i} o'_j$.*

Proof. To show the forward direction, we distinguish two cases. If $o'_j \in \beta_i$, then it must be a write operation and there is a read operation o_k in α_i that returns the value written by o'_j. According to Lemma 10, any other operation of C_j that precedes o'_j commits a version smaller than the version committed by o_k. In particular, this applies to o_j. Since $o_k \in \alpha_i$, we also have $o_j \in \alpha_i$ by construction and $o_j <_{\pi_i} o_k$ since π_i contains the operations of α_i ordered by the versions that they commit. Moreover, because o'_j appears in π_i immediately before o_k, it follows that $o_j <_{\pi_i} o'_j$.

If $o'_j \notin \beta_i$, on the other hand, then $o'_j \in \alpha_i$, and Lemma 9 shows that o_j commits a version that is smaller than the version committed by o'_j. Hence, by construction of α_i, we have that $o_j \in \alpha_i$ and $o_j <_{\pi_i} o'_j$.

To establish the reverse implication, we distinguish the same two cases as above. If $o'_j \in \beta_i$, then then it must be a write operation and there is a subsequent read operation $o_k \in \alpha_i$ that returns the value written by o'_j. Since $o_j \in \alpha_i$ by assumption and $o_j <_{\pi_i} o_k$, it must be that the version committed by o_j is smaller than the version committed by o_k because the operations of ρ_i are ordered according to the versions that they commit. Hence, $o_j <_\sigma o'_j$ by Lemma 9.

If $o'_j \notin \beta_i$, on the other hand, then $o'_j \in \alpha_i$. Since the operations of ρ_i are ordered according to the versions that they commit, the version committed by o_j is smaller than the version committed by o'_j. Lemma 9 now implies that $o_j <_\sigma o'_j$. □

Recall the function *lastops*(π_i) from the definition of weak real-time order, denoting the last operations of all clients in π_i.

Claim 18.2. *For any π_i, we have that $\beta_i \subseteq$ lastops(π_i).*

Proof. We have to show that operation $o_j \in \beta_i$ invoked by C_j is the last operation of C_j in π_i. Towards a contradiction, suppose there is another operation o^*_j of C_j that appears in π_i after o_j. Because the execution is well-formed, operations o_j and o^*_j are not concurrent. If $o_j <_\sigma o^*_j$, then Claim 18.1 implies that $o_j \in \alpha_i$, contradicting the assumption $o_j \in \beta_i$. On the other hand, if $o^*_j <_\sigma o_j$, then Claim 18.1 implies that $o^*_j <_{\pi_i} o_j$. Since each operation appears at most once in π_i, this contradicts the assumption on o^*_j. □

The next claim is only needed for the proof of Theorem 2 in Appendix B.

Claim 18.3. *Let o'_i be a complete operation of C_i, let o_k be any operation in $\pi_i|^{o'_i}$, let (V'_i, M'_i) be the version committed by o'_i, and let o_j be an operation that commits version (V_j, M_j) such that $(V'_i, M'_i) \dot{\leq} (V_j, M_j)$. Then o_k is invoked before o_j completes.*

Proof. Suppose o_k commits version (V_k, M_k). If $o_k \in \alpha_i$, then $(V_k, M_k) \dot{\leq} (V'_i, M'_i)$ by construction of α_i, and in particular $V'_i[k] \geq V_k[k]$. If $o_k \in \beta_i$, then there exists some read operation $o_r \in \alpha_i$ that commits $(V_r, M_r) \dot{\leq} (V'_i, M'_i)$ and returns the value written by o_k. Thus, $V'_i[k] \geq V_r[k] \geq V_k[k]$. In both cases, we have that $V'_i[k] \geq V_k[k]$. Since $V_j \geq V'_i$, we conclude that $V_j[k] \geq V_k[k] > 0$. According to the protocol logic, this means that o_k is invoked before o_j, and in particular before o_j completes. □

Claim 18.4. *π_i is a view of σ at C_i w.r.t. n SWMR registers.*

Proof. The first requirement of a view holds by construction of π_i.

We next show the second requirement of a view, namely that all complete operations in $\sigma|_{C_i}$ are contained in π_i. Because the o_i is the last complete operation of C_i, and all other operations of C_i commit smaller versions by Lemma 9, the statement follows immediately from Lemma 15.

Finally, we show that the operations of π_i satisfy the sequential specification of n SWMR registers. The specification requires for every read operation $o_r \in \pi_i$, which returns a value x written by an operation o_w of C_w, that o_w appears in π_i before o_r, and there must not be any other write operation by C_w in π_i between o_w and o_r.

Suppose o_r is executed by C_r and commits version (V_r, M_r); note that C_r in *checkData* makes sure that $V_r[w]$ is equal to the timestamp t that C_r receives together with the data (according to the verification of the DATA-signature in line 151 and the check in line 152). Since β_i contains only write operations, we conclude that $o_r \in \alpha_i$. Let o'_w be the operation of C_w with timestamp t. According to the protocol, o'_w is either equal to o_w or the last one in a sequence of read operations executed by C_w immediately after o_w.

We distinguish between two cases with respect to o'_w. The first case is $o'_w \in \beta_i$. Then $o'_w = o_w$ and o'_w appears in π_i immediately before the first read operation that returns x, and o'_w is the last operation of C_w in π_i as shown by Claim 18.2. Therefore, no further write operation of C_w appears in π_i and the sequential specification of the register holds.

The second case is $o'_w \in \alpha_i$; suppose o'_w commits version (V'_w, M'_w), where $V'_w[w] = t$ by definition. Lemma 12 shows that $o'_w \in \mathcal{VH}(o_r)$. Because o_r and o'_w are in α_i, versions (V_r, M_r) and (V'_w, M'_w) are ordered and we conclude from Lemma 15 that this is only possible when $(V'_w, M'_w) \dot{<} (V_r, M_r)$. Therefore, o'_w appears in π_i before o_r by construction.

We conclude the argument for the second case by showing that there is no further write operation by C_w between o'_w and o_r in π_i. Towards a contradiction, suppose there is such an operation $\tilde{o}_w$ of C_w. Suppose $\tilde{o}_w$ has timestamp $\tilde{t}$ and note that $V'_w[w] < \tilde{t}$ follows from Lemma 9.

We distinguish two further cases. First, suppose $\tilde{o}_w \in \alpha_i$. Since o'_w precedes $\tilde{o}_w$ and since $o'_w \in \alpha_i$, it follows from Lemma 9 that $V_r[w] = V'_w[w] < \tilde{t}$. This contradicts the assumption that $\tilde{o}_w$ appears before o_r in π_i because the operations in π_i restricted to α_i are ordered by the versions they commit.

Second, suppose $\tilde{o}_w \in \beta_i$. By construction $\tilde{o}_w$ appears in π_i immediately before some read operation $\tilde{o}_r \in \alpha_i$ that commits $(\tilde{V}_r, \tilde{M}_r)$. Note that $\tilde{o}_r$ precedes o_r and that $\tilde{t} = \tilde{V}_r[w]$ according to the verification in *checkData*. Hence, $V_r[w] = V'_w[w] < \tilde{t} = \tilde{V}_r$, and this contradicts the assumption that $\tilde{o}_r$ appears before o_r in π_i because the operations in π_i restricted to α_i are ordered according to the versions they commit. □

Claim 18.5. *π_i preserves the weak real-time order of σ. Moreover, let π_i^- be the sequence of operations obtained from π_i by removing all operations of β_i that complete in σ; then π_i^- preserves the real-time order of σ.*

Proof. We first show that ρ_i preserves the real-time order of σ. Let o_j and o_k be two operations in ρ_i that commit versions (V_j, M_j) and (V_k, M_k), respectively, such that o_j executed by C_j precedes o_k executed by C_k in σ. Since o_k is invoked only after o_j completes, C_j does not find in L any operation by C_k with a valid SUBMIT-signature on a timestamp equal to or greater than $V_k[k]$. Hence $V_j[k] < V_k[k]$, and, thus, $(V_j, M_j) \dot{<} (V_k, M_k)$. Since o_j and o_k are ordered in ρ_i according to their versions by construction, we conclude that o_j appears before o_k also in ρ_i. The extension to the weak real-time order and the operations in π_i follows immediately from Claim 18.2.

For the second part, note that we have already shown that every pair of operations from $\pi_i^- \cap \alpha_i$ preserves the real-time order of σ. Moreover, the claim also holds vacuously for every pair of operations from $\pi_i^- \setminus \alpha_i$ because neither operation completes before the other one. It remains to show that every two operations $o_j \in \pi_i^- \setminus \alpha_i \subseteq \beta_i$ and $o_k \in \alpha_i$ preserve the real-time order of σ. Suppose o_j is the operation of C_j with timestamp t. Since o_j does not complete, not preserving real-time order means that $o_k <_\sigma o_j$ and $o_j <_{\pi_i} o_k$. Suppose for the purpose of a contradiction that this is the case. Since $o_j \in \beta_i$, it appears in π_i immediately before some read operation $o_r \in \alpha_i$ that commits a version (V_r, M_r). From the check in line 152 in Algorithm 1 we know that $V_r[j] \geq t$. Since o_j has not been invoked by the time when o_k completes, o_k must be different from o_r and it follows $o_r <_{\rho_i} o_k$ by assumption. Hence, the version (V_k, M_k) committed by o_k is larger than (V_r, M_r), and this implies $V_k[j] \geq t$. But this contradicts the fact that o_j has not yet been invoked when o_k completes, because according to the protocol logic, when an operation commits a version (V_l, M_l) with $V_l[j] > 0$, then the operation of C_j with timestamp $V_l[j]$ must have been invoked before. □

Claim 18.6. *For every operation $o \in \pi_i$ and every write operation $o' \in \sigma$, if $o' \rightarrow_\sigma o$ then $o' \in \pi_i$ and $o' <_{\pi_i} o$.*

Proof. Recalling the definition of causal precedence, there are three ways in which $o' \rightarrow_\sigma o$ might arise:

1. Suppose o and o' are operations executed by the same client C_j and $o' <_\sigma o$. Since $o \in \pi_i$, Claim 18.1 shows that $o' \in \pi_i$ and $o' <_{\pi_i} o$.
2. If o is a read operation that returns x and o' is the operation that writes x, then the fact that π_i is a view of σ at C_i, as established by Claim 18.4, implies that $o' \in \pi_i$ and precedes o in π_i.
3. If there is another operation o'' such that $o' \rightarrow_\sigma o''$ and $o'' \rightarrow_\sigma o$, then, using induction, o'' is contained in π_i and precedes o, and o' is contained in π_i and precedes o'', and, hence, o' precedes o in π_i. □

Claim 18.7. *For every client C_j, consider an operation o_k of client C_k, such that either $o_k \in \alpha_i \cap \alpha_j$ or for which there exists an operation o'_k of C_k such that o_k precedes o'_k. Then $\pi_i|^{o_k} = \pi_j|^{o_k}$.*

Proof. In the first case that $o_k \in \alpha_i \cap \alpha_j$, then by construction of ρ_i and ρ_j, and by the transitive order on versions, $\rho_i|^{o_k}$ and $\rho_j|^{o_k}$ contain exactly those operations that commit a version smaller than the version committed by o_k. Hence, $\rho_i|^{o_k} = \rho_j|^{o_k}$. Any operation $o_w \in \beta_i$ that appears in $\pi_i|^{o_k}$ is present in β_i only because of some read operation $o_r \in \rho_i|^{o_k}$. Since o_r also appears in $\rho_j|^{o}$ as shown above, o_w is also included in β_j and appears in π_j immediately before o_r and at the same position as in π_i. Hence, $\pi_i|^{o_k} = \pi_j|^{o_k}$.

In the second case, the existence of o'_k implies that o_k is not the last operation of C_k in π_i and, hence, $o_k \in \alpha_i$ and $o_k \in \alpha_j$. The statement then follows from the first case. □

Claims 18.4–18.7 establish that the protocol is weak fork-linearizable w.r.t. n SWMR registers. □

B Analysis of the Fail-Aware Untrusted Storage Protocol

We prove Theorem 2, i.e., that protocol FAUST in Algorithm 3 satisfies Definition 6. The functionality F is n SWMR registers; this is omitted when clear from the context.

The FAUST protocol relies on protocol USTOR for untrusted storage. We refer to the operations of these two protocols as *fail-aware-level operations* and *storage-level operations*, respectively. In the analysis, we have to rely on certain properties of the low-level untrusted storage protocol, which are formulated in terms of the storage operations *read* and *write*. But we face the complication that here, the high-level FAUST protocol provides *read* and *write* operations, and these, in turn, access the *extended* read and write operations of protocol USTOR, denoted by *writex* and *readx*.

In this section, we denote storage-level operations by $o_i, o_j, \ldots$ as before. It is clear from inspection of Algorithm 1 that all of its properties for read and write operations also hold for its extended read and write operations with minimal syntactic changes. We denote all fail-aware-level operations in this section by $\tilde{o}_i, \tilde{o}_j, \ldots$, in order to distinguish them from the operations at the storage level.

The FAUST protocol invokes exactly one storage-level operation for every one of its operations and also invokes dummy read operations. Therefore, the fail-aware-level operations executed by FAUST correspond directly to a subset of the storage-level operations executed by USTOR.

We say we *sieve* a sequence of storage-level events σ to obtain a sequence of fail-aware-level events $\tilde{\sigma}$ by removing all storage-level events that are part of dummy read operations and by mapping every one of the remaining storage-level events to its corresponding fail-aware-level event.

Note that read operations can be removed from a sequence of events without affecting whether the sequence satisfies the sequential specification of read/write registers. More precisely, when we remove the events of a set of read operations $\mathcal{Q}$ from a sequence of events π that satisfies the sequential specification, the resulting sequence $\tilde{\pi}$ also satisfies the sequential specification, as is easy to verify. This implies that if π is a view of a history σ, then $\tilde{\pi}$ is a view of $\tilde{\sigma}$, where $\tilde{\sigma}$ is obtained from σ by removing the events of all operations in $\mathcal{Q}$. Analogously, if σ is linearizable or causally consistent, then $\tilde{\sigma}$ is linearizable or causally consistent, respectively. We rely on this property in the analysis.

Analogously, removing all events of a set of read operations from a sequence π and from a history σ does not affect whether π is a view of σ. Hence, sieving does not affect whether a history linearizable and whether some sequence is a view of a history. Furthermore, according to the algorithm, an invocation (in $\tilde{\sigma}$) of a fail-aware-level operation triggers immediately an invocation (in σ) at the storage level, and, analogously, a response at the fail-aware level (in $\tilde{\sigma}$) occurs immediately after a corresponding response (in σ) at the storage level. Thus, sieving preserves also whether a history wait-free. We refer to these three properties as the *invariant of sieving* below.

Lemma 19 (Integrity). *When an operation $\tilde{o}_i$ of C_i returns a timestamp t, then t is bigger than any timestamp returned by an operation of C_i that precedes $\tilde{o}_i$.*

Proof. Note that $t = V_i[i]$, where (V_i, M_i) is the version committed by the corresponding storage-level operation (lines 316 and 325). By Lemma 9, $V_i[i]$ is larger than the timestamp of any preceding operation of C_i. □

Lemma 20 (Failure-detection accuracy). *If Algorithm 3 outputs $fail_i$, then S is faulty.*

Proof. According to the protocol, client C_i outputs *$fail_i$* only if one of three conditions are met: (1) the untrusted storage protocol outputs *$USTOR.fail_i$*; (2) in *update*, the version (V, M) received from a client C_j during a read operation or in a VERSION message is incomparable to *$VER_i[max_i]$*; or (3) C_i receives a FAILURE message from another client.

For the first condition, Theorem 1 guarantees that Algorithm 1 does not output *$USTOR.fail_i$* when S is correct. The second condition does not occur since the view history of every operation is a prefix of the schedule produced by the correct server, and all versions are therefore comparable, according to Lemma 15 in the analysis of the untrusted storage protocol. And the third condition cannot be met unless at least one client sends a FAILURE message after detecting condition (1) or (2). Since no client deviates from the protocol, this does not occur. □

The next lemma establishes requirements 1–3 of Definition 6. The causal consistency property follows because weak fork-linearizability implies causal consistency.

Lemma 21 (Linearizability and wait-freedom with correct server, causality). *Let $\tilde{\sigma}$ be a fair execution of Algorithm 3 such that $\tilde{\sigma}|_F$ is well-formed. If S is correct, then $\tilde{\sigma}|_F$ is linearizable w.r.t. F and wait-free. Moreover, $\tilde{\sigma}|_F$ is weak fork-linearizable w.r.t. F.*

Proof. As shown in the preceding lemma, a correct the server does not cause any client to output *fail*. Since S is correct, the corresponding execution σ of the untrusted storage protocol is linearizable and wait-free by Theorem 1. According to the invariant of sieving, also $\tilde{\sigma}|_F$ is linearizable and wait-free.

In case S is faulty, the execution σ at the storage level is weak fork-linearizable w.r.t. F according to Theorem 18. Note that in case a client detects incomparable versions, its last operation in σ does not complete in $\tilde{\sigma}|_F$. But omitting a response from σ does not change the fact that it is weak fork-linearizable because it can be added again by Definition 8. The invariant of sieving then implies that $\tilde{\sigma}|_F$ is also weak fork-linearizable w.r.t. F. □

Lemma 22. *Let $\tilde{o}_j$ be a complete fail-aware-level operation of C_j and suppose the corresponding storage-level operation o_j commits version (V_j, M_j). Then the value of $VER_i[j]$ at C_i at any time of the execution is comparable to (V_j, M_j).*

Proof. Let $(V^*, M^*) = VER_i[j]$ at any time of the execution. If C_i has assigned this value to $VER_i[j]$ during a read operation from X_j, then an operation of C_j committed (V^*, M^*) and the claim is immediate from Lemma 9. Otherwise, C_i has assigned (V^*, M^*) to $VER_i[j]$ after receiving a VERSION message containing (V^*, M^*) from C_j.

Notice that when C_j sends this message, it includes its maximal version at that time, in other words, $(V^*, M^*) = VER_j[max_j]$. Consider the point in the execution when $VER_j[max_j] = (V^*, M^*)$ for the first time. If o_j completes before this point in time, then $(V_j, M_j) \dot{\leq} VER_j[max_j] = (V^*, M^*)$ by the maintenance of the maximal version (line 342) and by the transitivity of versions. On the other hand, consider the case that o_j completes after this point in time. Since $\tilde{o}_j$ completes in $\tilde{\sigma}|_F$, the check on line 336 has been successful, and thus $(V_j, M_j) \dot{\leq} (V^\circ, M^\circ)$, where (V°, M°) is the value of $VER_j[max_j]$ at the time when $\tilde{o}_j$ completes. Because (V°, M°) is also greater than or equal to (V^*, M^*) by the maintenance of the maximal version (line 342), Lemma 16 (no-join) implies that (V_j, M_j) and (V^*, M^*) are comparable. □

Lemma 23. *Suppose a fail-aware-level operation $\tilde{o}_i$ of C_i is stable w.r.t. C_j and suppose the corresponding storage-level operation o_i commits version (V_i, M_i). Let $\tilde{o}_j$ be any complete fail-aware-level operation of C_j and suppose the corresponding storage-level operation o_j commits version (V_j, M_j). Then (V_i, M_i) and (V_j, M_j) are comparable.*

Proof. Let $(V^*, M^*) = VER_i[j]$ at the time when $\tilde{o}_i$ becomes stable w.r.t. C_j, and denote the operation that commits (V^*, M^*) by o^*.

It is obvious from the transitivity of versions and from the maintenance of the maximal version (line 342) that $(V_i, M_i) \dot{\leq} VER_i[max_i]$. For the same reasons, we have $(V^*, M^*) \dot{\leq} VER_i[max_i]$. Hence, Lemma 16 (no-join) shows that (V_i, M_i) and (V^*, M^*) are comparable.

We now show that $(V_i, M_i) \dot{\leq} (V^*, M^*)$. Note that when $stable_i(W_i)$ occurs at C_i, then $W_i[j] \geq V_i[i]$. According to lines 343–345 in Algorithm 3, we have that $V^*[i] = W_i[j] \geq V_i[i]$. Then Lemma 12 implies that o_i appears in $\mathcal{VH}(o^*)$. By Lemma 15, since (V_i, M_i) is comparable to (V^*, M^*), either $\mathcal{H}^v(o_i)$ is a prefix of $\mathcal{H}^v(o^*)$ or $\mathcal{H}^v(o^*)$ is a prefix of $\mathcal{H}^v(o_i)$. But since $o_i \in \mathcal{VH}(o^*)$, it must be that $\mathcal{H}^v(o_i)$ is a prefix of $\mathcal{H}^v(o^*)$. From Lemma 15, it follows that $(V_i, M_i) \dot{\leq} (V^*, M^*)$.

Considering the relation of (V^*, M^*) to (V_j, M_j), it must be that either $(V_j, M_j) \dot{\leq} (V^*, M^*)$ or $(V^*, M^*) \dot{\leq} (V_j, M_j)$ according to Lemma 22. In the first case, the lemma follows from Lemma 16 (no-join), and in the second case, the lemma follows by the transitivity of versions. □

Lemma 24 (Stability-detection accuracy). *If $\tilde{o}_i$ is a fail-aware-level operation of C_i that is stable w.r.t. some set of clients $\mathcal{C}$, then there exists a sequence of events $\tilde{\pi}$ that includes $\tilde{o}_i$ and a prefix $\tilde{\tau}$ of $\tilde{\sigma}|_F$ such that $\tilde{\pi}$ is a view of $\tilde{\tau}$ at all clients in $\mathcal{C}$ w.r.t. F. If $\mathcal{C}$ includes all clients, then $\tilde{\tau}$ is linearizable w.r.t. F.*

Proof. Let o_i be the storage-level operation corresponding to $\tilde{o}_i$, and let (V_i, M_i) be the version committed by o_i. Let σ be any history of the execution of protocol USTOR induced by $\tilde{\sigma}$. Let α_i, β_i, ρ_i, and π_i be sets and sequences of events, respectively, defined from σ according to the text before Theorem 18. We sieve $\pi_i|^{o_i}$ to obtain a sequence of fail-aware-level operations $\tilde{\pi}$ and let $\tilde{\tau}$ be the shortest prefix of $\tilde{\sigma}|_F$ that includes the invocations of all operations in $\tilde{\pi}$.

We next show that $\tilde{\pi}$ is a view of $\tilde{\tau}$ at C_j w.r.t. F for any $C_j \in \mathcal{C}$. According to the definition of a view, we create a sequence of events $\tilde{\tau}'$ from $\tilde{\tau}$ by adding a response for every operation in $\tilde{\pi}$ that is *incomplete* in $\tilde{\sigma}|_F$; we add these responses to the end of $\tilde{\tau}$ (there is at most one incomplete operation for each client).

In order to prove that $\tilde{\pi}$ is a view of $\tilde{\tau}$ at C_j w.r.t. F, we show (1) that $\tilde{\pi}$ is a sequential permutation of a subsequence of *complete*$(\tilde{\tau}')$; (2) that $\tilde{\pi}|_{C_j} =$ *complete*$(\tilde{\tau}')|_{C_j}$; and (3) that $\tilde{\pi}$ satisfies the sequential specification of F. Property (1) follows from the fact that $\tilde{\pi}$ is sequential and includes only operations that are invoked in $\tilde{\tau}$ and by construction of *complete*$(\tilde{\tau}')$ from $\tilde{\tau}$. Property (3) holds because π_i is a view of σ at C_i w.r.t. F according to Claim 18.4, and because the sieving process that constructs $\tilde{\pi}$ from $\pi|^{o_i}$ preserves the sequential specification of F.

Finally, we explain why property (2) holds. We start by showing that the set of operations in $\tilde{\pi}|_{C_j}$ and *complete*$(\tilde{\tau}')|_{C_j}$ is the same. For any operation $\tilde{o}_j \in \tilde{\pi}|_{C_j}$, property (1) already establishes that $\tilde{o}_j \in$ *complete*$(\tilde{\tau}')$. It remains to show that any $\tilde{o}_j \in$ *complete*$(\tilde{\tau}')$ also satisfies $\tilde{o}_j \in \tilde{\pi}|_{C_j}$.

The assumption that $\tilde{o}_j$ is in *complete*$(\tilde{\tau}')$ means that either $\tilde{o}_j \in \tilde{\pi}$ or that $\tilde{o}_j$ is complete already in $\tilde{\tau}$. In the former case, the implication holds trivially. In the latter case, because the corresponding storage-level operation $o_j \in \pi_i|^{o_i}$ is complete and commits (V_j, M_j), Lemma 23 implies that (V_j, M_j) and (V_i, M_i) are comparable. If $(V_j, M_j) \dot{\leq} (V_i, M_i)$, then $o_j \in \pi_i|^{o_i}$ by construction of π_i, and furthermore, $\tilde{o}_j \in \tilde{\pi}|_{C_j}$ by construction of π_i. Otherwise, it may be that $(V_i, M_i) \dot{<} (V_j, M_j)$, but we show next that this is not possible.

If $(V_i, M_i) \dot{<} (V_j, M_j)$, then by definition of $\tilde{\tau}$, the invocation of some operation $\tilde{o}_k \in \tilde{\pi}$ appears in $\tilde{\sigma}|_F$ after the response of $\tilde{o}_j$. By construction of $\tilde{\pi}$, the corresponding storage-level operation o_k is contained in $\pi_i|^{o_i}$. According to the protocol, operations and upon clauses are executed atomically, and therefore the invocation of o_k appears in σ after the response of o_j. At the same time, Claim 18.3 implies that o_k is invoked before o_j completes, a contradiction.

To complete the proof of property (2), it is left to show that the order of the operations in $\tilde{\pi}|_{C_j}$ and in *complete*$(\tilde{\tau}')|_{C_j}$ is the same. By Claim 18.1, π_i preserves the real-time order of σ among the operations of C_j. Therefore, $\tilde{\pi}$ also preserves the real-time order of $\tilde{\sigma}|_F$ among the operations of C_j. On the other hand, since $\tilde{\tau}$ is a prefix of $\tilde{\sigma}|_F$ and since $\tilde{\tau}'$ is created from $\tilde{\tau}$ by adding responses at the end, it easy to see that the operations of C_j in $\tilde{\tau}'$ are in the same order as in $\tilde{\sigma}|_F$.

For the last part of the lemma, it suffices to show that when $\mathcal{C}$ includes all clients, and, hence, $\tilde{\pi}$ is a view of $\tilde{\tau}$ at all clients, then $\tilde{\pi}$ preserves the real-time order of $\tilde{\tau}$. By Lemma 23, every complete operation in $\tilde{\sigma}|_F$ corresponds to a complete storage-level operation that commits a version comparable to (V_i, M_i). Therefore, all operations of $\pi_i|^{o_i}$ that correspond to a complete fail-aware-level operation are in $\pi_i|^{o_i} \cap \alpha_i$. There may be incomplete fail-aware-level operations as well, and the above argument shows that the corresponding storage-level operations are contained in $\pi_i|^{o_i} \cap \beta_i$. We create a sequence of events σ' from $\sigma|^{o_i}$ by removing the responses of all operations in $\pi_i|^{o_i} \cap \beta_i$. Claim 18.5 implies that $\pi_i|^{o_i}$ preserves the real-time order of σ'. Notice that sieving σ' also yields $\tilde{\sigma}|_F$. Therefore, $\tilde{\pi}$ preserves the real-time order of $\tilde{\sigma}|_F$ and since $\tilde{\tau}$ is a prefix of $\tilde{\sigma}|_F$, we conclude that $\tilde{\pi}$ also preserves the real-time order of $\tilde{\tau}$. □

Lemma 25 (Detection completeness). *For every two correct clients C_i and C_j and for every timestamp t returned by some operation $\tilde{o}_i$ of C_i, eventually either fail occurs at all correct clients or stable$_i(W)$ occurs at C_i with $W[j] \geq t$.*

Proof. Notice that whenever *fail* occurs at a correct client, the client also sends a FAILURE message to all other clients. Since the offline communication method is reliable, all correct clients eventually receive this message, output *fail*, and halt. Thus, for the remainder of this proof we assume that C_i and C_j do not output *fail* and do not halt. We show that $stable_i(W)$ occurs eventually at C_i such that $W[j] \geq t$. Let o_i be the storage-level operation corresponding to $\tilde{o}_i$. Note that o_i completes and suppose it commits version (V_i, M_i). Thus, $V_i[i] = t$.

We establish the lemma in two steps: First, we show that $VER_j[max_j]$ eventually contains a version that is greater than or equal to (V_i, M_i). Second, we show that also $VER_i[j]$ eventually contains a version that is greater than or equal to (V_i, M_i).

For the first step, note that every VERSION message that C_i sends to C_j after completing $\tilde{o}_i$ contains a version that is greater than or equal to (V_i, M_i), by the maintenance of the maximal version (line 342) and by the transitivity of versions. Since the offline communication method is reliable and both C_i and C_j are correct, C_j eventually receives this message and updates $VER_j[max_j]$ to this version that is greater than or equal to (V_i, M_i).

Suppose that C_i does not send any VERSION message to C_j after completing $\tilde{o}_i$. This means that C_i never receives a PROBE message from C_j and hence, $C_i \notin D$ at C_j. This is only possible if C_j updates $T_j[i]$ periodically, at the latest every δ time units, when receiving a version from C_i during a read operation from X_i. Therefore, one of these read operations eventually returns a version (V_i', M_i') committed by an operation o_i' of C_i, where $o_i' = o_i$ or o_i precedes o_i'. Thus, $(V_i, M_i) \dot{\leq} (V_i', M_i')$ and by the maintenance of the maximal version at C_j (line 342) and by the transitivity of versions, we conclude that $(V_i, M_i) \dot{\leq} VER_j[max_j]$ when the read operation completes. This concludes the first step of the proof.

We now address the the second step. Note when C_j sends to C_i a VERSION message at a time when $(V_i, M_i) \dot{\leq} VER_j[max_j]$ holds, the message includes a version that is also greater than or equal to (V_i, M_i). When C_j receives this message, it stores this version in $VER_i[j]$.

Suppose that after the first time when $(V_i, M_i) \dot{\leq} VER_j[max_j]$ holds, C_j does not send any VERSION message to C_i. Using the same argument as above with the roles of C_i and C_j reversed, we conclude that C_i periodically executes a read operation from X_j and stores the received versions in $VER_i[j]$. Eventually some read operation o_i' commits a version (V_i', M_i') and returns a version (V_j, M_j) committed by an operation of C_j that was invoked after o_i completed. Lemma 10 shows that $(V_j, M_j) \dot{\leq} (V_i', M_i')$, and since o_i and o_i' are both operations of C_i and o_i precedes o_i', it follows $(V_i, M_i) \dot{\leq} (V_i', M_i')$ from Lemma 9. Then Lemma 16 (no-join) implies that (V_i, M_i) is comparable to (V_j, M_j), and it must be that $(V_i, M_i) \dot{\leq} (V_j, M_j)$ since o_i precedes o_j. Thus, after completing o_i', we observe that $VER_i[j]$ is greater than or equal to (V_i, M_i).

To conclude the argument, note that when $VER_i[j]$ contains a version greater than or equal to (V_i, M_i) for the first time, then $wchange_i = $ TRUE and this triggers a $stable_i(W)$ notification with $W[j] \geq t$. □