Fail-Aware Untrusted Storage [*]

Christian Cachin
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
cca@zurich.ibm.com

Idit Keidar
Alexander Shraer
Department of Electrical Engineering, Technion
Haifa 32000, Israel
{idish@ee, shralex@tx}.technion.ac.il

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 Google Docs [9], and cloud computing. 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.

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 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 [23] 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, 24]. 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 [8] additionally introduces a notification to the clients in case the service cannot provide its specified

[*] This work is partially supported by the Israel Science Foundation and the European Commission through the IST Programme under Contract IST-4-026764-NOE ReSIST.

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, provides accurate and complete consistency and failure notifications; it requires the service to be linearizable and wait-free when the provider is correct, and to be causally consistent when the provider is faulty.

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, 5]. 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, such 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, 5] 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 [5].

A weaker notion called *fork-*-linearizability* has been proposed recently [17]. But as we show in the full paper [3], the notion (when adapted to our model) cannot provide wait-free client operations either, and in addition also permits a faulty server to violate causal consistency. 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, eventually issues 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 uses offline message exchange among clients.

In summary, the contributions of this work 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 causality;
2. The insight that no existing forking consistency notion can be used for fail-aware untrusted storage, because they inherently rule out wait-free implementations; and
3. An efficient wait-free protocol for implementing fail-aware untrusted storage, relying on the novel notion of weak fork-linearizability.

Although this paper focuses on storage, which is but one example of a fail-aware untrusted service, we believe that the notion is useful for tolerating Byzantine faults in a variety of additional services.

Related Work. In order to provide wait-freedom when linearizability cannot be guaranteed, numerous real-world systems guarantee eventual consistency, for example, Coda [14], Bayou [23], Tempest [19], and Dynamo [7]. 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.

The notion of fail-awareness [8] is exploited by many systems in the timed asynchronous model, where nodes are subject to crash failures [6]. 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 [5] 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 [5, 4]. 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.

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 [22]; 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, whereas FAUST never compromises linearizability when the server is correct.

2 Model and Definitions

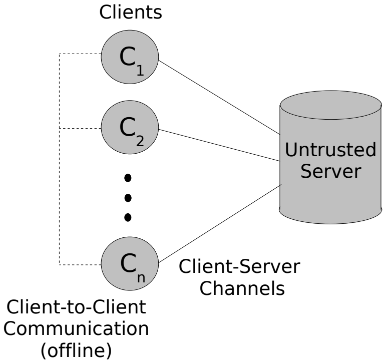

Figure 1. System architecture.

System model. We consider an asynchronous distributed system consisting of n clients $C_1, \ldots, C_n$, a server S, and asynchronous reliable FIFO channels between the clients and S. In addition, there is a reliable offline communication method between clients, which eventually delivers messages, even if the clients are not simultaneously connected; see Figure 1. 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*.

The protocol emulates a *shared functionality* F, i.e., a shared object, to the clients. F is defined via a *sequential specification*, which indicates its behavior in sequential executions. The functionality considered in this paper is n SWMR registers $X_1, \ldots, X_n$, storing values from a domain $\mathcal{X}$. Initially, each register holds a special value $\perp \notin \mathcal{X}$. A client C_i writes only to register X_i by invoking $\mathit{write}_i(X_i, x)$ with some value x, which returns OK. The client can read from any register X_j, which responds with a value x, denoted $\mathit{read}_i(X) \to x$. 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. For simplicity, we assume that the values written to all registers are unique.

We use a collision-resistant *hash function* H and *digital signatures* from cryptography: all parties know and can use H, only client C_i can create a signature by invoking sign_i, and any party can verify that signature supposedly issued by C_i by calling verify_i. For formal definitions, see [3].

Operations and Histories. Clients interact with the functionality 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, $\mathit{complete}(\sigma)$ 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 $\sigma|_{C_i}$. For some operation o, the prefix of σ that ends with the last event of o is denoted by $\sigma|^o$.

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]).

Traditional Consistency and Liveness Properties. Our definitions rely on the notion of a possible *view* of a client.

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' s.t.:

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 exist a sequence of events π s.t.:

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* weakens linearizability and allows that clients observe a different order of those operations that do not conflict with each other. It is based on the notion of *potential causality* [15]. We formalize it for a general F by adopting the *reads-from* relation and the distinction between *query* and *update* operations from database theory [2]. Identifying operations of F with transactions, an operation o' reads-from an operation o when o writes a value v to some low-level data item x, and o' reads v from x.

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. o' reads-from o; 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. Our definition of causal consistency reduces to the intuitive notion of causal consistency for shared memory by Hutto and Ahamad [13], when instantiated for read/write registers.

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 s.t.:

1. π_i is a view of σ at C_i w.r.t. F;
2. π_i contains all update operations $o \in \sigma$ that causally precede some operation $o' \in \pi_i$; 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 common 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.

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.

Second, we add two new output actions at client C_i, called *stable*$_i$ and *fail*$_i$, which occur asynchronously. (From now on, we always add the subscript i to actions at client C_i.) The action *stable*$_i$ includes a timestamp vector W as a parameter and informs C_i about the stability of its operations with respect to the other clients. When *stable*$_i(W)$ occurs, then we say that all operations of C_i that returned a timestamp less than or equal to $W[j]$ are *stable w.r.t.* C_j, for $j = 1, \ldots, n$. 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.

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, denoted by $\sigma|_F$.

Definition 5 (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 s.t. $\sigma|_F$ 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 $\mathit{stable}_i(W)$ occurs at C_i with $W[j] \geq t$.

We now illustrate how the fail-aware service can be used by clients who collaborate on editing a file from across the world. Suppose that the server S is correct and three correct clients are using it to collaborate: 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 day time in Europe, and Alice and Bob use the service and see the effects of each other's updates. However, they do not observe any operations of Carlos because he is asleep.

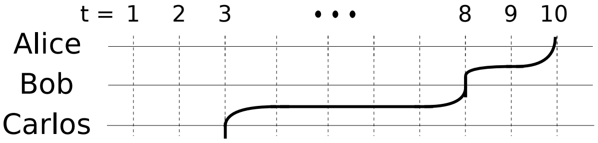

Figure 2. The stability cut of Alice indicated by the notification $\mathit{stable}_{\mathit{Alice}}([10, 8, 3])$. The values of t are the timestamps returned by the operations of Alice.

Suppose Alice completes an operation that returns timestamp 10, and receives a notification $\mathit{stable}_{\mathit{Alice}}([10, 8, 3])$, indicating that she is consistent with Bob up to her operation with timestamp 8, consistent with Carlos up to her operation with 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.

4 Weak Fork-Linearizability

Intuitively, fork-linearizability [20] allows clients' views to diverge, but once there is any inconsistency in the views of two clients, these clients can never again see common operations. This is called the *no-join* property of fork-linearizability.

Fork*-linearizability [17] weakens this notion to an *at-most-one-join* property, where once the view of one client becomes inconsistent with that of a second client, the former can see at most one additional operation of the latter, and vice versa. Thus, the views of two clients that contain common operations are identical up to the penultimate common operation by each client. 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.

We introduce a new consistency notion, called *weak fork-linearizability*, which permits wait-free protocols and is therefore suitable for implementing our notion of a fail-aware untrusted service. 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 $\mathit{lastops}(\pi)$ be a function of π returning the set containing the last operation from every client in π; in other words, $\mathit{lastops}(\pi) = \bigcup_{i\in\{1,\ldots,n\}}\{o \in \pi : o \text{ is the last operation in } \pi|_{C_i}\}$. 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 $\mathit{lastops}(\pi)$ preserves the real-time order of σ. In addition to weakening the real-time-order condition, we also require causal consistency.

Definition 6 (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 s.t.:

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 update operation $o' \in \sigma$ s.t. $o' \rightarrow_\sigma o$, 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, the second condition only requires preservation of the real-time order of the execution for each view *excluding* the last operation of every client that appears in it. The third condition requires causal consistency, which is implicit in fork-linearizability. The fourth condition allows again an inconsistency for the last operation of every client in a view. Weak fork-linearizability is neither stronger nor weaker than fork-*-linearizability, as illustrated in the full paper [3].

Consider the following history, shown in Figure 3: Initially, X_1 contains $\bot$. Client C_1 executes $\mathit{write}_1(X_1, u)$, then client C_2 executes $\mathit{read}_2(X_1) \rightarrow \bot$ and $\mathit{read}_2(X_1) \rightarrow u$. During the execution of the first read operation of C_2, the server pretends that the write operation of C_1 did not occur.

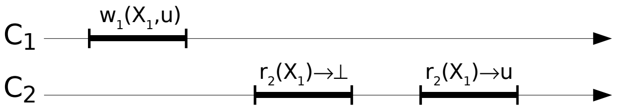

Figure 3. A weak fork-linearizable history that is not fork-linearizable.

This example is weak fork-linearizable. The sequences:

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

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 (a formal proof appears in a companion paper [4]).

5 A Weak Fork-Linearizable Protocol

We present a weak fork-linearizable untrusted storage protocol (USTOR) that implements n SWMR registers $X_1, \ldots, X_n$ with an untrusted server, where client C_i writes to register X_i. When the server is correct, our protocol guarantees linearizability and wait-freedom in all fair and well-formed executions. The protocol for clients is presented in Algorithm 1, and the protocol for the server S appears in Algorithm 2. Proofs and complexity analysis are deferred to the full paper [3].

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 client then 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 server processes arriving SUBMIT messages in FIFO order, and the execution of each event handler is atomic. 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. The bulk of the protocol logic is devoted to dealing with a faulty server.

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 12 and 25). Client C_i also stores a hash $\bar{x}_i$ of the value most recently written to X_i (line 6).

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 $(i, \mathit{oc}, j, \sigma)$, 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.

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 in the COMMIT message.

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 102), and stores the timestamp vector and the signature of the last COMMIT message received from every client in an array *SVER* (line 104).

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 sends explicit information 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 105). Furthermore, S stores the index of the client that executed o^* in c (lines 103 and 120). The REPLY message from S to C_i contains c, L, and the timestamp vector V^c from the COMMIT message of o^*.

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

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

where $\parallel$ denotes concatenation. 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 16 and 28), 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 37), incrementing one entry in the vector for every operation represented in L (line 42), and finally incrementing its own entry (line 46).

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* of $\omega^1, \ldots, \omega^m$ is defined 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 38, 45, and 47).

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 7 (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 $M_i[k] = M_j[k]$.

We say that (V_i, M_i) is *smaller* than (V_j, M_j), denoted $(V_i, M_i) \dot{<} (V_j, M_j)$, whenever $(V_i, M_i) \dot{\leq} (V_j, M_j)$ and $(V_i, M_i) \neq (V_j, M_j)$.

Suppose that an operation o_i of client C_i commits (V_i, M_i) and an operation o_j of C_j commits (V_j, M_j). The first condition orders the operations according to their timestamp vectors, while the second 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. We show [3] 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 $\mathcal{VH}(o_j)$. 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.

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 106) 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 their 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 15 and 27). 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 110 and 113). 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.

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 35). 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 36). These conditions always hold when S is correct, since the channels are reliable with FIFO order

Algorithm 1 USTOR, code for client C_i.

```
1:  notation
2:      $Strings = \{0,1\}^* \cup \{\bot\}$; $Clients = \{1, \ldots, n\}$
3:      $Opcodes = \{\text{READ}, \text{WRITE}, \bot\}$
4:      $Invocations = Clients \times Opcodes \times Clients \times Strings$
5:  state
6:      $\bar{x}_i \in Strings$, initially $\bot$
7:      $(V_i, M_i) \in \mathbb{N}_0^n \times Strings^n$, initially $(0^n, \bot^n)$
8:  operation $write_i(x)$                                // write $x$ to register $X_i$
9:      $(\cdots) \leftarrow writex_i(x)$
10:     return OK
11: operation $writex_i(x)$                    // extended write $x$ to register $X_i$
12:     $t \leftarrow V_i[i] + 1$                          // timestamp of the operation
13:     $\bar{x}_i \leftarrow H(x)$
14:     $\tau \leftarrow \text{sign}_i(\text{SUBMIT}\|\text{WRITE}\|i\|t)$; $\delta \leftarrow \text{sign}_i(\text{DATA}\|t\|\bar{x}_i)$
15:     send $\langle \text{SUBMIT}, t, (i, \text{WRITE}, i, \tau), x, \delta \rangle$ to $S$
16:     wait for receiving $\langle \text{REPLY}, c, (V^c, M^c, \varphi^c), L, P \rangle$ from $S$
17:     $updateVersion(i, (c, V^c, M^c, \varphi^c), L, P)$
18:     $\varphi \leftarrow \text{sign}_i(\text{COMMIT}\|V_i\|M_i)$; $\psi \leftarrow \text{sign}_i(\text{PROOF}\|M_i[i])$
19:     send $\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle$ to $S$
20:     return $(V_i, M_i)$
21: operation $read_i(j)$                                 // read from register $X_j$
22:     $(x^j, \cdots) \leftarrow readx_i(j)$
23:     return $x^j$
24: operation $readx_i(j)$                      // extended read from register $X_j$
25:     $t \leftarrow V_i[i] + 1$                          // timestamp of the operation
26:     $\tau \leftarrow \text{sign}_i(\text{SUBMIT}\|\text{READ}\|j\|t)$; $\delta \leftarrow \text{sign}_i(\text{DATA}\|t\|\bar{x}_i))$
27:     send $\langle \text{SUBMIT}, t, (i, \text{READ}, j, \tau), \bot, \delta \rangle$ to $S$
28:     wait for receiving $\langle \text{REPLY}, c, (V^c, M^c, \varphi^c), (V^j, M^j, \varphi^j),$
            $(t^j, x^j, \delta^j), L, P \rangle$ from $S$
29:     $updateVersion(j, (c, V^c, M^c, \varphi^c), L, P)$
30:     $checkData(c, (V^c, M^c, \varphi^c), j, (V^j, M^j, \varphi^j), (t^j, x^j, \delta^j))$
31:     $\varphi \leftarrow \text{sign}_i(\text{COMMIT}\|V_i\|M_i)$; $\psi \leftarrow \text{sign}_i(\text{PROOF}\|M_i[i])$
32:     send $\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle$ to $S$
33:     return $(x^j, V_i, M_i, V^j, M^j)$
34: procedure $updateVersion(j, (c, V^c, M^c, \varphi^c), L, P)$
35:     if not $\big((V^c, M^c) = (0^n, \bot^n)$ or
            $verify_c(\varphi^c, \text{COMMIT}\|V^c\|M^c)\big)$ then output $fail_i$; halt
36:     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
37:     $(V_i, M_i) \leftarrow (V^c, M^c)$
38:     $d \leftarrow M^c[c]$
39:     for $q = 1, \ldots, |L|$ do
40:         $(k, oc, l, \tau) \leftarrow L[q]$
41:         if not $\big(M_i[k] = \bot$ or $verify_k(P[k], \text{PROOF}\|M_i[k])\big)$ then
                output $fail_i$; halt
42:         $V_i[k] \leftarrow V_i[k] + 1$
43:         if $k = i$ or not $verify_k(\tau, \text{SUBMIT}\|oc\|l\|V_i[k])$ then
                output $fail_i$; halt
44:         $d \leftarrow H(d\|k)$
45:         $M_i[k] \leftarrow d$
46:     $V_i[i] = V_i[i] + 1$
47:     $M_i[i] \leftarrow H(d\|i)$
48: procedure $checkData(c, (V^c, M^c, \varphi^c), j, (V^j, M^j, \varphi^j),$
        $(t^j, x^j, \delta^j))$
49:     if not $\big((V^j, M^j) = (0^n, \bot^n)$ or
            $verify_j(\varphi^j, \text{COMMIT}\|V^j\|M^j)\big)$ then output $fail_i$; halt
50:     if not $\big(t^j = 0$ or $verify_j(\delta^j, \text{DATA}\|t^j\|H(x^j))\big)$ then
            output $fail_i$; halt
51:     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
52:     if not $\big(V^j[j] = t^j$ or $V^j[j] = t^j - 1\big)$ then output $fail_i$; halt
```

Algorithm 2 USTOR, code for server.

```
101: state
102:     $MEM[i] \in \mathbb{N}_0 \times \mathcal{X} \times Strings$, initially $(0, \bot, \bot)$, $i = 1, \ldots, n$
             // last timestamp, value, and DATA-sig. from $C_i$
103:     $c \in Clients$, initially 1
             // client who committed last operation in schedule
104:     $SVER[i] \in \mathbb{N}_0^n \times Strings^n \times Strings$,
             initially $(0^n, \bot^n, \bot)$, for $i = 1, \ldots, n$
             // last version and COMMIT-signature received from $C_i$
105:     $L \in Invocations^*$, initially empty
             // invocation tuples of concurrent operations
106:     $P \in Strings^n$, initially $\bot^n$                      // PROOF-signatures
107: upon receiving $\langle \text{SUBMIT}, t, (i, oc, j, \tau), x, \delta \rangle$ from $C_i$
108:     if $oc = \text{READ}$ then
109:         $(t', x', \delta') \leftarrow MEM[i]$
110:         $MEM[i] \leftarrow (t, x', \delta)$
111:         $msg \leftarrow \langle \text{REPLY}, c, SVER[c], SVER[j], MEM[j], L, P \rangle$
112:     else
113:         $MEM[i] \leftarrow (t, x, \delta)$
114:         $msg \leftarrow \langle \text{REPLY}, c, SVER[c], L, P \rangle$
115:     send $msg$ to $C_i$
116:     append $(i, oc, j, \tau)$ to $L$
117: upon receiving $\langle \text{COMMIT}, V_i, M_i, \varphi, \psi \rangle$ from $C_i$:
118:     $(V^c, M^c, \varphi^c) \leftarrow SVER[c]$
119:     if $V_i > V^c$ then
120:         $c \leftarrow i$
121:         remove last tuple $(i, \cdots)$ and all preceding tuples from $L$
122:     $SVER[i] \leftarrow (V_i, M_i, \varphi)$
123:     $P[i] \leftarrow \psi$
```

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 computes its new version. It starts from (V^c, M^c) and for every invocation tuple in L, representing a concurrent operation by C_k, it checks the following (lines 39–45): first, that S received the COMMIT message of C_k's previous operation and included the corresponding PROOF-signature in $P[k]$ (line 41); second, that $k \neq i$, i.e., that C_i has no concurrent operation with itself (line 43); 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 43). 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. Finally, C_i increments its own timestamp $V_i[i]$, computes the new digest, and assigns it to $M_i[i]$ (line 47). If any of the checks fail, the protocol outputs $fail_i$, which signals to a higher-layer protocol that the client has detected an inconsistency caused by S, 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). If S is correct, this is the largest version 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 50). Furthermore, it checks that the version (V^j, M^j) is smaller than or equal to (V^c, M^c) (line 51). 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 51). 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 52).

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 $M_i[i]$. When the server receives the COMMIT message from C_i containing a version (V, M), it stores the version and the PROOF-signature in *SVER*$[i]$ and the COMMIT-signature in $P[i]$ (lines 122 and 123). Last but not least, the server checks if this operation is now the last committed operation in the schedule by testing $V > V^c$; 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.

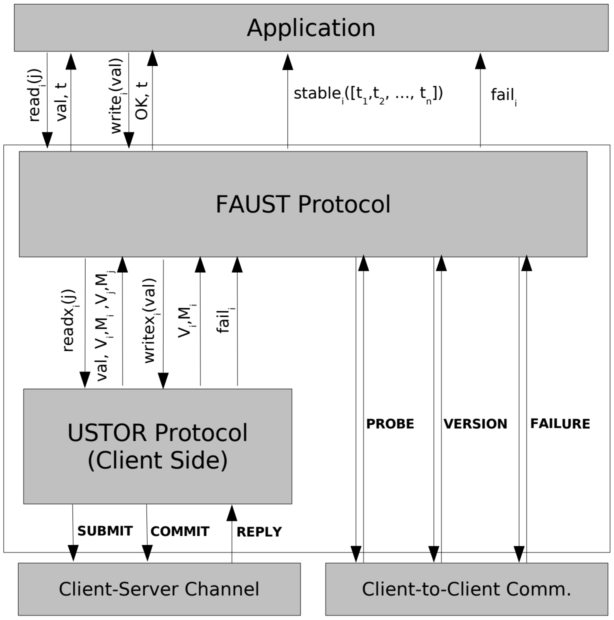

Figure 4. Architecture of the FAUST protocol.

6 Fail-Aware Untrusted Storage Protocol

In this section, we extend the USTOR protocol to a fail-aware untrusted storage protocol (FAUST) which satisfies Definition 5. As illustrated in Figure 4, FAUST is executed by clients and uses the extended read and write operations of USTOR as well as off-line client-to-client communication. Below we describe at a high level how FAUST achieves its goals, and refer to the full paper [3] for details.

To detect server failures, a client that ceases to obtain new versions from another client via the server 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. For *stability detection*, the protocol periodically invokes *dummy* read operations, and uses the versions received from other clients in VERSION messages. Although using client-to-client communication has been suggested before to detect server failures [20, 16], this mechanism of FAUST is the first in the context of untrusted storage to employ offline communication for detecting stability and for aiding progress when no inconsistency occurs.

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 the operation of the USTOR protocol and takes the timestamp from the i-th entry in the timestamp vector. More precisely, client C_i stores an array VER_i containing the maximal version that it has received from every other client. It sets $\mathit{VER}_i[i]$ to the version committed by the most recent operation of its own and updates the value of $\mathit{VER}_i[j]$ when a $\mathit{readx}_i(j)$ operation of the USTOR 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, when no operation invoked by the user is ongoing. However, dummy read operations alone do not guarantee stability-detection completeness according to Definition 5 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 $\mathit{VER}_i[j]$, the protocol stores 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 directly to C_j. Upon receiving a PROBE message, C_j replies with a VERSION message containing $\mathit{VER}_j[\mathit{max}_j]$, the maximal version that C_j knows. Client C_i also updates the value of $\mathit{VER}_i[j]$ 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 not necessarily contain a version committed by an operation of C_i.

The client detects server failures in one of three ways: First, the USTOR protocol may output $\mathit{USTOR.fail}_i$ if it detects any inconsistency in the messages from the server. Second, 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 checks (V, M) for consistency with the versions that it already knows, by verifying that (V, M) is comparable to $\mathit{VER}_i[\mathit{max}_i]$. 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 sends a FAILURE message to alert all other clients, outputs fail_i, and halts.

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., $W_i[j] = V_j[i]$ for every j, where $(V_j, M_j) = \mathit{VER}_i[j]$. 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.

7 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 in computing “clouds,” the importance of addressing this problem becomes more prominent. We introduced the abstraction of a fail-aware untrusted service, generalizing the concepts of eventual consistency and fail-awareness to account for Byzantine faults. We realize this new abstraction with an online storage service providing so-called forking semantics. It guarantees linearizability and wait-freedom when the server is correct, provides failure and stability detection, and ensures causality at all times. No previous forking consistency notion can be used for building fail-aware untrusted storage, because these notions inherently rule out wait-free implementations. Our new notion of weak fork-linearizability is key to circumventing this limitation. We developed an efficient wait-free storage protocol with weak fork-linearizability and used it to implement fail-aware untrusted storage.

Acknowledgments. We thank Alessia Milani, Dani Shaket, and Marko Vukolić for their valuable comments.

References

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

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. Technical Report CCIT 712, Department of Electrical Engineering, Technion, Dec. 2008.

[4] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *IPL*, 109(7), 2009.

[5] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. PODC*, pages 129–138, 2007.

[6] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE TPDS*, 10(6), 1999.

[7] 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. SOSP*, 2007.

[8] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proc. PODC*, 1996.

[9] Google Docs. `http://docs.google.com/`.

[10] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, 2007.

[11] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 11(1), 1991.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[13] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *Proc. ICDCS*, 1990.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM TOCS*, 10(1), 1992.

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

[16] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, 2004.

[17] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. 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. DSN DCCS*, 2008.

[20] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. PODC*, 2002.

[21] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. DISC*, 2006.

[22] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, , and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *Proc. NSDI*, 2009.

[23] 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 *SOSP*, 1995.

[24] J. Yang, H. Wang, N. Gu, Y. Liu, C. Wang, and Q. Zhang. Lock-free consistency control for Web 2.0 applications. In *Proc. WWW*, 2008.