Reliable Collaboration Using Unreliable Storage

ALEXANDER SHRAER

Reliable Collaboration Using Unreliable Storage

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ALEXANDER SHRAER

Submitted to the Senate of the Technion – Israel Institute of Technology
ELUL 5770 HAIFA September 2010

The Research Was Done Under the Supervision of Prof. Idit Keidar in the Department of Electrical Engineering, Technion.

THE GENEROUS FINANCIAL HELP OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY IS GRATEFULLY ACKNOWLEDGED

Acknowledgments

I would like to express my deepest gratitude to Idit Keidar, my advisor, for introducing me to the fascinating world of distributed computing and to research in general, sharing her vast knowledge and experience, and for her valuable guidance, patience, and support throughout my graduate studies. Idit is the best advisor any graduate student can hope for and I was lucky that I had the opportunity to learn from her in the beginning of my research career. I'm sure that the tools and the general approach to research that I have learned from Idit will help me wherever I go next.

During my studies I was fortunate to collaborate with many leading researchers. I'm grateful to Christian Cachin, who hosted me in IBM Research Zurich, and collaborated with me from the beginning of my PhD to its completion. I learned a lot from Christian about research in general and industrial research in particular, and I consider Christian to be my second advisor. Christian's deep understanding of both cryptography and distributed computing were truly an inspiration.

I'd like to thank Dahlia Malkhi for hosting me in Microsoft Research Silicon Valley and introducing me to reconfigurable distributed storage. I learned a lot from Dahlia's insight, experience and unique research approach. I would also like to thank Marcos Aguilera and Jean-Philippe Martin for co-hosting me in MSR and their collaboration on the first part of this thesis.

Special thanks go to Eliezer Dekel and Gregory Chockler from IBM Research Haifa for welcoming me to their group, and their collaboration in the initial stages of my PhD. I'd also like to thank Roie Melamed, Yoav Tock and Roman Vitenberg, with whom I worked in IBM.

I'd like to thank friends and colleagues from the Technion: Edward (Eddie) Bortnikov, Maxim Gurevich, Liran Katzir, Gabriel (Gabi) Kliot, Sivan Bercovici, Ittay Eyal, Dmitry Perlman, Dmitry Basin, and all other members of Idit's research group for their valuable comments and suggestions. I'd like to thank Asaf Cidon, Yan Michalevsky and Dani Shaket for their collaboration on the Venus paper.

Many thanks go to Ittai Abraham, Hagit Attiya, Tsahi Birk, Eli Gafni, Rachid Guerraoui, Roy Friedman, Amir Herzberg, Leslie Lamport, Alessia Milani and Marko Vukolić for many discussions of this work and a special thanks to Yoram Moses who served on all my MSc and PhD committees and provided valuable input. I thank Abhi Shelat, for introducing me to Untrusted Storage and for his enthusiastic approach to research.

I'd like to thank the Israeli Ministry of Science for its generous financial support through the Eshkol fellowship I was granted.

I'd like to dedicate this thesis to my mom, Zoya, without whom I wouldn't be where I am today. Last, but certainly not least, I'd like to thank my wonderful wife, Tanya, for her endless support and encouragement.

Contents

List of Figures

List of Tables

Abstract

This thesis concerns the reliability, security, and consistency of storage in distributed systems. Distributed storage architectures provide a cheap and scalable alternative to expensive monolithic disk array systems currently used in enterprise environments. Such distributed architectures make use of many unreliable servers (or storage devices) and provide reliability through replication. Another emerging alternative is cloud storage, offered remotely by multiple providers.

The first problem addressed in this thesis is the support of reconfiguration in distributed storage systems. The large number of fault-prone servers in such systems requires supporting dynamic changes when faulty servers are removed from the system and new ones are introduced. In order to maintain reliability when such changes occur, it is essential to ensure proper coordination, e.g., when multiple such changes occur simultaneously. Existing solutions are either centralized, or use strong synchronization algorithms (such as consensus) among the servers to agree on every change in the system. In fact, it was widely believed that reconfigurations require consensus and just like consensus cannot be achieved in asynchronous systems. In this work we refute this belief and present DynaStore, an asynchronous and completely decentralized reconfiguration algorithm for read/write storage.

Cloud storage is another setting where reliability is a challenge. While cloud storage becomes increasingly popular, repeated incidents show that clouds fail in a variety of ways. Yet, clients must currently trust cloud providers to handle their information correctly, and do not have tools to verify this. Previously proposed solutions that aim to protect clients from faulty cloud storage providers sacrifice liveness of client operations in the normal case, when the storage is working properly. For example, if a client crashes in the middle of making an update to a remote object, no other client can ever read the same object. We prove that this problem is inherent in all theoretical semantics previously defined for this model. We define new semantics that can be guaranteed to clients even

when the storage is faulty without sacrificing liveness, and present FAUST, an algorithm providing these guarantees. We then present Venus, a practical system based on a variation of FAUST. Venus guarantees data consistency and integrity to clients that collaborate using commodity cloud storage, and alerts clients when the storage is faulty or malicious (e.g., as a result of a software bug, misconfiguration, or a hacker attack). Venus does not require trusted components or changes to the storage provider. Venus offers simple semantics, which further enhances its usability. We evaluate Venus with Amazon S3, and show that it is scalable and adds no noticeable overhead to storage operations.

Chapter 1

Introduction

Enterprise storage systems are monolithic disk arrays currently used in many enterprise environments. These systems are built from expensive customized hardware and provide high reliability even in the most extreme and unlikely failure scenarios. These systems, however, have several important shortcomings. First, the organization of company storage in a single (or several) physical storage racks often causes the I/O ports and controllers of the storage to become a bottleneck. Second, the extensibility of such solutions is often limited, and many manufacturers maintain separate product lines of enterprise storage, suitable for enterprises of different scale (called "entry level", "midrange" and "high-end" by IBM and HP). Finally, these systems are very expensive.

Distributed storage architectures provide a cheap and scalable alternative to such enterprise storage systems. Such distributed architectures make use of many unreliable servers (or storage devices) and provide reliability through replication. Another popular alternative is cloud storage, offered remotely by multiple providers. This thesis deals with two problems concerning the reliability of such solutions. First, we study reconfiguration in distributed storage systems: the large number of fault-prone servers requires supporting dynamic changes when faulty servers are removed from the system and new ones are introduced. The second topic studied in this thesis is utrusted storage, namely, providing reliability when using remote cloud storage that by itself is untrusted and can be arbitrarily faulty.

Although distributed replication protocols have been extensively studied in the past, replication alone provides limited fault-tolerance – in an asynchronous system it is impossible to tolerate the failure of more than a minority of the replicas [5]. Reconfiguring the system, i.e., changing the

set of replicas, increases its fault-tolerance: Suppose that the data is replicated over 5 servers in an asynchronous system. Initially, only two replica failures can be tolerated. Obviously, if the data is now copied to new replicas then additional replicas can fail. Note, however, that even by only removing the faulty replicas we can gain fault-tolerance. If two replicas fail and we remove them from the system, our system is now composed of 3 replicas and we can tolerate one more failure, i.e., 3 replica failures overall, breaking the "minority barrier". Obviously, the failures must be gradual and additional failures can be allowed only after making sure that a majority of the new system configuration stores the data.

In order to maintain reliability when such changes occur and to avoid "split-brain" behavior, it is essential to ensure proper coordination, e.g., when multiple configuration changes occur simultaneously. Existing solutions are either centralized, or use strong synchronization algorithms (such as consensus) among the servers to agree on every change in the system. In fact, it was widely believed that reconfigurations require consensus and just like consensus cannot be achieved in asynchronous systems. In Chapter 3 we refute this belief and present DynaStore [3], an asynchronous and completely decentralized reconfiguration algorithm for read/write storage.

Another emerging alternative for enterprise storage systems is cloud storage, which is now provided by many companies remotely over the Internet. Cloud storage, and other cloud services, allows users to collaborate with each other and to access shared data from anywhere. Unlike enterprise storage solutions, cloud services allow users to acquire resources on-demand and pay only for the resources currently in use. However, as we explain in Chapter 4, concerns about the trustworthiness of cloud services abound [14]. It is important, for example, to guarantee the integrity of user data, and to ensure that different collaborating users see their data consistently. Very little research has previously tackled this subject. Moreover, the industry mostly focuses on securing the cloud provider, and not on protecting the client from possible cloud malfunctions.

In this work we develop tools and semantics that enable clients using online cloud storage to monitor the storage, making sure that the cloud behaves as expected. Our work enables a variety of applications that already use the cloud to benefit from increased security, and, no less important, it can encourage applications that require verifiable guarantees, and cannot afford to blindly trust the cloud, to consider taking advantage of what the cloud has to offer.

In Chapter 4 we define the Untrusted Storage model. Chapter 5 shows that traditional strong consistency semantics cannot be guaranteed with an untrusted remote storage [15] and studies

other semantics that can be ensured in this model even when the storage is faulty. We identify an important limitation with previously proposed solutions: These solutions sacrifice liveness of client operations in the normal case, when the storage is working properly. For example, if a client crashes in the middle of making an update to an object, no other client can ever read the same object. We prove that this problem is inherent in all previously defined theoretical semantics for this model [13, 12, 15].

In Chapter 6 we present FAUST [12], a service that utilizes techniques from both distributed computing and cryptography, guaranteeing meaningful semantics to clients even when the cloud provider is faulty. In the common case, when the storage is working properly, FAUST guarantees both strong data consistency and strong liveness for client operations.

Still, FAUST and previous work share a second problem that prevents their use with any of the currently available cloud storage: they require complicated protocols to be run between the clients and the cloud storage, instead of the currently available simple read/write storage interface. To solve this problem, we design and implement Venus [71], presented in Chapter 7. Venus is a verification service for monitoring the integrity and consistency of cloud storage. Venus can be used with unmodified commodity cloud storage. In addition, Venus offers much simpler semantics than FAUST and the previous solutions, which further enhances its usability. Venus copes with failures ranging from simple data corruption to malicious failures of the cloud. We evaluated Venus with Amazon S3, and showed that it is scalable and adds no noticeable overhead to storage operations.

Chapter 2

Related Work

In Section 2.1 we review previous work on reconfiguration in distributed storage systems. Then, Section 2.2 focuses on previous work related to interaction with a remote untrusted storage.

2.1 Reconfigurable Distributed Storage

Several existing solutions can be viewed in retrospect as solving a dynamic (reconfigurable) storage problem. Most closely related are works on reconfigurable R/W storage. RAMBO [52, 31] solves a similar problem to the one addressed by DynaStore; other works [58, 67, 68] extend this concept for Byzantine fault tolerance. All of these works have processes agree upon a unique sequence of configuration changes. Some works use an auxiliary source (such as a single reconfigurer process or an external consensus algorithm) to determine configuration changes [50, 28, 52, 31, 58, 68, 35], while others implement fault-tolerant consensus decisions on view changes [17, 67]. In contrast, our work implements reconfigurable R/W storage without any agreement on view changes.

Since the closest related work is on RAMBO, we further elaborate on the similarities and differences between RAMBO and DynaStore. In RAMBO, a new configuration can be proposed by any process, and once it is installed, it becomes the current configuration. In DynaStore, processes suggest changes and not configurations, and thus, the current configuration is determined by the set of all changes proposed by complete reconfigurations. For example, if a process suggests to add p_1 and to remove p_2, while another process concurrently suggests to add p_3, DynaStore will install a configuration including both p_1 and p_3 and without p_2, whereas in RAMBO there is no

guarantee that any future configuration will reflect all three proposed changes, unless some process explicitly proposes such a configuration. In DynaStore, a quorum of a configuration is any majority of its members, whereas RAMBO allows for general quorum-systems, specified explicitly for each configuration by the proposing process. In both algorithms, a non-faulty quorum is required from the current configuration. A central idea in allowing dynamic changes is that a configuration can be replaced, after which a quorum of the old configuration can crash. In DynaStore, a majority of a current configuration C is allowed to crash as soon as C is no longer current. In RAMBO, two additional conditions are needed: C must be garbage-collected at every non-faulty process $p \in C$, and all *read* and *write* operations that began at p before C was garbage-collected must complete. Thus, whereas in DynaStore the conditions allowing a quorum of C to fail can be evaluated based on events visible to the application, in RAMBO these conditions are internal to the algorithm. Note that if some process $p \in C$ might fail, it might be impossible for other processes to learn whether p garbage-collected C or not. Assuming that all quorums required by RAMBO and DynaStore are responsive, both algorithms require additional assumptions for liveness. In both, the liveness of *read* and *write* operations is conditioned on the number of reconfigurations being finite. In addition, in both algorithms, the liveness of reconfigurations does not depend on concurrent *read* and *write* operations. However, whereas reconfigurations in RAMBO rely on additional synchrony or failure-detection assumptions required for consensus, reconfigurations in DynaStore, just like its *read* and *write* operations, only require the number of reconfigurations to be finite.

View-oriented group communication systems provide a membership service whose task is to maintain a dynamic view of active members. These systems solve a dynamic problem of maintaining agreement on a sequence of views, and additionally provide certain services within the members of a view, such as atomic multicast and others [19]. Maintaining agreement on group membership in itself is impossible in asynchronous systems [16]. However, perhaps surprisingly, we show that the dynamic R/W problem is solvable in asynchronous systems. This appears to contradict the impossibility but it does not: We do not implement group membership because our processes do not have to agree on and learn a unique sequence of view changes.

The State Machine Replication (SMR) approach [45, 69] provides a fault tolerant emulation of arbitrary data types by forming agreement on a sequence of operations applied to the data. Paxos [45] implements SMR, and allows one to dynamically reconfigure the system by keeping the configuration itself as part of the state stored by the state machine. Another approach for

reconfigurable SMR is to utilize an auxiliary configuration-master to determine view changes, and incorporate directives from the master into the replication protocol. This approach is adopted in several practical systems, e.g., [47, 53, 75], and is formulated in [46]. Naturally, a reconfigurable SMR can support our dynamic R/W memory problem. However, our work solves it without using the full generality of SMR and without reliance on consensus.

An alternative way to break the minority barrier in R/W emulation is by strengthening the model using a failure detector. Delporte et al. [26] identify the weakest failure detector for solving R/W memory with arbitrary failure thresholds. Their motivation is similar to ours– solving R/W memory with increased resilience threshold. Unlike our approach, they tackle more than a minority of failures right from the outset. They identify the *quorums failure detector* as the weakest detector required for strengthening the asynchronous model, in order to break the minority resilience threshold. Our approach is incomparable to theirs, i.e., our model is neither weaker nor stronger. On the one hand, we do not require a failure detector, and on the other, we allow the number of failures to exceed a minority only after certain actions are taken. Moreover, their model does not allow for additions as ours does. Indeed, our goal differs from [26], namely, our goal is to model dynamic reconfiguration in which resilience is adaptive to actions by the processes.

Data-centric read/write storage [20], where servers do not communicate with one another (and clients communicate directly with multiple servers), is considered in many works, e.g., [1, 18, 56, 58]. Most of these, however, assume a static world, where the set of servers is fixed from the outset. Two exceptions are Ursa Minor [1] and the work of Martin et al. [58] allow dynamicity and employ a centralized sequencer for configuration changes. Unlike Ursa Minor [1], the protocol of Martin et al. [58] allows read/write operations to continue during reconfigurations. DynaStore works in a different model, where a client submits operations to any one of the servers, which in turn contacts other servers and then replies to the client. DynaStore is completely decentralized and allows read/write operations to continue while reconfigurations are in progress. In a followup work [72], we design an implement DynaDisk, a data-centric version of DynaStore, which retains these properties of DynaStore.

Friedman et al. [30] implement atomic read/write objects in a data-centric dynamic system. Their solution assumes two abstractions – dynamic quorums and persistent reliable broadcast. Our work does not assume any high-level abstractions – we design low-level mechanisms that can preserve consistency in face of reconfigurations. In [30], dynamic service liveness is stated in terms

of properties that must be preserved by their quorum and broadcast abstractions. In particular, it is required that typed quorums (e.g., a read and a write quorum) accessed by two consecutive read/write operations intersect. In contrast, in DynaStore, it is possible for no such intersection to exist, due to *reconfig* operations that completely change system membership between two consecutive read/write operations. In part, this difference stems from the fact that we explicitly model *reconfig* operations, and treat them similarly to reads and writes. DynaStore also uses a broadcast primitive, however we only assume delivery to processes in the same configuration, and explicitly make sure that the information propagates to following configurations, which can implement the persistent broadcast of [30]. As do we, Friedman et al. [30] assume a crash model, where servers must change their identifiers if they wish to re-join the system. A minor difference is that Friedman et al. [30] assume an infinite arrival process with finite concurrency [61], where finitely many clients take steps during any finite time interval, whereas in DynaStore we allow infinitely many read/write operations to execute concurrently, as long as the number of concurrent *reconfig* operations is finite (this is modeled by assuming a finite number of membership changes proposed in the execution).

2.2 Untrusted Storage

Data integrity on untrusted storage accessed by a single client with small trusted memory can be protected by storing the root of a hash tree locally [8]. In cryptographic storage systems with multiple clients, such "root hashes" are signed; TDB [54], SiRiUS [32], and Plutus [40] are some representative examples implementing this method. In order to ensure freshness, the root hashes must be propagated by components that are at least partially trusted, however. Going beyond ensuring the integrity of data that is actually read from an untrusted service by a single client, recent work by Juels and Kaliski [39] and by Ateniese et al. [4] introduces protocols for assuring the client that it can retrieve its data in the future, with high probability. Unlike FAUST and Venus, this work does not guarantee consistency for multiple clients accessing the data concurrently.

One of the key principles in our solutions is that clients must be able to detect server (i.e., cloud storage) malfunction, i.e., its inability to provide normal service. This principle is known as fail-awareness [29] and it has been previously exploited by many systems in the timed asynchronous model, where nodes are subject to crash failures [23]. 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.

Several recent systems provide integrity using trusted components, which cannot be subverted by intrusions. In contrast, the solutions presented in this thesis use client signatures on the data, but no trusted components. The CATS system [78] 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 our service 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, *attested append-only memory* [21] introduces the abstraction of a small trusted module, implemented in hardware or software, which provides the function of an immutable log to clients in a distributed system. The *A2M-Storage* [21] service relying on such a module for consistency guarantees linearizability, even when the server is faulty. Although FAUST guarantees weaker consistency when the server is faulty, its liveness guarantee is stronger than that of A2M-Storage in the common case, when the storage is correct. Specifically, A2M storage has two variants: an "pessimistic" protocol, where a client first reserves a sequence number for an operation and then submits the actual operation with that sequence number, and an "optimistic" protocol, where the client submits an operation right away, optimistically assuming that it knows the latest sequence number, and then restarts in case its sequence number was in fact outdated. When the server is correct, the pessimistic version of the protocol may prevent progress from all clients in case some client fails after reserving a sequence number but before submitting the actual operation. The optimistic protocol has stronger liveness, and specifically it is lock-free, i.e., *some* client always makes progress (but progress is not guaranteed for every individual client). On the other hand, FAUST guarantees wait-freedom when the server is correct, i.e., *all* clients can complete their operations regardless of failures or concurrent operations executed by other clients.

When using untrusted remote (cloud) storage without the use of external trusted components, strong consistency semantics, such as linearizability [37] or sequential consistency cannot be guar-

anteed [15]. This limitation holds unless clients can communicate with other clients before completing each operation. In practice, clients should be able to complete operations independently, i.e., in a wait-free [36] manner. Intuitively, even if the clients sign all their updates, the storage can always hide client updates and create "split-brain" scenarios where clients believe they execute in isolation. In order to provide wait-freedom when linearizability cannot be guaranteed, numerous real-world systems guarantee eventual consistency, for example, Coda [41], Bayou [74], Tempest [57], and Dynamo [25]. 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 pioneering work of Mazières and Shasha [59] introduces untrusted storage protocols and the notion of fork-linearizability (under the name of *fork consistency*). To date, this is the strongest known consistency notion that can be achieved with a possibly Byzantine remote storage server when the clients do not communicate with one another. SUNDR [48] and later work [15] implement storage systems respecting this notion. The weaker notion of *fork-sequential consistency* has been suggested by Oprea and Reiter [64]. Neither fork-linearizability nor fork-sequential consistency can guarantee wait-freedom for client operations in all executions where the server is correct [15, 13].

Fork-*-linearizability [49] 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. We show that in the single server setting, just like the other consistency notions mentioned above, fork-* consistency does not allow for protocols that are always wait-free when the server is correct.

Orthogonal to this work, many storage systems have been proposed that internally use replication across several nodes to tolerate a fraction of corrupted nodes (e.g., [34] and references therein). For instance, HAIL [10] is a recent system that relies replicated storage servers internally, of which at least a majority must be correct at any time. It combines data replication with a method that gives proofs of retrievability to the clients. But a storage service employing replication within its cloud infrastructure does not solve the problem addressed by FAUST and Venus — from the perspective of the client, the cloud service is still a single trust domain.

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 [33]. Even-

tual consistency has recently been used in the context of Byzantine faults by Zeno [73]; 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, in which case inconsistencies are reconciled at a later point in time, whereas in FAUST linearizability can only be violated if the server is Byzantine, however the application might be notified of operation stability (consistency) after the operation completes (i.e., eventually).

Chapter 3

Dynamic Atomic Storage Without Consensus

This chapter deals with the emulation of atomic read/write (R/W) storage in dynamic asynchronous message passing systems. In static settings, it is well known that atomic R/W storage can be implemented in a fault-tolerant manner even if the system is completely asynchronous, whereas consensus is not solvable. In contrast, all existing emulations of atomic storage in dynamic systems rely on consensus or stronger primitives, leading to a popular belief that dynamic R/W storage is unattainable without consensus.

In this chapter, we specify the problem of dynamic atomic read/write storage in terms of the interface available to the users of such storage. We discover that, perhaps surprisingly, dynamic R/W storage is solvable in a completely asynchronous system: we present DynaStore, an algorithm that solves this problem. Our result implies that atomic R/W storage is in fact easier than consensus, even in dynamic systems.

A preliminary version of the work presented in this chapter appears in proceedings of the 28th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2009).

3.1 Introduction

Distributed systems provide high availability by replicating the service state at multiple processes. A fault-tolerant distributed system may be designed to tolerate failures of a minority of its processes. However, this approach is inadequate for long-lived systems, because over a long period, the chances of losing more than a minority inevitably increase. Moreover, system administrators may wish to deploy new machines due to increased workloads, and replace old, slow machines with new, faster ones. Thus, real-world distributed systems need to be *dynamic*, i.e., adjust their membership over time. Such dynamism is realized by providing users with an interface to *reconfiguration* operations that *add* or *remove* processes.

Dynamism requires some care. First, if one allows arbitrary reconfiguration, one may lose liveness. For example, say that we build a fault tolerant solution using three processes, p_1, p_2, and p_3. Normally, the adversary may crash one process at any moment in time, and the up-to-date system state is stored at a majority of the current configuration. However, if a user initiates the removal of p_1 while p_1 and p_2 are the ones holding the up-to-date system state, then the adversary may not be allowed to crash p_2, for otherwise the remaining set cannot reconstruct the up-to-date state. Providing a general characterization of allowable failures under which liveness can be ensured is a challenging problem.

A second challenge dynamism poses is ensuring safety in the face of concurrent reconfigurations, i.e., when some user invokes a new reconfiguration request while another request (potentially initiated by another user) is under way. Early work on replication with dynamic membership could violate safety in such cases [24, 66, 27] (as shown in [77]). Many later works have rectified this problem by using a centralized sequencer or some variant of consensus to agree on the order of reconfigurations.

Interestingly, consensus is not essential for implementing replicated storage. The ABD algorithm [5] shows that atomic *read/write (R/W)* shared memory objects can be implemented in a fault-tolerant manner even if the system is completely asynchronous. Nevertheless, to the best of our knowledge, all previous dynamic storage solutions rely on consensus or similar primitives, leading to a popular belief that dynamic storage is unattainable without consensus.

In this work, we address the two challenges mentioned above, and debunk the myth that consensus is needed for dynamic storage. We first provide a precise specification of a dynamic prob-

lem. To be concrete, we focus on atomic R/W storage, though we believe the approach we take for defining a dynamic problem can be carried to other problems. We then present *DynaStore*, a solution to this problem in an asynchronous system where processes may undetectably crash, so that consensus is not solvable. We note that our solution is given as a possibility proof, rather than as a blueprint for a new storage system. Given our result that consensus-less solutions are possible, we expect future work to apply various practical optimizations to our general approach, in order to build real-world distributed services. We next elaborate on these two contributions.

Dynamic Problem Specification

In Section 3.2, we define the problem of an atomic R/W register in a dynamic system. Similarly to a static R/W register, the dynamic variant exposes a *read* and *write* interface to users, and atomicity[44] is required for all such operations. In addition, users can trigger reconfigurations by invoking *reconfig* operations, which return OK when they complete. Exposing *reconfig* operations in the model allows us to provide a protocol-independent specification of service liveness guarantees, as we explain next.

Clearly, the progress of such a service is conditioned on certain failure restrictions in the deployed system. A fault model specifies the conditions under which progress is guaranteed. It is well understood how to state a liveness condition of the static version of this problem: t-resilient R/W storage guarantees progress if fewer than t processes crash. For an n-process system, it is well known that t-resilient R/W storage exists when $t < n/2$, and does not exist when $t \geq n/2$ (see [5]). A dynamic fault model serves the same purpose, but needs to additionally capture changes introduced by the user through the *reconfig* interface. Under reasonable use of *reconfig*, and some restricted fault conditions, the system will make progress. For example, an administrative-user can deploy machines to replace faulty ones, and thereby enhance system longevity. On the other hand, if used carelessly, reconfiguration might cause the service to halt, for example when servers are capriciously removed from the system.

Suppose the system initially has four processes $\{p_1, p_2, p_3, p_4\}$ in its configuration (also called its *view*). Initially, any one process may crash. Suppose that p_1 crashes. Then, additional crashes would lead to a loss of liveness. Now suppose that the user requests to reconfigure the system to remove p_1. While the request is pending, no additional crashes can happen, because the system must transfer the up-to-date state from a majority of the previous view to a majority of the new

one. However, once the removal is completed, the system can tolerate an additional crash among the new view $\{p_2, p_3, p_4\}$. Overall, two processes may crash during the execution. Viewed as a simple threshold condition, this exceeds a minority threshold, which contradicts lower bounds. The liveness condition we formulate is therefore not in the form of a simple threshold; rather, we require crashes to occur gradually, contingent on reconfigurations.

A dynamic system also needs to support additions. Suppose that the system starts with three processes $\{p_1, p_2, p_3\}$. In order to reconfigure the system to add a new process p_4, a majority of the view $\{p_1, p_2, p_3\}$ must be alive to effect the change. Additionally, a majority of the view $\{p_1, p_2, p_3, p_4\}$ must be alive to hold the state stored by the system. Again, the condition here is more involved than a simple threshold. That is, if a user requests to *add* p_4, then while the request is pending, a majority of both old and new views need to be alive. Once the reconfiguration is completed, the requirement weakens to a majority of the new view.

Given these, we state the following requirement for liveness for dynamic R/W storage: At any moment in the execution, let the *current view* consist of the initial view with all completed reconfiguration operations (add/remove) applied to it. We require that the set of crashed processes and those whose removal is pending be a minority of the current view, and of any pending future views. Moreover, like previous reconfigurable storage algorithms [52, 31], we require that no new *reconfig* operations will be invoked for "sufficiently long" for the started operations to complete. This is formally captured by assuming that only a finite number of *reconfig* operations are invoked.

Note that a dynamic problem is harder than the static variant. In particular, a solution to dynamic R/W is a fortiori a solution to the static R/W problem. Indeed, the solution must serve read and write requests, and in addition, implement reconfiguration operations. If deployed in a system where the user invokes only read and write requests, and never makes use of the reconfiguration interface, it must solve the R/W problem with precisely the same liveness condition, namely, tolerating any minority of failures. Similarly, dynamic consensus is harder than static consensus, and is therefore a fortiori not solvable in an asynchronous setting with one crash failure allowed. As noted above, in this thesis, we focus on dynamic R/W storage.

DynaStore: Dynamic Atomic R/W Storage

Our algorithm does not need consensus to implement reconfiguration operations. Intuitively, previous protocols used consensus, virtual synchrony, or a sequencer, in order to provide processes

with an agreed-upon sequence of configurations, so that the membership views of processes do not diverge. The key observation in our work is that it is sufficient that such a sequence of configurations exists, and there is no need for processes to know precisely which configurations belong to this sequence, as long as they have some assessment which includes these configurations, possibly in addition to others that are not in the sequence. In order to enable this property, in Section 3.3 we introduce *weak snapshots*, which are easily implementable in an asynchronous system. Roughly speaking, such objects support *update* and *scan* operations accessible by a given set of processes, such that *scan* returns a set of updates that, if non-empty, is guaranteed to include the *first update* made to the object (but the object cannot identify which update that is).

In DynaStore, which we present in Section 3.4, each view w has a weak snapshot object $ws(w)$, which stores reconfiguration proposals for what the next view should be. Thus, we can define a unique global sequence of views, as the sequence that starts with some fixed initial view, and continues by following the first proposal stored in each view's *ws* object. Although it is impossible for processes to learn what this sequence is, they can learn a DAG of views that includes this sequence as a path. They do this by creating a vertex for the current view, querying the *ws* object, creating an edge to each view in the response, and recursing. Reading and writing from a chain of views is then done in a manner similar to previous protocols, e.g., [52, 31, 17, 67, 68].

Summary of Contributions

In summary, our work makes two contributions.

- We define a dynamic R/W storage problem that includes a clean and explicit liveness condition, which does not depend on a particular solution to the problem. The definition captures a dynamically changing resilience requirement, corresponding to reconfiguration operations invoked by users. The approach easily carries to other problems, such as consensus. As such, it gives a clean extension of existing static problems to the dynamic setting.

- We discover that dynamic R/W storage is solvable in a completely asynchronous system with failures, by presenting a solution to this problem. Along the way we define a new abstraction of weak snapshots, employed by our solution, which may be useful in its own right. Our result implies that the dynamic R/W is weaker than the (dynamic) consensus problem, which is not solvable in this setting. This was known before for static systems, but

not for the dynamic version. The result counters the intuition that emanates from all previous dynamic systems, which used agreement to handle configuration changes.

3.2 Dynamic Problem Definition

We specify a read/write service with atomicity guarantees. The storage service is deployed on a collection of processes that interact using asynchronous message passing. We assume an unknown, unbounded and possibly infinite universe of processes Π. Communication links between all pairs of processes do not create, duplicate, or alter messages. Moreover, the links are reliable. Below we formally define reliable links in a dynamic setting.

Executions and histories. System components, namely the processes and the communication links between them, are modeled as I/O Automata [51]. An automaton has a state, which changes according to *transitions* that are triggered by *actions*, which are classified as input, output, and internal[1]. A *protocol* P specifies the behaviors of all processes. 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*.

The application interacts with the service via *operations* defined by the service interface. As operations take time[2], they are represented by two events – an *invocation* (input action) and a *response* (output action). A process p_i interacts with its incoming link from process p_j via the *receive*$(m)_{i,j}$ input action, and with its outgoing link to p_j via the *send*$(m)_{i,j}$ output action. The failure of process p_i is modeled using the input action *crash*$_i$, which disables all input actions at p_i. In addition, p_i can disable all input actions using the internal action *halt*$_i$.

A *history* of an execution consists of the sequence of invocations and responses occurring in the execution. An operation is *complete* in a history if it has a matching response. An operation o *precedes* another operation o' in a sequence of events σ, 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 precedes o' in σ then o precedes o' in π. Two operations are *concurrent* if neither one of them precedes the other. A sequence of events is *sequential* if it does not contain concurrent

[1] A minor difference from I/O Automata as defined in [51], is that in our model input actions can be disabled, as explained below.

[2] By slight abuse of terminology, we use the terms *operation* and *operation execution* interchangeably.

operations.

We assume that executions of our algorithm are *well-formed*, i.e., the sequence of events at each client consists of alternating invocations and matching responses, starting with an invocation. Finally, we assume that every execution is *fair*, which means, informally, that 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 [51]).

Service interface. We consider a multi-writer/multi-reader (MWMR) service, from which any process may read or write. The service stores a value v from a domain $\mathcal{V}$ and offers an interface for invoking *read* and *write* operations and obtaining their result. Initially, the service holds a special value $\perp \notin \mathcal{V}$. When a read operation is invoked at a process p_i, the service responds with a value x, denoted $\textit{read}_i() \rightarrow x$. When a write is invoked at p_i with a value $x \in \mathcal{V}$, denoted $\textit{write}_i(x)$, the response is OK. We assume that the written values are unique, i.e., no value is written more than once. This is done so that we are able to link a value to a particular write operation in the analysis, and can easily be implemented by having *write* operations augment the value with the identifier of the writer and a local sequence number.

In addition to *read* and *write* operations, the service exposes an interface for invoking reconfigurations. We define $\textit{Changes} \stackrel{def}{=} \{\textit{Remove, Add}\} \times \Pi$. We informally call any subset of Changes a *set of changes*. A *view* is a set of changes. A *reconfig* operation takes as parameter a set of changes c and returns OK. We say that a change $\omega \in \textit{Changes}$ is *proposed* in an execution if a $\textit{reconfig}_i(c)$ operation is invoked at some process p_i s.t. $\omega \in c$.

Intuitively, only processes that are members of the current system configuration should be allowed to initiate actions. To capture this restriction, we define an output action *enable operations*; the *read*, *write* and *reconfig* input actions at a process p_i are initially disabled, until an *enable operations* event occurs at p_i.

Safety specification. The *sequential specification* of the service indicates its behavior in sequential executions. It requires that each *read* operation returns the value written by the most recent preceding *write* operation, if there is one, and the initial value $\perp$ otherwise.

Atomicity [44], also called linearizability [37], requires that for every execution, there exist a corresponding sequential execution, which preserves the real-time order, and which satisfies the

sequential specification. Formally, let σ_{RW} be the sub-sequence of a history σ consisting of all events corresponding to the *read* and *write* operations in σ, without any events corresponding to *reconfig* operations. Linearizability is defined as follows:

Definition 1 (linearizability [37]). A history σ is *linearizable* if σ_{RW} can be extended (by appending zero or more response events) to a history σ', and there exists a sequential permutation π of the sub-sequence of σ' consisting only of complete operations such that:

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

Active processes and reliable links. We assume a non-empty view *Init*, which is initially known to every process in the system. We say, by convention, that a *reconfig*(*Init*) completes by time 0. A process p_i is *active* in an execution if p_i does not crash, some process invokes a *reconfig* operation to add p_i, and no process invokes a *reconfig* operation to remove p_i. We do not require all processes in Π to start taking steps from the beginning of the execution, but instead we assume that if p_i is active then p_i takes infinitely many steps (if p_i is not active then it may stop taking steps).

A common definition of reliable links states that if processes p_i and p_j are "correct", then every message sent by p_i to p_j is eventually received by p_j. We adapt this definition to a dynamic setting as follows: for a message sent at time t, p_j eventually receives the message if both p_i and p_j are active and a *reconfig*(c) operation was invoked by time t s.t. $(Add, j) \in c$.

Dynamic service liveness. We first give preliminary definitions, required to specify service liveness. For a set of changes w, the *removal-set of* w, denoted w.*remove*, is the set $\{i \,|\, \textit{(Remove, i)} \in w\}$. The *join set of* w, denoted w.*join*, is the set $\{i \,|\, \textit{(Add, i)} \in w\}$. Finally, the *membership of* w, denoted w.*members*, is the set w.*join*$\backslash w$.*remove*.

At any time t in the execution, we define $V(t)$ to be the union of all sets c s.t. a *reconfig(c)* completes by time t. Thus, $V(0) =$ *Init*. Note that removals are permanent, that is, a process that is removed will never again be in members. More precisely, if a reconfiguration removing p_i from the system completes at time t_0, then p_i is excluded from $V(t)$.*members*, for every $t \geq t_0$[3]. Let $P(t)$ be the set of *pending changes* at time t, i.e., for each element $\omega \in P(t)$ some process invokes

[3]In practice, one can add back a process by changing its id.

a *reconfig*(c) operation s.t. $\omega \in c$ by time t, and no process completes such a *reconfig* operation by time t. Denote by $F(t)$ the set of processes that crashed by time t.

Intuitively, any pending future view should have a majority of processes that did not crash and were not proposed for removal; we specify a simple condition sufficient to ensure this. A dynamic R/W service guarantees the following liveness properties:

Definition 2. [Dynamic Service Liveness]
If at every time t in the execution, fewer than $|V(t).\textit{members}|/2$ processes out of $V(t).\textit{members} \cup P(t).\textit{join}$ are in $F(t) \cup P(t).\textit{remove}$, and the number of different changes proposed in the execution is finite, then the following holds:

1. Eventually, the *enable operations* event occurs at every active process that was added by a complete *reconfig* operation.

2. Every operation invoked at an active process eventually completes.

3.3 The Weak Snapshot Abstraction

A weak snapshot object S accessible by a set P of processes supports two operations, $update_i(c)$ and $scan_i()$, for a process $p_i \in P$. The $update_i(c)$ operation gets a value c and returns OK, whereas $scan_i()$ returns a set C of values. Note that the set P of processes is fixed (i.e., *static*). We require the following semantics from *scan* and *update* operations:

NV1 Let o be a *scan*$_i()$ operation that returns C. Then for each $c \in C$, an *update*(c) operation is invoked by some process prior to the completion of o.

NV2 Let o be a *scan*$_i()$ operation that is invoked after the completion of an *update*$_j(c)$ operation, and that returns C. Then $C \neq \emptyset$.

NV3 Let o be a *scan*$_i()$ operation that returns C and let o' be a *scan*$_j()$ operation that returns C' and is invoked after the completion of o. Then $C \subseteq C'$.

NV4 There exists c such that for every non-empty set C returned by a *scan*$()$ operation, it holds that $c \in C$.

Algorithm 1 Weak snapshot - code for process p_i.

```
1:  operation update_i(c)
2:      if collect() = ∅ then
3:          Mem[i].Write(c)
4:      return OK
5:  operation scan_i()
6:      C ← collect()
7:      if C = ∅ then return ∅
8:      C ← collect()
9:      return C
10: procedure collect()
11:     C ← ∅;
12:     for each p_k ∈ P
13:         c ← Mem[k].Read()
14:         if c ≠ ⊥ then C ← C ∪ {c}
15:     return C
```

NV5 If some majority M of processes in P keep taking steps then every *scan*$_i()$ and *update*$_i(c)$ invoked by every process $p_i \in M$ eventually completes.

Although these properties bear resemblance to the properties of atomic snapshot objects [2], NV1-NV5 define a weaker abstraction: we do not require that *all update*s are ordered as in atomic snapshot objects, and even in a sequential execution, the set returned by a *scan* does not have to include the value of the most recently completed *update* that precedes it. Intuitively, these properties only require that the "first" *update* is seen by all *scan*s that see any *update*s. As we shall see below, this allows for a simpler implementation than of a snapshot object.

DynaStore will use multiple weak snapshot objects, one of each view w. The weak snapshot of view w, denoted *ws*(w), is accessible by the processes in w.*members*. To simplify notation, we denote by *update*$_i(w, c)$ and *scan*$_i(w)$ the $update$ and $scan$ operation, respectively, of process p_i of the weak snapshot object *ws*(w). Intuitively, DynaStore uses weak snapshots as follows: in order to propose a set of changes c to the view w, a process p_i invokes *update*$_i(w, c)$; p_i can then learn proposals of other processes by invoking *scan*$_i(w)$, which returns a set of sets of changes.

Implementation Our implementation of *scan* and *update* is shown in Algorithm 1. It uses an array *Mem* of $|P|$ single-writer multi-reader (SWMR) atomic registers, where all registers are initialized to $\perp$. Such registers support *Read*$()$ and *Write*(c) operations s.t. only process $p_i \in P$ invokes *Mem*$[i]$.*Write*(c) and any process $p_j \in P$ can invoke *Mem*$[i]$.*Read*$()$. The implementation

of such registers in message-passing systems is described in the literature [5].

A *scan*$_i()$ reads from all registers in *Mem* by invoking *collect*, which returns the set C of values found in all registers. After invoking *collect* once, *scan*$_i()$ checks whether the returned C is empty. If so, it returns $\emptyset$, and otherwise invokes *collect* one more time. An *update*$_i(c)$ invokes *collect*, and in case $\emptyset$ is returned, writes c to *Mem*$[i]$. Intuitively, if *collect*$()$ returns a non-empty set then some process has already proposed changes to the view, and thus, the weak snapshot does not correspond to the most up-to-date view in the system and there is no need to propose additional changes to this view.

Standard emulation protocols for atomic SWMR registers [5] guarantee integrity (property NV1) and liveness (property NV5). We next explain why Algorithm 1 preserves properties NV2-NV4; the formal proof of correctness appears in Section 3.5. First, notice that for a given i at most one *Mem*$[i]$.*Write* operation can be invoked in the execution, since after the first *Mem*$[i]$.*Write* operation completes, any *collect* invoked by p_i (the only writer of this register) will return a non-empty set and p_i will never invoke another *Write*. This together with atomicity of all registers in *Mem* implies properties NV2-NV3. Property NV4 stems from the fact that every *scan*$()$ operation that returns $C \neq \emptyset$ executes *collect* twice. Observe that such operation o that is the first to complete one *collect*. Any other *scan*$()$ operation o' begins its second *collect* only after o completes its first *collect*. Atomicity of the registers in *Mem* along with the fact that each register is written at-most once, guarantee that any value returned by a *Read* during the first *collect* of o will be read during the second *collect* of o'.

3.4 DynaStore

This section describes DynaStore, an algorithm for multi-writer multi-reader (MWMR) atomic storage in a dynamic system, which is presented in Algorithm 2. A key component of our algorithm is a procedure *ContactQ* (lines 31-41) for reading and writing from/to a quorum of members in a given view, used similarly to the *communicate* procedure in ABD [5]. When there are no reconfigurations, *ContactQ* is invoked twice by the *read* and *write* operations – once in a read-phase and once in a write-phase. More specifically, both *read* and *write* operations first execute a read-phase, where they invoke *ContactQ* to query a quorum of the processes for the latest value and timestamp, after which both operations execute a write-phase as follows: a *read* operation invokes

ContactQ again to write-back the value and timestamp obtained in the read-phase, whereas a *write* operation invokes *ContactQ* with a higher and unique timestamp and the desired value.

To allow reconfiguration, the members of a view also store information about the current view. They can change the view by modifying this information at a quorum of the current view. We allow the reconfiguration to occur concurrently with any *read* and *write* operations. Furthermore, once reconfiguration is done, we allow future reads and writes to use (only) the new view, so that processes can be expired and removed from the system. Hence, the key challenge is to make sure that no reads linger behind in the old view while updates are made to the new view. Atomicity is preserved using the following strategy.

- The read-phase is modified so as to first read information on reconfiguration, and then read the value and its timestamp. If a new view is discovered, the read-phase repeats with the new view.

- The write-phase, which works in the last view found by the read-phase, is modified as well. First, it writes the value and timestamp to a quorum of the view, and then, it reads the reconfiguration information. If a new view is discovered, the protocol goes back to the read-phase (the write-phase begins again when the read-phase ends).

- The *reconfig* operation has a preliminary phase, writing information about the new view to the quorum of the old one. It then continues by executing the phases described above, starting in the old view.

The core of a read-phase is procedure *ReadInView*, which reads the configuration information (line 67) and then invokes *ContactQ* to read the value and timestamp from a quorum of the view (line 68). It returns a non-empty set if a new view was discovered in line 67. Similarly, procedure *WriteInView* implements the basic functionality of the write-phase, first writing (or writing-back) the value and timestamp by invoking *ContactQ* in line 73, and then reading configuration information in line 74 (we shall explain lines 71-72 in Section 3.4.3).

We next give intuition into why the above regime preserves read/write atomicity, by considering the simple case where only one reconfiguration request is ever invoked, *reconfig*(c), from c_1 to c_2 (where $c_2 = c_1 \cup c$); we shall refer to this reconfiguration operation as *RC*. Figure 3.1(a) depicts a scenario where *RC*, invoked by process p_1, completes while a second process p_2 concurrently

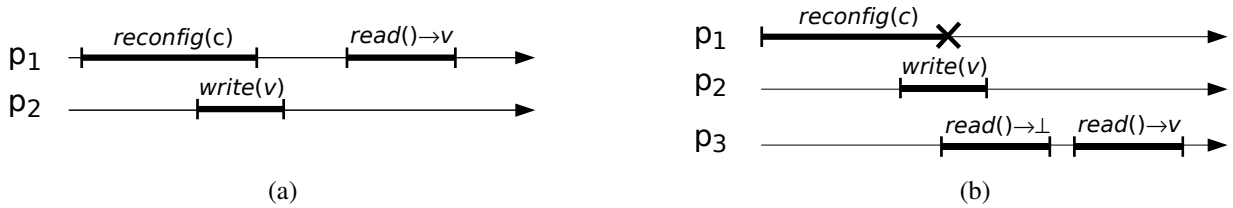

Figure 3.1: Operation flow in DynaStore. (a) A *reconfig* operation from c_1 to c_2 is concurrent with a *write*(v); one of them writes v to c_2. (b) The *reconfig* fails; either the first *read* completes in c_1, or the *write* writes v in c_2.

performs a *write*(v) operation. In our scenario p_2 is not initially aware of the existence of c_2, and hence the *write* operation performs a write-phase W writing in c_1 the value v with timestamp ts. After the *write* completes, p_1 executes a *read* operation, which returns v (the only possible return value according to atomicity). The *read* operation starts with a read-phase which operates in c_2 – the latest view known to p_1. Therefore, for v to be returned by the *read*, our algorithm must make sure that v and ts are transferred to c_2 by either *RC* or the *write* operation.

There are two possible cases with respect to *RC*. The first case is that *RC*'s read-phase observes W, i.e., during the execution of *ContactQ* in the read-phase of *RC*, p_1 receives v and ts from at least one process. In this case, *RC*'s write-phase writes-back v and ts into c_2. The second case is that *RC*'s read-phase does not observe W. In this case, as was explained above, our algorithm must not allow the *write* operation to complete without writing the value and timestamp to a quorum of the new view c_2. We next explain how this is achieved. Since *RC*'s read-phase does not observe W, when *RC* invokes *ContactQ* during its read-phase, W's execution of *ContactQ* writing a quorum of c_1 has not completed yet. Thus, W starts to read c_1's configuration information after *RC*'s preliminary phase has completed. This preliminary phase writes information about c_2 to a majority of c_1. Therefore, W discovers c_2 and the write operation continues in c_2.

Figure 3.1(b) considers a different scenario, where p_1 fails before completing *RC*. Again, we assume that p_2 is not initially aware of c_2, and hence the *write* operation performs a write-phase W in c_1 writing the value v with timestamp ts. Concurrently with p_2's *write*, p_3 invokes a *read* operation in c_1. Atomicity of the register allows this *read* to return either v or $\perp$, the initial value of the register; in the scenario depicted in Figure 3.1(b) $\perp$ is returned. After the *write* operation completes, p_3 invokes a second *read* operation, which returns v (the only possible value allowed by atomicity for this *read*). There are two cases to consider, with respect to the view in which the first read executes its final phase. The simple case is when this view is c_1. Then, the second *read*

starts by executing a read-phase in c_1 and hence finds out about v.

The second case is more delicate, and it occurs when the first *read* completes in c_2. Recall that this *read* returns $\perp$ and thus it does not observe W and the latest value v. Nevertheless, since the second *read* starts with a read-phase in c_2, the algorithm must ensure that v is stored at a quorum of c_2. This is done by the *write* operation, as we now explain. Since the first *read* operation starts in c_1 but completes in c_2, it finds c_2 when reading the reconfiguration information during a read-phase R in c_1. Since R does not observe W, it must be that W completes its *ContactQ* writing a majority of c_1 only after R invokes its *ContactQ* reading from a majority of c_1. Since R inspects reconfiguration information *before* invoking *ContactQ* while W does so *after* completing *ContactQ*, it must be that W starts inspecting reconfiguration information after R has finished inspecting reconfiguration information. Property NV3 guarantees that W finds all configuration changes observed by R, and hence finds out about c_2. Consequently, the *write* operation continues in c_2 and completes only after writing v in c_2. Here, it is important that the read-phase reads reconfiguration information *before* it performs *ContactQ*, while the write-phase reads reconfiguration information *after* it performs *ContactQ*. This inverse order is necessary to ensure atomicity in this scenario.

In our examples above, additional measures are needed to preserve atomicity if several processes concurrently propose changes to c_1. Thus, the rest of our algorithm is dedicated to the complexity that arises due to multiple contending reconfiguration requests. Our description is organized as follows: Section 3.4.1 introduces the pseudo-code of DynaStore, and clarifies its notations and atomicity assumptions. Section 3.4.2 presents the DAG of views, and shows how every operation in DynaStore can be seen as a traversal on that graph. Section 3.4.3 discusses *reconfig* operations. Finally, Section 3.4.4 presents the notion of established views, which is central to the analysis of DynaStore. Proofs are deferred to Section 3.6.

3.4.1 DynaStore Basics

DynaStore uses *operations*, *upon clauses*, and *procedures*. Operations are invoked by the application, whereas upon-clauses are triggered by messages received from the network: whenever a process p_i receives a message m from p_j (through a *receive*$(m)_{i,j}$ input action), m is stored in a buffer (this is not shown in the pseudo-code). The upon-clause is an internal action enabled when some condition on the message buffer holds. Procedures are called from an operation. Operations

and local variables at process p_i are denoted with subscript i.

Whereas upon-clauses are atomic, for simplicity of presentation, we do not formulate operations as atomic actions in the pseudo-code (with slight abuse of the I/O automata terminology), and operations sometimes block waiting for a response from a majority of processes in a view (in lines 30, 37, 55, 67 and 74), either explicitly (in lines 30 and 37), or in the underlying implementation of a SWMR register (e.g., [5]) which is used in the construction of weak snapshots. Note, however, that it is a trivial exercise to convert the pseudo-code to the I/O automata syntax, as each operation is atomic until it blocks waiting for a majority and thus the operation can be divided into multiple atomic actions: initially an action corresponding to the code that precedes the *wait* statement executes, and when messages are received from a majority, the upon-clause receiving the messages uses an additional internal flag to enable the execution of the operation part following the *wait*, which forms another atomic action, and to disable code which precedes the *wait*.

Operations and upon-clauses access different variables for storing the value and timestamp[4]: v_i and ts_i are accessed in upon-clauses, whereas operations (and procedures) manipulate v_i^{max} and ts_i^{max}. Procedure *ContactQ* sends a write-request including v_i^{max} and ts_i^{max} (line 35) when writing a quorum, and a read-request (line 36) when reading a quorum ($msgNum_i$, a local sequence number, is also included in such messages). When p_i receives a write-request, it updates v_i and ts_i if the received timestamp is bigger than ts_i, and sends back a REPLY message containing the sequence number of the request (line 45). When a read-request is received, p_i replies with v_i, ts_i, and the received sequence number (line 46).

Every process p_i executing Algorithm 2 maintains a local estimation of the latest view, $curView_i$ (line 9), initialized to *Init* when the process starts. Although p_i is able to execute all event-handlers immediately when it starts, recall that invocations of *read*, *write* or *reconfig* operations at p_i are only allowed once they are enabled for the first time; this occurs in line 11 (for processes in *Init.join*) or in line 81 (for processes added later). If p_i discovers that it is being removed from the system, it simply halts (line 53). In this section, we denote changes of the form (Add, i) by $(+, i)$ and changes of the form $(Remove, i)$ by $(-, i)$.

[4]This allows for a practical optimization, whereby operations and upon clauses act like separate monitors: an operation can execute concurrently with an upon-clause, and at most one of each kind can be executed at a time.

Algorithm 2 Code for process p_i.

state
  $v_i \in \mathcal{V} \cup \{\perp\}$, initially $\perp$ // latest value received in a WRITE message
  $ts_i \in \mathbb{N}_0 \times (\Pi \cup \{\perp\})$, initially $(0, \perp)$ // timestamp corresponding to v_i (timestamps have selectors num and pid)
  $v_i^{max} \in \mathcal{V} \cup \{\perp\}$, initially $\perp$ // latest value observed in *Traverse*
  $ts_i^{max} \in \mathbb{N}_0 \times (\Pi \cup \{\perp\})$, initially $(0, \perp)$ // timestamp corresponding to v_i^{max}
  pickNewTS$_i \in \{$FALSE, TRUE$\}$, initially FALSE // whether *Traverse* should pick a new timestamp
  M_i: *set of messages*, initially $\emptyset$
  msgNum$_i \in \mathbb{N}_0$, initially 0 // counter for sent messages
  curView$_i \in$ *Views*, initially *Init* // latest view

initially:
  if $(i \in$ *Init.join*$)$ **then** *enable operations*

operation *read*$_i()$:
  pickNewTS$_i \leftarrow$ FALSE
  newView $\leftarrow$ *Traverse*$(\emptyset, \perp)$
  NotifyQ(*newView*)
  return v_i^{max}

operation *write*$_i(v)$:
  pickNewTS$_i \leftarrow$ TRUE
  newView $\leftarrow$ *Traverse*$(\emptyset, v)$
  NotifyQ(*newView*)
  return OK

operation *reconfig*$_i$(*cng*):
  pickNewTS$_i \leftarrow$ FALSE
  newView $\leftarrow$ *Traverse*$(cng, \perp)$
  NotifyQ(*newView*)
  return OK

procedure *NotifyQ*(w)
  if did not receive $\langle$NOTIFY, $w\rangle$ **then**
    send $\langle$NOTIFY, $w\rangle$ to w.*members*
  wait for $\langle$NOTIFY, $w\rangle$ from majority of w.*members*

procedure *ContactQ*(*msgType*, D)
  $M_i \leftarrow \emptyset$
  msgNum$_i \leftarrow$ *msgNum*$_i + 1$;
  if *msgType* = W **then**
    send $\langle$REQ,W,*msgNum*$_i$,v_i^{max},$ts_i^{max}\rangle$ to D
  else send $\langle$REQ, R, *msgNum*$_i\rangle$ to D
  wait until M_i contains $\langle$REPLY, *msgNum*$_i, \cdots\rangle$ from a majority of D
  if *msgType* = R **then**
    tm$\leftarrow$*max*$\{t$:$\langle$REPLY,*msgNum*$_i$,v,$t\rangle \in M_i\}$
    vm $\leftarrow$ value corresponding to *tm*
    if *tm* $> ts_i^{max}$ **then** $(v_i^{max}, ts_i^{max}) \leftarrow (vm, tm)$

upon receiving $\langle$REQ,*msgType*,*num*,v,*ts*$\rangle$ from p_j:
  if *msgType* = W **then**
    if $(ts > ts_i)$ **then** $(v_i, ts_i) \leftarrow (v, ts)$

procedure *Traverse*(*cng*, *v*)
  desiredView $\leftarrow$ *curView*$_i \cup$ *cng*
  Front $\leftarrow \{$*curView*$_i\}$
  do
    $s \leftarrow$ *min*$\{|\ell| : \ell \in$ *Front*$\}$
    $w \leftarrow$ any $\ell \in$ *Front* s.t. $|\ell| = s$
    if $(i \notin w$.*members*$)$ **then** *halt*$_i$
    if $w \neq$ *desiredView* **then**
      update$_i(w,$ *desiredView*$\backslash w)$
    ChangeSets $\leftarrow$ *ReadInView*(w)
    if *ChangeSets* $\neq \emptyset$ **then**
      Front $\leftarrow$ *Front* $\setminus \{w\}$
      for each $c \in$ *ChangeSets*
        desiredView $\leftarrow$ *desiredView* $\cup\, c$
        Front $\leftarrow$ *Front* $\cup \{w \cup c\}$
    else *ChangeSets* $\leftarrow$ *WriteInView*(w, v)
  while *ChangeSets* $\neq \emptyset$
  curView$_i \leftarrow$ *desiredView*
  return *desiredView*

procedure *ReadInView*(w)
  ChangeSets $\leftarrow$ *scan*$_i(w)$
  ContactQ(R, w.*members*)
  return *ChangeSets*

procedure *WriteInView*(w, v)
  if *pickNewTS*$_i$ **then**
    $(pickNewTS_i, v_i^{max}, ts_i^{max}) \leftarrow$ $(\text{FALSE}, v, (ts_i^{max}.num + 1, i))$
  ContactQ(W, w.*members*)
  ChangeSets $\leftarrow$ *scan*$_i(w)$
  return *ChangeSets*

upon receiving $\langle$NOTIFY, $w\rangle$ for the first time:
  send $\langle$NOTIFY, $w\rangle$ to w.*members*
  if $(curView_i \subset w)$ **then**
    pause any ongoing *Traverse*
    curView$_i \leftarrow w$
    if $(i \in w$.*join*$)$ **then** *enable operations*
    if paused in line 79, restart *Traverse* from line 48

upon receiving $\langle$REPLY, $\cdots\rangle$:
  add the message and its sender-id to M_i

3.4.2 Traversing the Graph of Views

Weak snapshots organize all views in a given execution into a DAG, where views are the vertices and there is an edge from a view w to a view w' whenever an *update*$_j(w, c)$ has been made in the execution by some process $j \in w$.*members*, updating *ws*(w) to include the change $c \neq \emptyset$ s.t. $w' = w \cup c$; $|c|$ can be viewed as the weight of the edge – the distance between w' and w in changes. Our algorithm maintains the invariant that $c \cap w = \emptyset$, and thus w' always contains more changes than w, i.e., $w \subset w'$. Hence, the graph of views is acyclic.

The main logic of Algorithm 2 lies in procedure *Traverse*, which is invoked by all operations. This procedure traverses the DAG of views, and transfers the state of the emulated register from view to view along the way. *Traverse* starts from the view *curView*$_i$. Then, the DAG is traversed in an effort to find all membership changes in the system; these are collected in the set *desiredView*. After finding all changes, *desiredView* is added to the DAG by updating the appropriate *ws* object, so that other processes can find it in future traversals.

The traversal resembles the well-known Dijkstra algorithm for finding shortest paths from a single source [22], with the important difference that our traversal modifies the graph. A set of views, *Front*, contains the vertices reached by the traversal and whose outgoing edges were not yet inspected. Initially, *Front* $= \{$*curView*$_i\}$ (line 49). Each iteration processes the vertex w in *Front* closest to *curView*$_i$ (lines 51 and 52).

During an iteration of the loop in lines 50–63, it might be that p_i already knows that w does not contain all proposed membership changes. This is the case when *desiredView*, the set of changes found in the traversal, is different from w. In this case, p_i installs an edge from w to *desiredView* using *update*$_i$ (line 55). As explained in Section 3.3, in case another update to *ws*(w) has already completed, *update* does not install an additional edge from w; the only case when multiple outgoing edges exist is if they were installed concurrently by different processes.

Next, p_i invokes *ReadInView*(w) (line 56), which reads the state and configuration information in this view, returning all edges outgoing from w found when scanning *ws*(w) in line 67. By property NV2, if p_i or another process had already installed an edge from w, a non-empty set of edges is returned from *ReadInView*. If one or more outgoing edges are found, w is removed from *Front*, the next views are added to *Front*, and the changes are added to *desiredView* (lines 59–61). If p_i does not find outgoing edges from w, it invokes *WriteInView*(w) (line 62), which writes the latest known value to this view and again scans *ws*(w) in line 74, returning any outgoing edges that

are found. If here too no edges are found, the traversal completes.

Notice that *desiredView* is chosen in line 52 only when there are no other views in *Front*, since it contains the union of all views observed during the traversal, and thus any other view in *Front* must be of smaller size (i.e., contain fewer changes). Moreover, when $w \neq$ *desiredView* is processed, the condition in line 54 evaluates to true, and *ReadInView* returns a non-empty set of changes (outgoing edges) by property NV2. Thus, *WriteInView*$(w, *)$ is invoked only when *desiredView* is the only view in *Front*, i.e., $w =$ *desiredView* (this transfers the state found during the traversal to *desiredView*, the latest-known view). For the same reason, when the traversal completes, *Front* $= \{$*desiredView*$\}$. Then, *desiredView* is assigned to *curView*$_i$ in line 64 and returned from Traverse.

To illustrate such traversals, consider the example in Figure 3.2. Process p_i invokes *Traverse* and let *initView* be the value of *curView*$_i$ when *Traverse* is invoked. Assume that *initView.members* includes at least p_1 and p_i, and that $cng = \emptyset$ (this parameter of *Traverse* will be explained in Section 3.4.3). Initially, its *Front*, marked by a rectangle in Figure 3.2, includes only *initView*, and *desiredView* $=$ *initView*. Then, the condition in line 54 evaluates to false and p_i invokes *ReadInView*(*initView*), which returns $\{\{(+,3)\}, \{(+,5)\}, \{(-,1), (+,4)\}\}$. Next, p_i removes *initView* from *Front* and adds vertices V_1, V_2 and V_3 to *Front* as shown in Figure 3.2. For example, V_3 results from adding the changes in $\{(-,1),(+,4)\}$ to *initView*. At this point, *desiredView* $=$ *initView*$\cup\{(+,3), (+,5), (-,1), (+,4)\}$. In the next iteration of the loop in lines 50–63, one of the smallest views in *Front* is processed. In our scenario, V_1 is chosen. Since $V_1 \neq$ *desiredView*, p_i installs an edge from V_1 to *desiredView*. Suppose that no other updates were made to *ws*(V_1) before p_i's update completes. Then, *ReadInView*(V_1) returns $\{\{(+,5),(-,1),(+,4)\}\}$ (properties NV1 and NV2). Then, V_1 is removed from *Front* and $V_4 = V_1 \cup \{(+,5),(-,1),(+,4)\}$ is added to *Front*. In the next iteration, an edge is installed from V_2 to V_4 and V_2 is removed from *Front*.

Now, the size of V_3 is smallest in *Front*, and suppose that another process p_j has already completed *update*$_j(V_3, \{(+,7)\})$. p_i executes *update* (line 55), however since an outgoing edge already exists, a new edge is not installed. Then, *ReadInView*(V_3) is invoked and returns $\{\{(+,7)\}\}$. Next, V_3 is removed from *Front*, $V_5 = V_3 \cup \{(+,7)\}$ is added to *Front*, and $(+,7)$ is added to *desiredView*. Now, *Front* $= \{V_4, V_5\}$, and we denote the new *desiredView* by V_6. When V_4 and V_5 are processed, p_i installs edges from V_4 and V_5 to V_6. Suppose that *ReadInView* of V_4 and V_5 in line 56 return only the edge installed in the preceding line. Thus, V_4 and V_5 are removed from *Front*, and

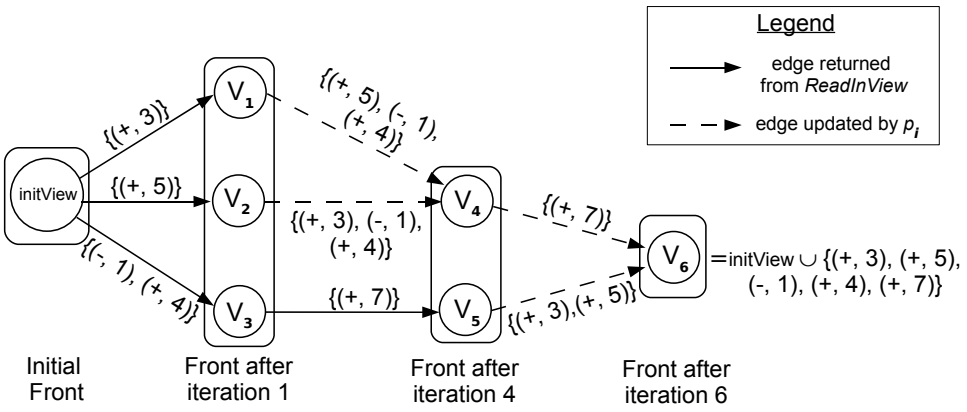

Figure 3.2: Example DAG of views.

V_6 is added to *Front*, resulting in *Front* $= \{V_6\}$. During the next iteration *ReadInView*(V_6) and *WriteInView*(V_6) execute and both return $\emptyset$ in our execution. The condition in line 63 terminates the loop, V_6 is assigned to *curView*$_i$ and Traverse completes returning V_6.

3.4.3 Reconfigurations (Liveness)

A *reconfig(cng)* operation is similar to a *read*, with the only difference that *desiredView* initially contains the changes in *cng* in addition to those in *curView*$_i$ (line 48). Since *desiredView* only grows during a traversal, this ensures that the view returned from *Traverse* includes the changes in *cng*. As explained earlier, *Front* $= \{$*desiredView*$\}$ when *Traverse* completes, which means that *desiredView* appears in the DAG of views.

When a process p_i completes *WriteInView* in line 62 of *Traverse*, the latest state of the register has been transfered to *desiredView*, and thus it is no longer necessary for other processes to start traversals from earlier views. Thus, after Traverse completes returning *desiredView*, p_i invokes *NotifyQ* with this view as its parameter (lines 15, 20 and 25), to let other processes know about the new view. *NotifyQ*(w) sends a NOTIFY message (line 29) to w*.members*. A process receiving such a message for the first time forwards it to all processes in w*.members* (line 77), and after a NOTIFY message containing the same view was received from a majority of w*.members*, *NotifyQ* returns. In addition to forwarding the message, a process p_j receiving $\langle$NOTIFY, $w\rangle$ checks whether *curView*$_j \subset w$ (i.e., w is more up-to-date than *curView*$_j$), and if so it pauses any ongoing *Traverse*, assigns w to *curView*$_j$, and restarts *Traverse* from line 48. As the execution of *Traverse* between *wait* statements is atomic, *Traverse* executed by p_j can be restarted only when it blocks waiting for

messages from a majority of some view w'. Restarting *Traverse* in such case can be necessary if less than a majority of members in w' are active. Intuitively, Definition 2 implies that in such case w' must be an old view, i.e., some *reconfig* operation completes proposing new changes to system membership. We prove in Section 3.6.3 that in this case p_j will receive a $\langle$NOTIFY, $w\rangle$ message s.t. $\mathit{curView}_j \subset w$ and restart its traversal.

Note that when a process p_i restarts *Traverse*, p_i may have an outstanding scan_i or update_i operation on a weak snapshot $\mathit{ws}(w)$ for some view w, in which case p_i restarts *Traverse* without completing the operation. It is possible that p_i might be unable to complete such outstanding operations because w is an old view, i.e., more than a majority of its members were removed. After *Traverse* is restarted, it is possible that p_i encounters w again in the traversal and needs to invoke another operation on $\mathit{ws}(w)$, in which case w is not known to be old. We require that in this case p_i first terminates previous outstanding operations on $\mathit{ws}(w)$ before it invokes the new operation. The mechanism to achieve this is a simple queue, and it is not illustrated in the code. Note that started snapshot operations on old views do not need to be completed.

Restarts of *Traverse* introduce an additional potential complication for *write* operations: suppose that during its execution of *write*(v), p_i sends a WRITE message with v and a timestamp ts. It is important that if *Traverse* is restarted, v is not sent with a different timestamp (unless it belongs to some other write operation). Before the first message with v is sent, we set the $\mathit{pickNewTS}_i$ flag to *false* (line 72). The condition in line 71 prevents *Traverse* from re-assigning v to v_i^{max} or incorrect ts_i^{max}, even if a restart occurs.

In Section 3.6.3 we prove that DynaStore preserves Dynamic Service Liveness (Definition 2). Thus, liveness is conditioned on the number of different changes proposed in the execution being finite. In reality, only the number of such changes proposed concurrently with every operation has to be finite. Then, the number of times Traverse can be restarted during that operation is finite and so is the number of views encountered during the traversal, implying termination.

3.4.4 Sequence of Established Views (Safety)

Our traversal algorithm performs a *scan*(w) to discover outgoing edges from w. However, different processes might invoke *update*(w) concurrently, and different *scan*s might see different sets of outgoing edges. In such cases, it is necessary to prevent processes from working with views on

different branches of the DAG. Specifically, we would like to ensure an intersection between views accessed in reads and writes. Fortunately, property NV4 guarantees that all *scan*(w) operations that return non-empty sets (i.e., return some outgoing edges from w), have at least one element (edge) in common. Note that a process cannot distinguish such an edge from others and therefore traverses all returned edges. This property of the algorithm enables us to define a totally ordered subset of the views, which we call *established*, as follows:

Definition 3. [Sequence of Established Views] The unique sequence of established views $\mathcal{E}$ is constructed as follows:

- the first view in $\mathcal{E}$ is the initial view *Init*;
- if w is in $\mathcal{E}$, then the next view after w in $\mathcal{E}$ is $w' = w \cup c$, where c is an element chosen arbitrarily from the intersection of all sets $C \neq \emptyset$ returned by some *scan*(w) operation in the execution.

Note that each element in the intersection mentioned in Definition 3 is a set of changes, and that property NV4 guarantees a non-empty intersection. In order to find such a set of changes c in the intersection, one can take an arbitrary element from the set C returned by the first *collect*(w) that returns a non-empty set in the execution. This unique sequence $\mathcal{E}$ allows us to define a total order relation on established views. For two established views w and w' we write $w \dot{\leq} w'$ if w appears in $\mathcal{E}$ no later than w'; if in addition $w \neq w'$ then $w \dot{<} w'$. Notice that for two established views w and w', $w \dot{<} w'$ if an only if $w \subset w'$.

Notice that the first graph traversal in the system starts from *curView*$_i$ = *Init*, which is established by definition. When *Traverse* is invoked with an established view *curView*$_i$, every time a vertex w is removed from *Front* and its children are added, one of the children is an established view, by definition. Thus, *Front* always includes at least one established view, and since it ultimately contains only one view, *desiredView*, we conclude that *desiredView* assigned to *curView*$_i$ in line 64 and returned from *Traverse* is also established. Thus, all views sent in NOTIFY messages or stored in *curView*$_i$ are established. Note that while a process p_i encounters all established views between *curView*$_i$ and the returned *desiredView* in an uninterrupted traversal, it only recognizes a subset of established views as such (whenever *Front* contains a single view, that view must be in $\mathcal{E}$).

It is easy to see that each traversal performs a *ReadInView* on every established view in $\mathcal{E}$ between $curView_i$ and the returned view *desiredView*. Notice that *WriteInView* (line 62) is always performed in an established view. Thus, intuitively, by reading each view encountered in a traversal, we are guaranteed to intersect any write completed on some established view in the traversed segment of $\mathcal{E}$. Then, performing the *scan* before *ContactQ* in *ReadInView* and after the *ContactQ* in *WriteInView* guarantees that in this intersection, indeed the state is transferred correctly, as explained in the beginning of this section. A formal correctness proof of our protocol appears in Section 3.6.

3.5 Analysis of Weak Snapshot Objects

First, note that whenever a process p_i performs $scan_i(w)$ or $update_i(w, c)$, it holds that $i \in w.members$ because of the check in line 53. Thus, it is allowed to perform these operations on w. The following lemmas prove correctness of a single weak snapshot object accessible by a set of processes P. We assume that all registers in *Mem* are initialized to $\perp$ and that no process invokes $update(\perp)$, which is indeed preserved by DynaStore. The first lemma shows that each register $Mem[i]$ can be assigned at most one non-initial value.

Lemma 1. *For any $i \in P$, the following holds: (a) if $Mem[i]$.Read() is invoked after the completion of $Mem[i]$.Write(c), and returns c', then $c' = c$; and (b) if two $Mem[i]$.Read() operations return $c \neq \perp$ and $c' \neq \perp$, then $c = c'$.*

Proof. Recall that only p_i can write to $Mem[i]$ (by invoking an *update* operation). We next show that $Mem[i]$.*Write* can be invoked at most once in an execution. Suppose by way of contradiction that $Mem[i]$.*Write* is invoked twice in the execution, and observe the second invocation. Section 3.4.3 mentions our assumption of a mechanism that always completes a previous operation on a weak snapshot object, if any such operation has been invoked and did not complete (because of restarts), whenever a new operation is invoked on the same weak snapshot object. Thus, when $Mem[i]$.*Write* is invoked for the second time, the first $Mem[i]$.*Write* has already completed. Before invoking the *Write*, p_i completes *collect*, which executes $Mem[i]$.*Read*. By atomicity of $Mem[i]$, since the first *Write* to $Mem[i]$ has already completed writing a non-$\perp$ value, *collect* returns a set containing this value, and the condition in line 2 in Algorithm 1 evaluates to FALSE, contradicting

our assumption that a *Write* was invoked after the *collect* completes.

(a) follows from atomicity of *Mem*$[i]$ since *Mem*$[i]$.*Write* is invoked at most once in the execution. In order to prove (b), notice that if $c \neq c'$, since p_i is the only writer of *Mem*$[i]$, this means that both *Mem*$[i]$.*Write*(c) and *Mem*$[i]$.*Write*(c') are invoked in the execution, which contradicts the fact that *Mem*$[i]$.*Write* is invoked at most once in the execution. □

Properties NV1 (integrity) and NV5 (liveness) can be guaranteed by using standard emulation protocols for atomic SWMR registers [5]. The following lemmas prove that Algorithm 1 preserves properties NV2-NV4.

Lemma 2. *Let o be a scan$_i()$ operation that is invoked after the completion of an update$_j(c)$ operation, and returns C. Then $C \neq \emptyset$.*

Proof. Since *update*$_j(c)$ completes, either *Mem*$[i].Write(c)$ completes or *collect* returns a non-empty set. In the first case, when o reads from *Mem*$[i]$ during both first and second *collect*, the *Read* returns c by Lemma 1. The second case is that *collect* completes returning a non-empty set. Thus, a read from some register *Mem*$[j]$ during this *collect* returns $c' \neq \perp$. By atomicity of *Mem*$[j]$ and Lemma 1, since o is invoked after *update*$_j(c)$ completes, any read from *Mem*$[j]$ performed during o returns c'. Thus, in both cases the first and second *collect* during o return a non-empty set, which means that $C \neq \emptyset$. □

Lemma 3. *Let o be a scan$_i()$ operation that returns C and let o' be a scan$_j()$ operation that returns C' and is invoked after the completion of o. Then $C \subseteq C'$.*

Proof. If $C = \emptyset$, the lemma trivially holds. Otherwise, consider any $c \in C$. Notice that c is returned by a *Read* r from some register *Mem*$[k]$ during the second *collect* of o. Atomicity of *Mem*$[k]$ and Lemma 1 guarantee that every *Read* r' from the same register invoked after the completion of r returns c. Both times *collect* is executed during o', it reads from *Mem*$[k]$ and since o' is invoked after o completes both times a set containing c is returned from *collect*, i.e, $c \in C'$. □

Lemma 4. *There exists c such that for every scan$()$ operation that returns $C' \neq \emptyset$, it holds that $c \in C'$.*

Proof. Let o be the first *scan$_i()$* operation during which *collect* in line 6 returns a non-empty set, and let $C \neq \emptyset$ be this set. Let o' be any *scan*$()$ operation that returns $C' \neq \emptyset$. We next show that

$C \subseteq C'$, which means that any $c \in C$ preserves the requirements of the lemma. Since $C' \neq \emptyset$, the first invocation of *collect*() during o' returns a non-empty set. By definition of o, the second *collect* during o' starts after the first *collect* of o completes. For every $c \in C$, there is a *Mem*$[k]$*.Read()* executed by the first *collect* of o that returns $c \neq \perp$. By Lemma 1 and atomicity of *Mem*$[k]$, a *Read* from the same register performed during the second *collect* of o' returns c. Thus, $C \subseteq C'$. □

3.6 Analysis Of DynaStore

3.6.1 *Traverse*

We use the convention whereby each time Traverse is restarted, a new execution of Traverse begins; this allows us to define one view from which a traversal starts – this is the value *curView*$_i$ when the execution of Traverse begins in line 48.

Lemma 5. *At the beginning and end of each iteration of the loop in lines 50-63, it holds that* $\bigcup_{w \in Front} w \subseteq$ *desiredView.*

Proof. We prove that if an iteration begins with $\bigcup_{w \in Front} w \subseteq$ *desiredView* then this invariant holds also when the iteration ends. The lemma then follows from the fact that at the beginning of the first iteration *Front* $= \{$*curView*$_i\}$ (line 49) and *desiredView* $=$ *curView*$_i \cup$ *cng* (line 48).

Suppose that at the beginning of an iteration $\bigcup_{w \in Front} w \subseteq$ *desiredView*. If the loop in lines 59-61 does not execute, then Front and *desiredView* do not change, and the condition holds at the end of the iteration. If the loop in lines 59-61 does execute, then $w \subseteq$ *desiredView* is removed from Front, $w \cup c$ is added to Front and c is added to *desiredView*, thus the condition is again true. □

Lemma 6. *Whenever update*$_i(w, c)$ *is executed,* $c \neq \emptyset$ *and* $c \cap w = \emptyset$.

Proof. *update*$_i(w, c)$ is executed only in line 55 when $w \neq$ *desiredView* and $c =$ *desiredView*$\backslash w$, which means that $c \cap w = \emptyset$. By Lemma 5, since $w \neq$ *desiredView*, it holds that $w \subset$ *desiredView*. Thus, $c =$ *desiredView*$\backslash w \neq \emptyset$. □

Lemma 7. *Let* T *be an execution of Traverse that starts from curView*$_i =$ *initView. For every view* w *that appears in Front at some point during the execution of* T*, it holds that initView* $\subseteq w$.

Proof. We prove that if an iteration of the loop in lines 50-63 begins such that each view in *Front* contains *initView*, then this invariant is preserved also when the iteration ends. The lemma then follows from the fact that at the beginning of the first iteration *Front* $= \{curView_i\}$ (line 49).

Suppose that at the beginning of an iteration each view in *Front* contains *initView*. *Front* can only change during this iteration if the condition in line 57 evaluates to true, i.e., if *ChangeSets* $\neq \emptyset$. In this case, the loop in lines 59-61 executes at least once, and $w \cup c$ is added to *Front* in line 61 for some c. Since w was in *Front* in the beginning of this iteration, by our assumption it holds that *initView* $\subseteq w$, and therefore $w \cup c$ also contains *initView*. □

Lemma 8. *Let $w \in$ Front be a view. During the execution of Traverse, if w is removed from Front in some iteration of the loop in lines 50-63, then the size of any view w' added to Front in the same iteration or a later one, is bigger than $|w|$.*

Proof. Suppose that w is removed from *Front* during an iteration. Then its size, $|w|$, is minimal among the views in *Front* (lines 51 and 52) at the beginning of this iteration. By line 61, whenever a view is inserted to *Front*, it has the form $w \cup c$ where $c \in$ *ChangeSets* returned by $scan_i$ in line 67. By property NV1, some $update(w, c)$ operation is invoked in the execution, and by Lemma 6, $c \neq \perp$ and $c \cap w = \emptyset$. Thus, the view $w \cup c$ is strictly bigger than w removed from *Front* in the same iteration. It follows that any view w' added to *Front* in this or in a later iteration has size bigger than $|w|$. □

Lemma 9. *If at some iteration of the loop in lines 50-63 ReadInView returns ChangeSets $= \emptyset$, then $w =$ desiredView and Front* = $\{$*desiredView*$\}$.

Proof. Suppose for the sake of contradiction that $w \neq$ *desiredView*. Before *ReadInView* is invoked, $update_i(w,$ *desiredView* $\setminus w)$ completes, and then by Lemma 2 when *ReadInView* completes it returns a non-empty set, a contradiction.

Suppose for the sake of contradiction that there exists a view $w' \in$ Front s.t. $w' \neq$ *desiredView*. By Lemma 5, $w' \subseteq$ *desiredView*. Since $w' \neq$ *desiredView*, we get that $w' \subset$ *desiredView* and thus $|w'| <$ |*desiredView*|, contradicting the fact that $w =$ *desiredView*, and not w', is chosen in line 52 in the iteration. □

Lemma 10. *desiredView returned from Traverse(cng, v) contains cng.*

Proof. At the beginning of Traverse, *desiredView* is set to $curView_i \cup cng$ in line 48, and during the execution of Traverse, no element is removed from *desiredView*. Thus, $cng \subseteq desiredView$ when Traverse completes. □

Lemma 11. *$curView_i$ is an established view. Moreover, desiredView in line 64 of Traverse is established and whenever WriteInView$(w, *)$ is invoked, w is an established view.*

Proof. We prove the lemma using the following claim:

Claim 11.1. *If $curView_i$ from which a traversal starts is an established view, then Front at the beginning and end of the loop in lines 50-63 contains an established view, and the view desiredView assigned to $curView_i$ in line 64 in Traverse is established. Moreover, whenever WriteInView$(w, *)$ is invoked, w is an established view.*

Proof. Initially, *Front* contains $curView_i$ (line 48), which is established by assumption, and therefore *Front* indeed contains an established view when the first iteration of the loop begins. If a view w is removed from *Front* in line 58, then $ChangeSets \neq \emptyset$. We distinguish between two cases: (1) if w is not an established view, then Front at the end of the iteration still contains an established view; (2) if w is an established view, then, by Lemma 4 and the definition of $\mathcal{E}$, since *ChangeSets* is a non-empty set returned by $scan_i(w)$, there exists $c \in ChangeSets$ such that $w \cup c$ is established. Since for every $c \in ChangeSets$, $w \cup c$ is added to *Front* in line 61, the established view succeeding w in the sequence is added to *Front*, and thus *Front* at the end of this iteration of the loop in lines 50-63 still contains an established view.

By Lemma 9, when the loop in lines 50-63 completes, as well as when *WriteInView*$(w, *)$ is invoked, Front $= \{desiredView\}$. Since during such iterations, *ReadInView* returns $\emptyset$, *Front* does not change from the beginning of the iteration. We have just shown that *Front* contains an established view at the beginning of the do-while loop, and thus, *desiredView* in line 64 is established, and so is any view w passed to *WriteInView*. □

We next show that the precondition of the claim above holds, i.e., that $curView_i$ is an established view, by induction on $|curView_i|$. The base is $curView_i = Init$, in which case it is established by definition. Assuming that $curView_i$ is established if its size is less than k, observe such view of size $k > |Init|$. Consider how $curView_i$ got its current value – it was assigned either by some earlier execution of Traverse at p_i in line 64, or in line 80 when a NOTIFY message is received,

which implies that some process completes a traversal returning this view. In either case, since *curView*$_i \neq$ *Init*, some process p_j has *desiredView* = *curView*$_i$ in line 64, while starting the traversal with a smaller view *curView*$_j$. Notice that *curView*$_j$ is established by our induction assumption, and since *curView*$_i$ is the value of *desiredView* in line 64 of a Traverse which started with an established view, it is also established by Claim 11.1. □

Lemma 12. *Let T be an execution of Traverse and initView be the value of curView$_i$ when p_i starts this execution, then (a) if T invokes WriteInView$(w, *)$ then T completes a ReadInView(w') which returns a non-empty set for every established view w' s.t. initView $\dot{\leq} w' \dot{<} w$, and a ReadInView(w) which returns $\emptyset$; and (b) if T reaches line 64 with desiredView $= w''$, then it completes WriteInView$(w'', *)$ which returns $\emptyset$.*

Proof. When T begins, the established view $w' =$ *initView* is the only view in Front. Since some iteration during T chooses w in lines 51 and 52, which has bigger size than w', it must be that w' is removed from Front. This happens only if some *ReadInView*(w') during T returns *ChangeSets* $\neq \emptyset$. After w' is removed from Front, for every $c \in$ *ChangeSets*, $w' \cup c$ is added to Front, and thus, the established view succeeding w' in $\mathcal{E}$ is added to *Front* (by Lemma 4 and the definition of $\mathcal{E}$). The arguments above hold for every established view w' s.t. *initView* $\dot{\leq} w' \dot{<} w$, since a bigger view w is chosen from Front during T. During the iteration when *WriteInView*$(w, *)$ is invoked, ReadInView(w) completes in line 56 and returns $\emptyset$, which completes the proof of (a).

Suppose that T reaches line 64 with *desiredView* $= w''$. By Lemma 9, w during the last iteration of the loop equals to w''. Observe the condition in line 63, which requires that *ChangeSets* $= \emptyset$ for the loop to end. Notice that *ChangeSets* is assigned either in line 56 or line 62. If it was assigned in line 62, then *WriteInView*$(w, *)$ was executed which completes the proof of (b). Otherwise, *ReadInView*(w) returns *ChangeSets* $= \emptyset$ in line 56, which causes line 62 to be executed. Then, since this is the last iteration, *WriteInView*$(w, *)$ returns $\emptyset$. □

3.6.2 Atomicity

We say that *WriteInView* writes a timestamp ts if ts_i^{max} sent in the REQ message by ContactQ(W, *) equals ts. Similarly, a *ReadInView* reads timestamp ts if at the end of ContactQ(R, *) invoked by the *ReadInView*, ts_i^{max} is equal to ts.

Lemma 13. *Let W be a WriteInView(w, *) that writes timestamp ts and returns C, and R be a ReadInView(w) that reads timestamp ts' and returns C'. Then, either $ts' \geq ts$ or $C' \subseteq C$. Moreover, if R is invoked after W completes, then $ts' \geq ts$.*

Proof. Because both operation invoke ContactQ in w, there exists a process p in *w.members* from which both W and R get a REPLY message before completing their ContactQ, i.e., p's answer counts towards the necessary majority of replies for both W and R. If p receives the $\langle \text{REQ}, \text{W}, \cdots \rangle$ message from W with timestamp ts before the $\langle \text{REQ}, \text{R}, \cdots \rangle$ message from R, then by lines 44 and 46 it responds to the message from R with a timestamp at least as big as ts. By lines 39-41, when R completes *ContactQ*(R, *w.members*), ts_i^{max} is set to be at least as high as ts, and thus $ts' \geq ts$. It is left to show that if p receives the $\langle \text{REQ}, \text{R}, \cdots \rangle$ message from R before the $\langle \text{REQ}, \text{W}, \cdots \rangle$ message from W, then $C' \subseteq C$.

Suppose that p receives the $\langle \text{REQ}, \text{R}, \cdots \rangle$ message from R first. Then, when this message is received by p, *ContactQ*(W, *w.members*) has not yet completed at W, and thus W has not yet invoked *scan*(w) in line 74. On the other hand, since R has started *ContactQ*(R, *w.members*), it has already completed its *scan*(w) in line 67, which returned C'. When W completes its ContactQ it invokes *scan*(w), which then returns C. By Lemma 3 it holds that $C' \subseteq C$.

Notice that if R is invoked after W completes then it must be the case that p receives the $\langle \text{REQ}, \text{W}, \cdots \rangle$ message from W first, and thus, in this case, $ts' \geq ts$. □

Lemma 14. *Let T be an execution of Traverse that completes returning w and upon completion its ts_i^{max} is equal to ts, and T' be an execution of Traverse that reaches line 64 with ts_i^{max} equal to ts' and with its desiredView equal to w'. If $w \dot{<} w'$ then $ts \leq ts'$.*

Proof. Consider the prefix of $\mathcal{E}$ up to w': $V_0, V_1, \cdots, V_l$ s.t. $V_0 = \textit{Init}$, $V_l = w'$, and $w = V_i$ where $i \in \{0, \ldots, l-1\}$. Moreover, let w'' be the view from which T' starts the traversal (w'' is established by Lemma 11).

First, consider the case that $w'' \dot{\leq} w$. By Lemma 12, since T returns w, it completes *WriteInView*($w, *$) which returns $C = \emptyset$. Since T' starts from $w'' \dot{\leq} w$ and reaches line 64 with *desiredView* $= w'$ s.t. $w \dot{<} w'$, by Lemma 12 it completes a *ReadInView*(w) which returns $C' \neq \emptyset$ (notice that *ReadInView*(w) might be executed in two consecutive iterations of T', in which case during the first iteration *ReadInView*(w) returns $\emptyset$; we then look on the next iteration, where a non-empty set is necessarily returned). Since $C' \not\subseteq C$, by Lemma 13 we have that ts_i^{max} upon the completion of

the *ReadInView*(w) by T' is at least as big as ts_i^{max} upon the completion of *WriteInView*$(w, *)$ by T, which equals to ts. Since ts_i^{max} does not decrease during T' and ts' is the value of ts_i^{max} when T' reaches line 64, we have that $ts' \geq ts$.

The second case to consider is $w \dot{<} w''$, which implies that $w'' \neq$ *Init*. In this case, there exists a traversal T'' which starts from a view $w''' \dot{<} w''$ and reaches line 64 before T begins, with *desiredView* $= w''$ (T'' is either an earlier execution of Traverse by the same process that executes T', or by another process, in which case T'' completes and sends a NOTIFY message with w'' which is then received by the process executing T' before T' starts). Let ts'' be the ts_i^{max} when T'' reaches line 64. Notice that T'' completes *WriteInView*$(w'', *)$ before T' starts *ReadInView*(w''), and by Lemma 13 when *ReadInView*(w'') completes at T' its ts_i^{max} is at least ts''. Since ts_i^{max} at T' can only increase from that point on, we get that $ts' \geq ts''$. It is therefore enough to show that $ts'' \geq ts$ in order to complete the proof. In order to do this, we apply the arguments above recursively, considering T'' instead of T', w'' instead of w' and ts'' instead of ts' accordingly (recall that $w \dot{<} w''$). Notice that since the prefix of $\mathcal{E}$ up to w' is finite, and since $w''' \dot{<} w''$, i.e., the starting point of T'' is before that of T' in $\mathcal{E}$, the recursion is finite and the starting point of the traversal we consider gets closer to *Init* in each recursive step. Therefore, the recursion will eventually reach a traversal which starts from an established view α and reaches line 64 with *desiredView* equal to an established view β s.t. $\alpha \dot{\leq} w$ and $w \dot{<} \beta$, which is the base case we consider. □

By definition of $\mathcal{E}$, if w is an established view then for every established view w' in the prefix of $\mathcal{E}$ before w (not including), some *scan*$_i(w')$ returns a non-empty set. However, the definition only says that such a *scan*$_i(w')$ exists, and not when it occurs. The following lemma shows that if w is returned by a Traverse T at time t, then some scan on w' returning a non-empty set must complete before time t. Notice that this scan might be performed by a different process than the one executing T.

Lemma 15. *Let T be an execution of Traverse that reaches line 64 at time t with desiredView equal to w s.t. $w \neq$ Init, and consider the prefix of $\mathcal{E}$ up to w: $V_0, V_1, \cdots, V_l$ s.t. $V_0 =$ Init and $V_l = w$. Then for every $k = 0, \ldots, l-1$, some scan(V_k) returns a non-empty set before time t.*

Proof. Since $w \neq$ *Init* there exists a traversal T' that starts from $V_i \dot{<} w$ and reaches line 64 with *desiredView* $= w$ no later than t. Notice that T' can be T if T starts from a view different than

w, or alternatively T' can be a traversal executed earlier by the same process, or finally, a traversal at another process that completes before T begins. By Lemma 12, a *ReadInView*(V_j) performed during T' returns a non-empty set for every $j = i, \ldots, l-1$. If $i = 0$ we are done. Otherwise, $V_i \neq$ *Init* and we continue the same argument recursively, now substituting V_l with V_i. Since the considered prefix of $\mathcal{E}$ is finite and since each time we recurse we consider a sub-sequence starting at least one place earlier than the previous starting point, the recursion is finite. □

Corollary 16. *Let T be an execution of Traverse that returns a view w and let T' be an execution of Traverse invoked after the completion of T, returning a view w'. Then $w \dot{\leq} w'$.*

Proof. First, note that by Lemma 11 both w and w' are established. Suppose for the purpose of contradiction that $w' \dot{<} w$. By Lemma 15, some *scan*(w') completes returning a non-empty set before T completes. Since T' returns w', its last iteration performs a *scan*(w') that returns an empty set. This contradicts Lemma 3 since T' starts after T completes. □

Corollary 17. *Let T be an execution of Traverse that returns a view w and let T' be an execution of Traverse invoked after the completion of T. Then T' does not invoke WriteInView$(w', *)$ for any view $w' \dot{<} w$.*

Proof. First, by Lemma 11, *WriteInView* is always invoked with an established view as a parameter. Suppose for the sake of contradiction that *WriteInView*$(w', *)$ is invoked during T' for some view $w' \dot{<} w$. Since T returns w and $w' \dot{<} w$, by Lemma 15 some *scan*(w') completes returning a non-empty set before T completes. Since T' invokes *WriteInView*$(w', *)$, by Lemma 12 a *ReadInView*(w') returned $\emptyset$ during T'. Thus, during the execution of this *ReadInView*(w'), a *scan*(w') returned $\emptyset$ during T'. This contradicts Lemma 3 since T' starts after T completes. □

We associate a timestamp with *read* and *write* operations as follows:

Definition 4 (Associated Timestamp). Let o be a *read* or *write* operation. We define $ats(o)$, the timestamp *associated* with o, as follows: if o is a *read* operation, then $ats(o)$ is ts_i^{max} upon the completion of Traverse during o; if o is a *write* operation, then $ats(o)$ equals to ts_i^{max} when its assignment completes in line 72.

Notice that not all operations have associated timestamps. The following lemma shows that all complete operations as well as writes that are read-from by some complete read operation have an associated timestamp.

Lemma 18. *We show three properties of associated timestamps: (a) for every complete operation o, $ats(o)$ is well-defined; (b) if o is a read operation that returns $v \neq \perp$, then there exists an $o' =$ write(v) operation such that $ats(o')$ is well-defined, and it holds that $ats(o) = ats(o')$; (c) if o and o' are write operations with associated timestamps, then $ats(o) \neq ats(o')$ and both are greater than $(0, \perp)$.*

Proof. There might be several executions of Traverse during a complete operation, but only one of these executions completes. Therefore, $ats(o)$ is well-defined for every complete *read* operation o. If o is a complete *write*, then notice that *pickNewTS*$_i$ = TRUE until it is set to FALSE in line 72, and therefore the condition in line 71 is TRUE until such time. Thus, for a *write* operation, line 72 executes at least once – in *WriteInView* which completes right before the completion of a Traverse during o (notice that *WriteInView* might be executed earlier as well). Once line 72 executes for the first time, *pickNewTS*$_i$ becomes FALSE. Thus, this line executes at-most once in every *write* operation and exactly once during a complete *write* operation, which completes the proof of (a).

To show (b), notice that v_i^{max} equals to v upon the completion of o. Moreover, since $v \neq \perp$, v is not the initial value of v_i^{max}. Observe the first operation o' that sets v_i^{max} to v during its execution, and notice that v_i^{max} is assigned only in lines 41 and 72. Suppose for the purpose of contradiction that the process executing o' receives v in a REPLY message from another process and sets v_i^{max} to v in line 41. A process p_i sending a REPLY message always includes v_i in this message, and v_i is set only to values received by p_i in $\langle \text{REQ}, \text{W}, \cdots \rangle$ messages. Thus, some process sends a $\langle \text{REQ}, \text{W}, \cdots \rangle$ message with v before o' sets its v_i^{max} to v. Since a $\langle \text{REQ}, \text{W}, \cdots \rangle$ message contains the v_i^{max} of the sender, we conclude that some process must have $v_i^{max} = v$ before o' sets its v_i^{max} to v, contradiction to our choice of o'. Thus, it must be that o' sets v_i^{max} to v in line 72. We conclude that o' is a *write*(v) operation which executes line 72. As mentioned above, this line is not executed more than once during o' and therefore $ats(o')$ is well-defined.

Recall our assumption that only one *write* operation can be invoked with v. Thus, o' is the operation that determines the timestamp with which v later appears in the system (any process that sets v_i to v, also sets ts_i to the timestamp sent with v by o', as the timestamp and value are assigned atomically together in line 44). This timestamp is $ats(o')$, determined when o' executes line 72. When o sets v_i^{max} to v, it also sets ts_i^{max} to $ats(o')$, as the timestamp and value are always assigned atomically together in line 41. Thus, $ats(o) = ats(o')$.

Finally, notice that the associated timestamp of a *write* operation is always of the form $(ts_i^{max}.num+$

$1, i)$, which is strictly bigger than $(0, \perp)$. Since i is a unique process identifier, if o and o' are two *write* operations executed by different processes, $ats(o) \neq ats(o')$. If they are executed by the same process, since ts_i^{max} pertains its value between operation invocations, increasing the first component of the timestamp by one makes sure that $ats(o) \neq ats(o')$, which completes the proof of (c). □

Lemma 19. *Let o and o' be two complete read or write operations such that o completes before o' is invoked. Then $ats(o) \leq ats(o')$ and if o' is a write operation, then $ats(o) < ats(o')$.*

Proof. Denote the complete execution of Traverse during o by T, and let w be the view returned by T and ts be the value of ts_i^{max} when T returns. Note that $ats(o) \leq ts$, since ts_i^{max} only grows during the execution of o, and if o is a *read* operation then $ats(o) = ts$. Notice that there might be several incomplete traversals during o' which are restarted, and there is exactly one traversal that completes.

There are two cases to consider. The first is that o' executes a *ReadInView*(w) that returns. Before this *ReadInView*(w) is invoked, T completes a *WriteInView*$(w, *)$, writing a value with timestamp ts. By Lemma 13, after the *ReadInView*(w) completes during o', $ts_i^{max} \geq ts \geq ats(o)$ and thus, when o' completes $ts_i^{max} \geq ats(o)$. If o' is a *read* operation then $ats(o')$ is equal to this ts_i^{max}, which proves the lemma. Suppose now that o' is a *write* operation. Then during o', *pickNewTS*$_i$ = TRUE until it is set to FALSE in line 72. By Corollary 17, no traversal during o' invokes *WriteInView* for any established view $\alpha \dot{<} w$. Thus, *ReadInView*(w) completes during o' before any *WriteInView* is invoked. By Lemma 18, $ats(o')$ is well-defined and therefore exactly one traversal during o' executes line 72. As explained, since *ReadInView*(w) has already completed when line 72 executes, $ts_i^{max} \geq ats(o)$ and then, ts_i^{max} is assigned $(ts_i^{max}.num + 1, i)$, implying that $ats(o') > ats(o)$.

The second case is that no *ReadInView*(w) completes during o'. Let T' be the traversal which determines $ats(o')$. Let w' be the view from which T' starts, and notice that since T' sets $ats(o')$, it completes *ReadInView*(w'). By Lemma 11, w' is an established view. We claim that $w \dot{<} w'$. First, if o' is a *read*, then T' completes and returns some view w''. By Corollary 16, $w \dot{\leq} w''$ and by Lemma 12, T' performs a *ReadInView* on all established views between w' and w''. Since o' does not complete *ReadInView*(w), it must be that $w \dot{<} w'$, which shows the claim. Now suppose that o' is a *write*. By Corollary 17, T' does not invoke *WriteInView*$(\alpha, *)$ for any view $\alpha \dot{<} w$. It is

also impossible that T' invokes *WriteInView*$(w, *)$ as it does not complete *ReadInView*(w). Thus, it must be that T' attains $ats(o')$ when it invokes *WriteInView*$(\alpha, *)$ where $w \dot{<} \alpha$. By Lemma 12, T' performs a *ReadInView* on all established views between w' and α. Since it does not complete *ReadInView*(w), it must be that $w \dot{<} w'$, which shows the claim.

Since $w \dot{<} w'$, $w' \neq$ *Init*. Moreover, since *curView*$_i = w'$ when T' starts, there exists a traversal T'' which reaches line 64 with *desiredView* equal to w' before T' begins. Let ts'' be the ts_i^{max} when T'' reaches line 64. By Lemma 14, since $w \dot{<} w'$, it holds that $ts \leq ts''$ and thus $ats(o) \leq ts''$. Since T'' performs *WriteInView*$(w', *)$ and after it completes, T' invokes and completes *ReadInView*(w'), by Lemma 13 we get that ts_i^{max} when *ReadInView*(w') completes is at least as high as ts''. If o' is a *read*, then $ats(o')$ equals to ts_i^{max} when T' completes, and since ts_i^{max} only grows during the execution of T', we have that $ats(o') \geq ts'' \geq ats(o)$. If o' is a *write*, then $ats(o')$ is determined when line 72 executes. Since this occurs only after *ReadInView*(w') completes, ts_i^{max} is already at least as high as ts''. Then, line 72 sets $ats(o')$ to be $(ts_i^{max}.num+1, i)$ and therefore $ats(o') > ts'' \geq ats(o)$, which completes the proof. □

Theorem 20. *Every history σ corresponding to an execution of DynaStore is linearizable.*

Proof. We create σ' from σ_{RW} by completing operations of the form *write*(v) where v is returned by some complete *read* operation in σ_{RW}. By Lemma 18 parts (a) and (b), each operation which is now complete in σ' has an associated timestamp. We next construct π by ordering all complete *read* and *write* operations in σ' according to their associated timestamps, such that a *write* with some associated timestamp ts appears before all *reads* with the same associated timestamp, and reads with the same associated timestamp are ordered by their invocation times. Lemma 18 part (c) implies that all *write* operations in π can be totally ordered according to their associated timestamps.

First, we show that π preserves real-time order. Consider two complete operations o and o' in σ' s.t. o' is invoked after o completes. By Lemma 19, $ats(o') \geq ats(o)$. If $ats(o') > ats(o)$ then o' appears after o in π by construction. Otherwise $ats(o') = ats(o)$ and by Lemma 19 this means that o' is a *read* operation. If o is a *write* operation, then it appears before o' since we placed each *write* before all *reads* having the same associated timestamp. Finally, if o is a *read*, then it

appears before o' since we ordered reads having the same associated timestamps according to their invocation times.

To prove that π preserves the sequential specification of a MWMR register we must show that a *read* always returns the value written by the closest *write* which appears before it in π, or the initial value of the register if there is no preceding write in π. Let o_r be a *read* operation returning a value v. If $v = \perp$ then since v_i^{max} and ts_i^{max} are always assigned atomically together in lines 41 and 72, we have that $ats(o_r) = (0, \perp)$, in which case o_r is ordered before any *write* in π by Lemma 18 part (c). Otherwise, $v \neq \perp$ and by part (b) of Lemma 18 there exists a *write*(v) operation, which has the same associated timestamp, $ats(o_r)$. In this case, this *write* is placed in π before o_r, by construction. By part (c) of Lemma 18, other *write* operations in π have a different associated timestamp and thus appear in π either before *write*(v) or after o_r. □

3.6.3 Liveness

Recall that all active processes take infinitely many steps. As explained in Section 2, termination has to be guaranteed only when certain conditions hold. Thus, in our proof we make the following assumptions:

A1 At any time t, fewer than $|V(t).\textit{members}|/2$ processes out of $V(t).\textit{members} \cup P(t).join$ are in $F(t) \cup P(t).\textit{remove}$.

A2 The number of different changes proposed in the execution is finite.

Lemma 21. *Let ω be any change s.t. $\omega \in$ desiredView at time t. Then a reconfig(c) operation was invoked before t such that $\omega \in c$.*

Proof. If $\omega \in$ *Init*, the lemma follows from our assumption that a *reconfig*(*Init*) completes by time 0. In the remainder of the proof we assume that $\omega \notin$ *Init*. Let T' be a traversal that adds ω to its *desiredView* at time t' s.t. t' is the earliest time when $\omega \in$ *desiredView* for any traversal in the execution. Thus, $t' \leq t$. Suppose for the purpose of contradiction that ω is added to *desiredView* in line 60 during T'. Then $\omega \in c$, s.t. c is in the set returned by a *scan* in line 67. By property NV1, an *update* completes before this time with c as parameter. By line 55, $\omega \in$ *desiredView* at the traversal that executes the *update*, which contradicts our choice of T' as the first traversal that includes ω in *desiredView*. The remaining option is that ω is added to *desiredView* in line 48 during T'. Since no

traversal includes ω in *desiredView* before t', and since $\omega \notin$ *Init*, we conclude that $\omega \notin$ *curView*$_i$. Thus, $\omega \in cng$. This means that T' is executed during a *reconfig*(c) operation invoked before time t, such that $\omega \in c$, which is what we needed to show. □

Lemma 22. *(a) If w is an established view, then for every change $\omega \in w$, a reconfig(c) operation is invoked in the execution s.t. $\omega \in c$; (b) If w is a view s.t. $w \in$ Front at time t then for every change $\omega \in w$, a reconfig(c) operation is invoked before t such that $\omega \in c$.*

Proof. We prove the claim by induction on the position of w in $\mathcal{E}$. If $w =$ *Init*, then the claim holds by our assumption that a *reconfig*(*Init*) completes by time 0. Assume that the claim holds until some position $k \geq 0$ in $\mathcal{E}$. Let w be the k-th view in $\mathcal{E}$ and observe w', the $k+1$-th established view. By definition of $\mathcal{E}$, there exists a set of changes c such that $w' = w \cup c$, where c was returned by some *scan*(w) operation in the execution. By property NV1, some *update*(w, c) operation is invoked. By line 55, $c \subseteq$ *desiredView* at the traversal that executes the *update*. (a) then follows from Lemma 21. (b) follows from Lemma 21 since by Lemma 5 we have that $w \subseteq$ *desiredView* and therefore $\omega \in$ *desiredView* at time t. □

Corollary 23. *The sequence of established views $\mathcal{E}$ is finite.*

Proof. By Lemma 22, established views contain only changes proposed in the execution. Since all views in $\mathcal{E}$ are totally ordered by the "$\subset$" relation, and by assumption A2, $\mathcal{E}$ is finite. □

Definition 5. We define t_{fix} to be any time s.t. $\forall t \geq t_{fix}$ the following conditions hold:

1. $V(t) = V(t_{fix})$
2. $P(t) = P(t_{fix})$
3. $(V(t).\textit{join} \cup P(t).\textit{join}) \cap F(t) = (V(t_{fix}).\textit{join} \cup P(t_{fix}).\textit{join}) \cap F(t_{fix})$
   (i.e., all processes in the system that crash in the execution have already crashed by t_{fix}).

The next lemma proves that t_{fix} is well-defined.

Lemma 24. *There exists t_{fix} as required by Definition 5.*

Proof. $V(t)$ contains only changes that were proposed in the execution (for which there is a reconfiguration proposing them that completes). Since no element can leave $V(t)$ once it is in this

set, $V(t)$ only grows during the execution, and from assumption A2 there exists a time t_v starting from which $V(t)$ does not change. No *reconfig* operation proposing a change $\omega \notin V(t)$ can complete from t_v onward, and therefore no element leaves the set P from that time and P can only grow. From assumption A2 there exists a time t_p starting from which $P(t)$ does not change. Thus, from time $t_{vp} = max(t_v, t_p)$ onward, V and P do not change. By assumption A2, $V(t_{vp})$.*join* $\cup\, P(t_{vp})$.*join* is a finite set of processes. Thus, we can take t_{fix} to be any time after t_{vp} s.t. all processes from this set that crash in the execution have already crashed by t_{fix}. □

Recall that an active process is one that did not fail in the execution, whose Add was proposed and whose Remove was never proposed.

Lemma 25. *If w is a view in Front s.t. $V(t_{fix}) \subseteq w$, then at least a majority of w.members are active.*

Proof. By Lemma 22, all changes in w were proposed in the execution. Since all changes proposed in the execution are proposed by time t_{fix}, $w \subseteq V(t_{fix}) \cup P(t_{fix})$. Denote the set of changes $w \backslash V(t_{fix})$ by AC. Notice that $AC \subseteq P(t_{fix})$. Each element in AC either adds or removes one process. Observe the set of members in w, and let us build this set starting with $M = V(t_{fix})$.*members* and see how this set changes as we add elements from AC. First, consider changes of the form $(+, j)$ in AC. Each change of this form adds a member to M, unless $j \in V(t_{fix})$.*remove*, in which case it has no effect on M. A change of the form $(-, k)$ removes p_k from M. According to this, we can write *w.members* as follows: *w.members* $= (V(t_{fix})$.*members* $\cup\, J_w) \setminus R_w$, where $J_w \subseteq P(t_{fix})$.*join* $\setminus\, V(t_{fix})$.*remove* and $R_w \subseteq P(t_{fix})$.*remove*. We denote $V(t_{fix})$.*members* $\cup\, J_w$ by L and we will show that a majority of L is active. Since R_w contains only processes that are not active, when removing them from L (in order to get *w.members*), it is still the case that a majority of the remaining processes are active, which proves the lemma.

We next prove that a majority of L are active. By definition of t_{fix}, all processes proposed for removal in the execution have been proposed by time t_{fix}. Notice that no process in $V(t_{fix})$.*members*$\cup$ J_w is also in $V(t_{fix})$.*remove* by definition of this set, and thus, if the removal of a process in L was proposed by time t_{fix}, this process is in $P(t_{fix})$.*remove*. Since $L \subseteq V(t_{fix})$.*join* $\cup\, P(t_{fix})$.*join*, by definition of t_{fix} every process in L that crashes in the execution does so by time t_{fix}. Thus, $F(t_{fix}) \cup P(t_{fix})$.*remove* includes all processes in L that are not active. Assumption A1 says that fewer than $|V(t_{fix})$.*members*$|/2$ out of $V(t_{fix})$.*members*$\cup P(t_{fix})$.*join* are in $F(t_{fix}) \cup P(t_{fix})$.*remove*.

Thus, fewer than $|V(t_{fix})$.*members*$|/2$ out of $V(t_{fix})$.*members* $\cup J_w$, which equals to L, are in $F(t_{fix}) \cup P(t_{fix})$.*remove*. This means that a majority of the processes in L are active. □

Lemma 26. *Let p_i be an active process and w be an established view s.t. $i \in w$.members. Then $i \in w'$.members for every established view w' s.t. $w \dot{\leq} w'$.*

Proof. Since $w \subseteq w'$ and $i \in w$.*members*, we have that $(+, i) \in w'$. Since p_i is active, no *reconfig*(c) is invoked s.t. $(-, i) \in c$, and by Lemma 22 we have that $(-, i) \notin w'$. Thus, $i \in w'$.*members*. □

Lemma 27. *If p_i and p_j are active processes and p_i sends a message to p_j during the execution of DynaStore, then p_j eventually receives this message.*

Proof. Recall that the link between p_i and p_j is reliable. Since p_i and p_j are active, it remains to show that if the message is sent at time t then $j \in V(t)$.*join*$\cup P(t)$.*join*. Note that p_i sends messages only to processes in w.*members*, where w is a view in Front during Traverse, and therefore $(+, j) \in w$ at time t. By Lemma 22, a *reconfig*(c) was invoked before time t s.t. $(+, j) \in c$. If such operation completes by time t, then $j \in V(t)$.*join*, and otherwise $j \in P(t)$.*join*. □

Lemma 28. *If a reconfig operation o completes in which Traverse returns the view w, then every active process p_j s.t. $j \in w$.members eventually receives a message $\langle$NOTIFY, $\tilde{w}\rangle$ where $w \dot{\leq} \tilde{w}$.*

Proof. Since o completes, there is at least one complete *reconfig* operation in the execution. Let w_{max} be a view returned by a *Traverse* during some complete *reconfig* operation, such that no *reconfig* operation completes in the execution during which *Traverse* returns a view w' where $w_{max} \dot{<} w'$. w_{max} is well defined since every view returned from *Traverse* is established (Lemma 11), and $\mathcal{E}$ is finite by Corollary 23. Notice that $w \dot{\leq} w_{max}$. We next prove that $V(t_{fix}) \subseteq w_{max}$. Suppose for the purpose of contradiction that there exists a change $\omega \in V(t_{fix}) \setminus w_{max}$. Since $\omega \in V(t_{fix})$, a *reconfig*(c) operation completes where $\omega \in c$. By Lemma 10, Traverse during this operation returns a view w' containing ω. By Lemma 11 w' is established, and recall that all established views are totally ordered by the "$\subset$" relation. Since $\omega \in w' \setminus w_{max}$ it must be that $w_{max} \dot{<} w'$. This contradicts the definition of w_{max}. We have shown that $V(t_{fix}) \subseteq w_{max}$, which implies that a majority of w_{max} are active, by Lemma 25.

Since a *reconfig* operation completes where Traverse returns w_{max}, a $\langle$NOTIFY, $w_{max}\rangle$ message is sent in line 29, and it is received by a majority of w_{max}.*members*. Each process receiving

this message forwards it in line 77. Since a majority of w_{max} are active, and every two majority sets intersect, one of the processes that forwards this message is active. By Lemma 26, since $w \dot{\leq} w_{max}$, every active process p_j s.t. $j \in$ *w.members* is also in w_{max}*.members*. By Lemma 27, every such p_j eventually receives this message. □

Lemma 29. *Consider an operation executed by an active process p_i that invokes Traverse at time t_0 starting from curView$_i$ = initView. If no $\langle$*NOTIFY*, $newView\rangle$ messages are received by p_i from time t_0 onward s.t. initView $\subset$ newView then Traverse eventually returns and the operation completes.*

Proof. Since operations are enabled at p_i only once $i \in$ *curView$_i$.join* (lines 11 and 81) and *curView$_i$* only grows during the execution, $i \in$ *initView.join*. By Lemma 7, for every view w which appears in *Front* during the traversal it holds that *initView* $\subseteq w$ and therefore $i \in$ *w.join*. Since p_i is active, no *reconfig*(c) is invoked such that $(-, i) \in c$. By Lemma 22 we have that $(-, i) \notin w$ and therefore $i \in$ *w.members*. This means that p_i does not halt in line 53, and by Lemma 27 p_i receives every message sent to it by active processes in w.

Let w be any view that appears in *Front* during the execution of *Traverse*. Notice that w is not necessarily established, however we show that $V(t_{fix}) \subseteq w$. Suppose for the purpose of contradiction that there exists $\omega \in V(t_{fix}) \setminus w$. Since *initView* $\subseteq w$, $\omega \in V(t_{fix}) \setminus$ *initView*. Since $\omega \in V(t_{fix})$, a *reconfig*(c) operation completes where $\omega \in c$, and by Lemma 10 this operation returns a view w' s.t. $\omega \in w'$. By Lemma 11 both *initView* and w' are established, and since $\omega \in w' \setminus$ *initView*, we get that *initView* $\dot{<} w'$. Since $i \in$ *initView.members* and p_i is active, by Lemma 26 we have that $i \in$ *w'.members*. By Lemma 28, a $\langle$NOTIFY, $w''\rangle$ message where $w' \dot{\leq} w''$ is eventually received by p_i. Since *initView* $\dot{<} w''$, this contradicts the assumption of our lemma.

We have shown that $V(t_{fix}) \subseteq w$, and from Lemma 25 there exists an active majority Q of *w.members*. By Lemma 27, all messages sent by p_i to *w.members* are eventually received by every process in Q, and every message sent to p_i by a process in Q is eventually received by p_i. Thus all invocations of *ContactQ*$(*,$ *w.members*$)$, which involves communicating with a majority of *w.members*, eventually complete, and so do invocations of *scan$_i$* and *update$_i$* by property NV5. Given that all such procedures complete during a *Traverse* and it is not restarted (this follows from the statement of the lemma since no NOTIFY messages that can restart *Traverse* are received at p_i starting from t_0), it is left to prove that the termination condition in line 63 eventually holds.

After Traverse completes, *NotifyQ*(w) is invoked where w is a view returned from Traverse. By Lemma 9, *Front* $= \{w\}$ when Traverse returns, and therefore *NotifyQ*(w) completes as well since there is an active majority in w.*members*, as explained above.

By assumption A2 and Lemma 22, the number of different views added to *Front* in the execution is finite. Suppose for the purpose of contradiction that *Traverse* does not terminate and consider iteration k of the loop starting from which views are not added to *Front* unless they have been already added before the k-th iteration (notice that by Lemma 8, when a view is removed from *Front*, it can never be added again to *Front*; thus, from iteration k onward views can only be removed from Front and the additions have no affect in the sense that they can add views that are already present in Front but not new views or views that have been removed from Front). We first show that in some iteration $k' \geq k$, $|$*Front*$| = 1$. Consider any iteration where $|$*Front*$| > 1$, and let w be the view chosen from Front in line 52 in this iteration. By Lemma 5, in this case $w \neq$ *desiredView*, as *desiredView* contains the changes of all views in Front, and $|$*Front*$| > 1$ means that there is at least one view in Front which contains changes that are not in w. Then, line 55 executes, and by Lemma 2, *ReadInView* returns a non-empty set. Next, the condition in line 57 evaluates to true and w is removed from Front in line 58. Since no new additions are made to Front starting with the k-th iteration (i.e., only a view that is already in Front can be added in line 61), the number of views in Front decreases by 1 in this iteration. Thus, there exists an iteration $k' \geq k$ where only a single view remains in Front.

Observe iteration k', where $|$*Front*$| = 1$, and let w be the view chosen from Front in line 52 in this iteration. Suppose for the purpose of contradiction that the condition on line 57 evaluates to true. Then, w is removed from *Front*, and the loop on lines 59–61 executes at least once, adding views to *Front*. By Lemma 8, the size of these views is bigger than w, and therefore every such view is different than w, contradicting the fact that starting from iteration k only views that are already in *Front* can be added to *Front* (recall that $k' \geq k$). Thus, starting from iteration k' the condition on line 57 evaluates to false, and *WriteInView* is invoked in iteration k'. Assume for the sake of contradiction that *WriteInView* does not return $\emptyset$. In this case, the loop would continue and w (the only view in Front) is chosen again from Front in iteration $k' + 1$. Then, *ReadInView*(w) returns a non-empty set by Lemma 3 and the condition in line 57 evaluates to *true*, which cannot happen, as explained above. Thus, in iteration k', the condition in line 57 evaluates to false, *WriteInView*$(w, *)$ returns $\emptyset$, and the loop terminates. □

Theorem 30. *DynaStore preserves Dynamic Service Liveness (Definition 2). Specifically, (a) Eventually, the* enable operations *event occurs at every active process that was added by a complete reconfig operation, and (b) Every operation o invoked by an active process p_i eventually completes.*

Proof. (a) Let p_i be an active process that is added to the system by a complete *reconfig* operation. If $i \in$ *Init.join* then the operations at p_i are enabled from the time it starts taking steps (line 11). Otherwise, a *reconfig* adding p_i completes, and let w be the view returned by Traverse during this operation. By Lemma 10, $(+, i) \in w$. Since p_i is active, no *reconfig*(c) operation is invoked s.t. $(-, i) \in c$. By Lemma 22 we get that $(-, i) \notin w$, which means that $i \in$ *w.members*. By Lemma 28, p_i eventually receives a $\langle$NOTIFY, $w'\rangle$ message such that $w \stackrel{\cdot}{\leq} w'$. By Lemma 26, $(+, i) \in w'$, i.e., $i \in w'$*.join*. This causes operations at p_i to be enabled in line 81 (if they were not already enabled by that time).

(b) Every operation o invokes Traverse and during its execution, whenever a $\langle$NOTIFY, *newView*$\rangle$ message is received by p_i s.t. *curView*$_i \subset$ *newView*, *curView*$_i$ becomes *newView* in line 80, and *Traverse* is restarted. By Corollary 23, $\mathcal{E}$ is finite. By Lemma 11, only established views are sent in NOTIFY messages. Thus, the number of times a Traverse can be restarted is finite and at some point in the execution, no more $\langle$NOTIFY, *newView*$\rangle$ messages can be received s.t. *curView*$_i \subset$ *newView*. By Lemma 29, Traverse eventually returns and the operation completes. □

Chapter 4

Untrusted Storage

Many providers now offer a wide variety of flexible online data storage services, ranging from passive ones, such as online archiving, to active ones, such as collaboration and social networking. They have become known as computing and storage "clouds." Such clouds allow users to abandon local storage and use online alternatives, such as Amazon S3, Nirvanix CloudNAS, or Microsoft SkyDrive. Some cloud providers utilize the fact that online storage can be accessed from any location connected to the Internet, and offer additional functionality; for example, Apple MobileMe allows users to synchronize common applications that run on multiples devices. Clouds also offer computation resources, such as Amazon EC2, which can significantly reduce the cost of maintaining such resources locally. Finally, online collaboration tools, such as Google Apps or versioning repositories for source code, make it easy to collaborate with colleagues across organizations and countries.

The remainder of this thesis deals with tools and semantics enabling clients that use online cloud services to monitor or audit them, making sure that the cloud behaves as expected. In this chapter we motivate this study of untrusted storage and define a system model used in the following chapters. Sections 4.1 and 4.2 are based on a paper published in the ACM SIGACT News [14]. A preliminary version of the material in Section 4.3 appeared in 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) [12].

4.1 What Can Go Wrong?

Although the advantages of using clouds are unarguable, there are many risks involved with releasing control over your data. One concern that many users are aware of is loss of privacy. Nevertheless, the popularity of social networks and online data sharing repositories suggests that many users are willing to forfeit privacy, at least to some extent. Setting privacy aside, we now briefly survey what else "can go wrong" when your data is stored in a cloud.

Availability is a major concern with any online service, as such services are bound to have some downtime. This was recently the case with Google Mail[1], Hotmail[2], Amazon S3[3] and MobileMe[4]. Users must also understand their service contract with the storage provider. For example, what happens if your payment for the storage is late? Can the storage provider decide that one of your documents violates its policy and terminate your service, denying you access to the data? Even the worst scenarios sometimes come true — a cloud storage-provider named LinkUp (MediaMax) went out of business last year after losing 45% of stored client data due to an error of a system administrator[5]. This incident also revealed that it is sometimes very costly for storage providers to keep storing old client data, and they look for ways to offload this responsibility to a third party. Can a client make sure that his data is safe and available?

No less important is guaranteeing the integrity of remotely stored data. One risk is that data can be damaged while in transit to or from the storage provider. Additionally, cloud storage, like any remote service, is exposed to malicious attacks from both outside and inside the provider's organization. For example, the servers of the Red Hat Linux distribution were recently attacked and the intruder managed to introduce a vulnerability and even sign some packages of the Linux operating-system distribution[6]. In its Security Advisory about the incident, Red Hat stated:

> *... we remain highly confident that our systems and processes prevented the intrusion from compromising RHN or the content distributed via RHN and accordingly believe that customers who keep their systems updated using Red Hat Network are not at risk.*

[1] `http://googleblog.blogspot.com/2009/02/current-gmail-outage.html`
[2] `http://www.datacenterknowledge.com/archives/2009/03/12/downtime-for-hotmail`
[3] `http://status.aws.amazon.com/s3-20080720.html`
[4] `http://blogs.zdnet.com/projectfailures/?p=908`
[5] `http://blogs.zdnet.com/projectfailures/?p=999`
[6] `https://rhn.redhat.com/errata/RHSA-2008-0855.html`

Unauthorized access to user data can occur even when no hackers are involved, e.g., resulting from a software malfunction at the provider. Such data breach occurred in Google Docs[7] during March 2009 and led the Electronic Privacy Information Center to petition[8] with the Federal Trade Commission asking to *"open an investigation into Google's Cloud Computing Services, to determine the adequacy of the privacy and security safeguards..."*. Another example, where data integrity was compromised as a result of provider malfunctions, is a recent incident with Amazon S3, where users experienced silent data corruption[9]. Later Amazon stated in response to user complaints[10]:

> *We've isolated this issue to a single load balancer that was brought into service at 10:55pm PDT on Friday, 6/20. It was taken out of service at 11am PDT Sunday, 6/22. While it was in service it handled a small fraction of Amazon S3's total requests in the US. Intermittently, under load, it was corrupting single bytes in the byte stream ... Based on our investigation with both internal and external customers, the small amount of traffic received by this particular load balancer, and the intermittent nature of the above issue on this one load balancer, this appears to have impacted a very small portion of PUTs during this time frame.*

A further complication arises when multiple users collaborate using cloud storage (or simply when one user synchronizes multiple devices). Here, consistency under concurrent access must be guaranteed. A possible solution that comes to mind is using a Byzantine fault-tolerant replication protocol within the cloud (e.g., [34]); indeed this solution can provide perfect consistency and at the same time prevent data corruption caused by some threshold of faulty components within the cloud. However, since it is reasonable to assume that most of the servers belonging to a particular cloud provider run the same system installation and are most likely to be physically located in the same place (or even run on the same machine), such protocols might be inappropriate. Moreover, cloud-storage providers might have other reasons to avoid Byzantine fault-tolerant consensus protocols, as explained by Birman et al. [7]. Finally, even if this solves the problem from the perspective of the storage provider, in this thesis we are more interested in the users' perspective. A user

[7] `http://blogs.wsj.com/digits/2009/03/08/1214/`

[8] `http://cloudstoragestrategy.com/2009/03/trusting-the-cloud-the-ftc-and-google.html`

[9] `http://blogs.sun.com/gbrunett/entry/amazon_s3_silent_data_corruption`

[10] `http://developer.amazonwebservices.com/connect/thread.jspa?threadID=22709`

perceives the cloud as a single trust domain and puts trust in it, whatever the precautions taken by the provider internally might be; in this sense, the cloud is not different from a single remote server. Note that when multiple clouds from different providers are used, running Byzantine-fault-tolerant protocols across several clouds might be appropriate.

4.2 What Can We Do?

Users can locally maintain a small amount of trusted memory and use well-known cryptographic methods in order to significantly reduce the need for trust in the storage cloud. A user can verify the integrity of his remotely stored data by keeping a short hash in local memory and authenticating server responses by re-calculating the hash of the received data and comparing it to the locally stored value. When the volume of data is large, this method is usually implemented using a hash tree [60], where the leaves are hashes of data blocks, and internal nodes are hashes of their children in the tree. A user is then able to verify any data block by storing only the root hash of the tree corresponding to his data [8]. This method requires a logarithmic number of cryptographic operations in the number of blocks, as only one branch of the tree from the root to the hash of an actual data block needs to be checked. Hash trees have been employed in many storage-system prototypes (TDB [54] and SiRiUS [32] are just two examples) and are used commercially in the Solaris ZFS filesystem[11]. Research on efficient cryptographic methods for authenticating data stored on servers is an active area [63, 65].

Although these methods permit a user to verify the integrity of data returned by a server, they do not allow a user to ascertain that the server is able to answer a query correctly without actually *issuing* that particular query. In other words, they do not assure the user that all the data is "still there". As the amount of data stored by the cloud for a client can be enormous, it is impractical (and might also be very costly) to retrieve all the data, if one's purpose is just to make sure that it is stored correctly. In recent work, Juels and Kaliski [39] and Ateniese et al. [4] introduced protocols for assuring a client that his data is retrievable with high probability, under the name of *Proofs of Retrievability* (PORs) and *Proofs of Data Possession* (PDP), respectively. They incur only a small, nearly constant overhead in communication complexity and some computational overhead by the server. The basic idea in such protocols is that additional information is encoded in the data prior to

[11] `http://blogs.sun.com/bonwick/entry/zfs_end_to_end_data`

storing it. To make sure that the server really stores the data, a user submits challenges for a small sample of data blocks, and verifies server responses using the additional information encoded in the data. Recently, some improved schemes have been proposed and prototype systems have been implemented [70, 9, 10].

The above tools allow a single user to verify the integrity and availability of his own data. But when multiple users access the same data, they cannot guarantee integrity between a writer and multiple readers. Digital signatures may be used by a client to verify integrity of data created by others. Using this method, each client needs to sign all his data, as well as to store an authenticated public key of the others or the root certificate of a public-key infrastructure in trusted memory. This method, however, does not rule out all attacks by a faulty or malicious storage service. Even if all data is signed during write operations, the server might omit the latest update when responding to a reader, and even worse, it might "split its brain," hiding updates of different clients from each other. Some solutions use trusted components in the system [21, 78] which allow clients to audit the server, guaranteeing atomicity even if the server is faulty. In Section 5.4 we show that without additional trust assumptions, the atomicity of all operations in the sense of linearizability [37] cannot be guaranteed; in fact, we show that even weaker consistency notions, such as sequential consistency [43], are not possible either. Though a user may become suspicious when he does not see any updates from a collaborator, the user can only be certain that the server is not holding back information by communicating with the collaborator directly; such user-to-user communication is indeed employed in some systems for this purpose.

If not atomicity, then what consistency can be guaranteed to clients? The first to address this problem were Mazières and Shasha [59], who defined a so-called *forking* consistency condition. This condition ensures that if certain clients' perception of the execution becomes different, for example if the server hides a recent value of a completed write from a reader, then these two clients will never again see each other's later operations, or else the server will be exposed as faulty. This prevents a situation where one user sees part of the updates issued by another user, and the server can choose which ones. Moreover, fork-consistency prevents Alice from seeing new updates by Bob and by Carol, while Bob sees only Alice's updates, where Alice and Bob might think they are mutually consistent, though they actually see different states. Essentially, with fork consistency, each client has a linearizable view of a sub-sequence of the execution, and client views can only become disjoint once they diverge from a common prefix; a simple definition can be found in

Section 5.2. The first protocol of this kind, realizing fork-consistent storage, was implemented in the SUNDR system [48].

To save cost and to improve performance, several weaker consistency conditions have been proposed. The notion of fork-sequential-consistency, introduced by Oprea and Reiter [64], allows client views to violate real-time order of the execution. The fork-* consistency condition due to Li and Mazières [49] allows the views of clients to include one more operation without detecting an attack after their views have diverged. This condition was used to provide meaningful service in a Byzantine-fault-tolerant replicated system, even when more than a third of the replicas are faulty [49].

Although consistency in the face of failures is crucial, it is no less important that the service is unaffected in the common case by the precautions taken to defend against a faulty server. In Section 5.5 we show that for all previously existing forking consistency conditions, and thus in the protocols that implement them with a single remote server, concurrent operations by different clients may block each other even if the provider is correct. More formally, these consistency conditions do not allow for protocols that are wait-free [36] when the storage provider is correct. In Section 5.7 we introduce a new consistency notion, called weak fork-linearizability, that does not suffer from this limitation, and yet provides meaningful semantics to clients.

One disadvantage of forking consistency conditions is that they are not as intuitive to understand as atomicity, for example. Aiming to provide simpler guarantees, we introduce the notion of a Fail-Aware Untrusted Service in Chapter 6. Its basic idea is that each user should know which of his operations are seen consistently by each of the other users, and in addition, find out whenever the server violates atomicity. When all goes well, each operation of a user eventually becomes "stable" with respect to every other correct user, in the sense that they have a common view of the execution up to this operation. Thus, in all cases, users get either positive notifications indicating operation stability, or negative notifications when the server violates atomicity. Our Fail-Aware Untrusted Services rely on the well-established notions of eventual consistency [74] and fail-awareness [29], and adapt them to this setting. The FAUST protocol [12], presented in Chapter 6, implements this notion for a storage service, using an underlying weak fork-linearizable storage protocol. Intuitively, FAUST indicates stability as soon as additional information is gathered, either through the storage protocol, or whenever the clients communicate directly. However, all complete operations, even those not yet known to be stable, preserve causality [38]. Moreover, when the storage server

is correct, FAUST guarantees strong safety (linearizability) and liveness (wait-freedom).

Although FAUST is an important step towards providing tools and semantics for secure interaction with untrusted cloud storage, several aspects in FAUST limit its usability in practice. The stability notion in FAUST is not transitive and requires users to explicitly track the other clients in the system and to assess their relation to the data accessed by the operation. FAUST is therefore not easily amenable to dynamic changes in the set of clients. Furthermore, global consistency in FAUST (among all clients) is guaranteed only if no client ever crashes. FAUST does not work with commodity storage – like other proposals it integrates storage operations with the consistency mechanism and moreover it does not allow multiple clients to modify the same object, which is the usual semantics of commodity storage services.

These shortcomings led to the development of Venus [71], a system presented in Chapter 7. In Venus, stability indications simply specify the last operation of the client that has been verified to be globally consistent, which is easy to integrate with an application. Venus eliminates the need for clients to track one another, and enables dynamic client changes. Unlike the previous protocols, Venus allows all clients to modify the same shared object. Most importantly, the design of Venus is modular, so that it can be deployed with a commodity storage service. We deployed and evaluated Venus with the Amazon S3 cloud storage service, demonstrating its usefulness in practice.

4.3 System Model

This section formally defines the system model we use in the following chapters to represent untrusted storage remotely accessed by clients.

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. Clients do not communicate with each other (we relax this restriction in later chapters to allow infrequent offline communication among clients). The clients and the server are collectively called *parties*. System components are modeled as deterministic I/O Automata [51]. 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*.

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*.

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 sub-sequence 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 sub-sequence 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 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 [51]).

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 thesis 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 $\perp \notin \mathcal{X}$. When a client C_i invokes a read operation, the register

responds with a value x, denoted $\mathit{read}_i(X) \to x$; when C_i invokes a write operation with value x, denoted $\mathit{write}_i(X, x)$, 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 considered in Chapter 6 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 $j = 1, \ldots, n$. In Chapter 7 we consider a single multi-writer/multi-reader (MWMR) register, which all clients can read and write.

Cryptographic primitives. The protocols in this thesis 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*.

Chapter 5

Consistency Semantics for Untrusted Storage

This chapter defines consistency semantics for the untrusted storage model, and studies their properties. We start by re-stating in Section 5.1 some well-known consistency and liveness properties used to characterize distributed shared memory. Section 5.2 surveys known "forking" consistency conditions. Even though these conditions were previously defined by others, we consider the formal statement of these semantics to be a contribution of this work. Section 5.3 introduces the notion of Byzantine emulation, i.e., the emulation of shared memory using untrusted storage. Section 5.4 motivates the need for forking semantics showing that traditional semantics cannot be guaranteed when the server is faulty. Section 5.5 shows a limitation inherent in all previously known forking consistency conditions, namely that they hamper service availability in the common case, when the storage provider is correct. This motivates the need to weaken forking conditions even further. Section 5.6 compares forking conditions with causal consistency, showing that some variations of forking consistency are too weak, in the sense that they may violate causality. Finally, in Section 5.7 we introduce a new notion of weak fork-linearizability which on the one hand does not affect availability when the provider is correct, and on the other hand it is strong enough and in particular implies causality. This notion underlies our algorithms presented in the following chapters. Parts of this chapter appeared in [12], [13] and [15].

5.1 Traditional Consistency and Liveness Properties

Our definitions rely on the notion of a possible *view* of a client, defined as follows.

Definition 6 (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 sub-sequence of *complete*(σ');
2. $\pi|_{C_i} =$ *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, w.r.t. F.

One of the most important consistency conditions for concurrent access is sequential consistency [43].

Definition 7 (Sequential consistency [43]). A history σ is *sequentially consistent* w.r.t. a functionality F if there exists a sequence of events π that is a view of σ w.r.t. F at all clients.

Intuitively, sequential consistency requires that every operation takes effect at some point and occurs somewhere in the permutation π. This guarantees that every write operation is eventually seen by all clients. In other words, if an operation writes v to a register X, there cannot be an infinite number of subsequent read operations from register X that return a value written to X prior to v.

A stronger consistency condition is linearizability [37]. Whereas sequential consistency preserves the real-time order only for operations by the same client, linearizability guarantees that real-time order is preserved for all operations.

Definition 8 (Linearizability [37]). 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 [38] 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* [42]. 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 [38].

Definition 9 (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* [36].

Definition 10 (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.

5.2 Forking Consistency Conditions

The notion of fork-linearizability [59] (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 11 (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 $\pi_i|^o = \pi_j|^o$.

Oprea and Reiter [64] define *fork-sequential-consistency* by replacing the real-time order condition of fork-linearizability with:

2. (*local-real-time-order*) For every client C_j, the sequence $\pi_i|_{C_j}$ preserves the real-time order of σ.

Because preservation of real-time order is required only for a subset of every view, fork-sequential-consistency is weaker than fork-linearizability. Note that the local-real-time-order condition is weaker than the third condition of causal consistency, since all operations of each client are causally ordered. Oprea and Reiter [64] do not give an emulation protocol for this condition, and indeed, we show (in Section 5.5) that no such protocol is wait-free.

Li and Mazières [49] relax the notion for fork-linearizability differently, 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.

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 permits wait-free client operations at the same time, as we show in Section 5.5.

5.3 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* (defined later in this chapter).

Definition 12 (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.

Similarly, we define a fork-sequentially-consistent Byzantine emulation. It should guarantee sequential consistency when the server is correct, and fork sequential consistency otherwise.

Definition 13 (fork-sequentially-consistent Byzantine emulation). A protocol P *emulates* a functionality F *on a Byzantine server* S *with fork-sequential-consistency* whenever the following conditions hold:

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

5.4 Impossibility of Linearizability and Sequential Consistency with an Untrusted Server

This section explains why neither linearizability nor sequential consistency can be guaranteed to clients when F is implemented on a Byzantine server (at least not for functionalities F where some operations do not commute), which motivates the need for considering weaker (e.g., forking) semantics. To see why linearizability is impossible suppose that C_i was the last client to execute an operation on F; no matter what protocol the clients use to interact with the server, a faulty server might roll back its internal memory to the point in time before executing the operation on behalf of C_i, and pretend to a client C_j that C_i's operation did not occur. As long as C_j and C_i do not communicate with each other, neither party can detect this violation and thus linearizability cannot be satisfied.

The example above does not rule out that S may emulate a sequentially consistent register. Sequential consistency does not have to preserve the real-time order of operations, thus not showing C_i's last update to C_j does not violate sequential consistency. It would be acceptable for a correct server to return old register values, as long as it preserves the relative order in which it shows them to every client. However, we show in the following theorem that a faulty server may also violate sequential consistency when it emulates more than one register:

Theorem 31. *There is no protocol that emulates $n > 1$ SWMR registers on a Byzantine server with sequential consistency.*

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

The execution consists of four operations by the clients C_1 and C_2. Client C_1 executes $\mathit{write}_1(X_1, v) \rightarrow \text{OK}$ and $\mathit{read}_1(X_2) \rightarrow \perp$. The server interacts with C_1 as if it was the only client executing any operation. Concurrently, C_2 executes $\mathit{write}_2(X_2, v) \rightarrow \text{OK}$ and $\mathit{read}_2(X_1) \rightarrow \perp$ and S also pretends to C_2 that it is the only client executing any operation. Such "split-brain" behavior is obviously possible when S is faulty: it can act as if the write operations to X_1 and X_2 have completed, as far as the writing client is concerned, but still return the old values of X_1 and X_2 in the read operations. Since the only interaction of the clients is with S, neither client can distinguish execution λ from a sequentially consistent execution where it executes alone.

Notice that λ is not sequentially consistent: There is no permutation of the operations in λ in which the sequential specification of both X_1 and X_2 is preserved and, at the same time, the order of operations occurring at each client is the same as their real-time order in λ. Specifically, in any possible permutation of λ, the operation $\textit{read}_1(X_2) \rightarrow \perp$ cannot be positioned after $\textit{write}_2(X_2, v)$, since the read would have to return $v \neq \perp$ according to the sequential specification of X_2. However, $\textit{read}_1(X_2) \rightarrow \perp$ cannot occur before $\textit{write}_2(X_2, v)$ as we now argue. Since the local order of operations has to be the same as in λ in this case, $\textit{write}_1(X_1, u)$ must occur before $\textit{read}_1(X_2) \rightarrow \perp$ and hence also before $\textit{write}_2(X_2, v)$. But since the latter operation precedes $\textit{read}_2(X_1) \rightarrow \perp$ in the local order seen by C_2, we conclude that $\textit{write}_1(X_1, u)$ precedes $\textit{read}_2(X_1) \rightarrow \perp$, which contradicts the sequential specification of X_1. Thus, λ is not sequentially consistent. □

Note that execution λ constructed in the proof above is fork-linearizable but not sequentially consistent. On the other hand, execution γ exhibited in the proof of Theorem 33 below and shown in Figure 5.3 is sequentially consistent but not fork-linearizable. Hence, we obtain the following result.

Corollary 32. *Fork-linearizability is neither stronger nor weaker than sequential consistency.*

5.5 Limited Service Availability with Forking Semantics

We have shown that it is impossible to guarantee traditional strong semantics such as linearizability and sequential consistency with an untrusted server. In contrast, emulations of shared memory with forking semantics are possible. Such semantics provide well-defined guarantees to clients even when the server is faulty. However, we show in this section that many of these semantics have an inherent limitation – they hamper service availability in the common case, i.e., when the server is correct.

We start by proving that fork-*-linearizable Byzantine emulations cannot be wait-free in all executions where the server is correct. We then continue to prove that fork sequential consistency suffers from the same limitation. Both results separately imply the corresponding impossibility for fork-linearizable Byzantine emulations, which appeared [15].

Theorem 33. *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.

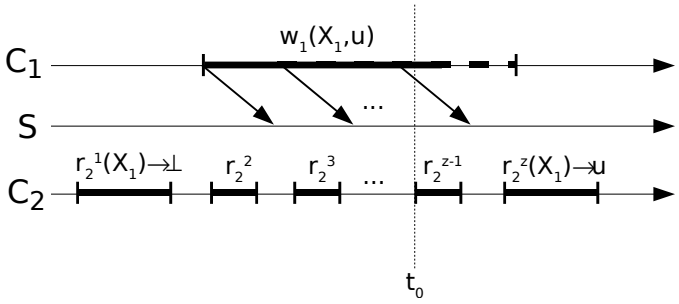

Figure 5.1: Execution α: S is correct.

Execution α. We construct an execution α, shown in Figure 5.1, in which S is correct. Client C_1 executes a write operation $write_1(X_1, u)$ and C_2 executes multiple read operations from X_1, denoted r_2^i for $i = 1, \ldots, z$, 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 $w_1 = write_1(X_1, u)$. 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 5.1-5.3.

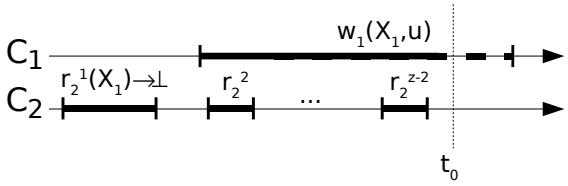

Figure 5.2: Execution β: S is correct.

Execution β. We next define execution β, in which S is also correct. The execution is shown in Figure 5.2. 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.

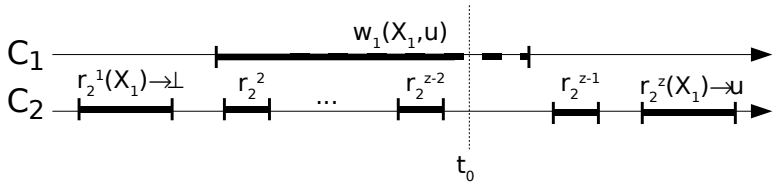

Figure 5.3: Execution γ: S is faulty. It is indistinguishable from α to C_2 and indistinguishable from β to C_1.

Execution γ. Our final execution is γ, shown in Figure 5.3, 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. □

The next theorem shows that fork-sequential-consistency has the same inherent limitation as fork-*-linearizability and fork-linearizability.

Theorem 34. *There is no wait-free fork-sequentially-consistent Byzantine emulation of $n \geq 2$ SWMR registers on a Byzantine server S.*

Proof. Towards a contradiction assume that there exists such a protocol P. Then in any admissible execution of P with a correct server, every operation of a correct client completes. We next construct three executions α, β, and γ of P, shown in Figures 5.4–5.6. All three executions are admissible, since clients issue operations sequentially, and every message sent between two correct parties is eventually delivered. There are two clients C_1 and C_2, which are always correct, and access two SWMR registers X_1 and X_2. Protocol P describes the asynchronous interaction of the clients with S; this interaction is depicted in the figures only when necessary.

Execution α. In execution α, the server is correct. The execution is shown in Figure 5.4 and begins with four operations by C_2: first C_2 executes a write operation with value v_1 to register X_2, denoted w_2^1, then an operation reading register X_1, denoted r_2^1, then an operation writing v_2 to X_2, denoted w_2^2, and finally again a read operation of X_1, denoted r_2^2. Since S and C_2 are correct and P is wait-free with a correct server, all operations of C_2 eventually complete.

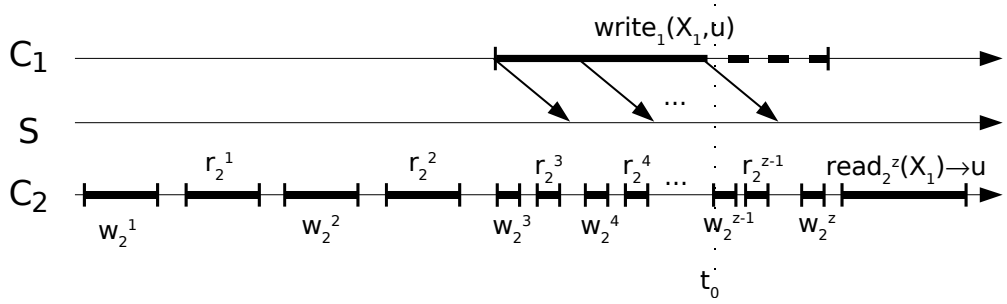

Figure 5.4: Execution α, where S is correct.

Execution α continues as follows. C_1 starts to execute a single write operation with value u to X_1, denoted w_1. Every time a message is sent from C_1 to S during this operation, and as long as no read operation by C_2 from X_1 returns a value different from $\perp$, the following steps are repeated in order, for $i = 3, 4, \ldots$:

(a) The message from C_1 is delayed by the asynchronous network;

(b) C_2 executes an operation writing v_i to X_2, denoted w_2^i;

(c) C_2 executes an operation reading X_1, denoted r_2^i; and

(d) the delayed message from C_1 is delivered to S.

Note that w_2^i and r_2^i complete by the assumptions that P is wait-free and that S is correct. For the same reason, operation w_1 eventually completes. After w_1 completes, and while C_2 does not read any non-$\perp$ value from X_1, C_2 continues to execute alternating operations w_2^i and r_2^i, writing v_i to X_2 and reading X_1, respectively. This continues until some read returns a non-$\perp$ value. Because S is correct, eventually some read of X_1 is guaranteed to return $u \neq \perp$ by sequential consistency of the execution. We denote the first such read by r_2^z. This is the last operation of C_2 in α. If messages are sent from C_1 to S after the completion of r_2^z, they are not delayed.

Note that the prefix of α up to the completion of r_2^3 is indistinguishable to C_2 and S from an execution in which no client writes to X_1, and therefore r_2^1, r_2^2, and r_2^3 return the initial value $\perp$. Hence, $z \geq 4$.

We denote the point of invocation of w_2^{z-1} in α by t_0. It is marked by a dotted line. Executions β and γ constructed below are identical to α before t_0, but differ from α starting at t_0.

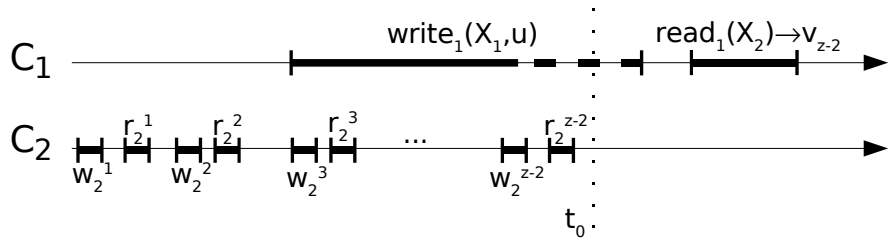

Figure 5.5: Execution β, where S is correct.

Execution β. We next define execution β, shown in Figure 5.5, in which the server is also correct. Execution β is identical to α up to the end of r_2^{z-2} (before t_0), but then C_2 halts. In other words, the last two write-read pairs of C_2 in α are missing in β. Operation w_1 is invoked in β like in α and begins after the completion of r_2^2 (notice that r_2^2 is in β since $z \geq 4$). Because the protocol is wait-free with the correct server, operation w_1 completes. Afterwards, C_1 invokes a read of X_2, denoted by r_1, which also eventually completes. Since the server is correct, β is sequentially consistent. Observe a sequential permutation π guaranteed by sequential consistency. Since r_2^{z-2} returns $\perp$, w_1 appears in π after r_2^{z-2}. Since π preserves the order of C_1's operations in β, r_1 appears after w_1 in π, and since the order of C_2's operations is also preserved, this means that r_1 appears after w_2^{z-2} in π, and therefore returns v^{z-2}.

Execution γ. The third execution γ is shown in Figure 5.6; here, the server is faulty. Execution γ proceeds just like the common prefix of α and β before t_0, and client C_1 invokes w_1 in the same way as in α and in β. From t_0 onward, the server simulates β to C_1. This is easy because S simply hides from C_1 all operations of C_2 starting with w_2^{z-1}. The server also simulates α to 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 C_1 sends this message before t_0 by construction of α. If

such a message exists in α, it also exists in γ because γ is identical to α before t_0. Therefore, the server has all of the information needed to simulate α to C_2 and r_2^z returns u.

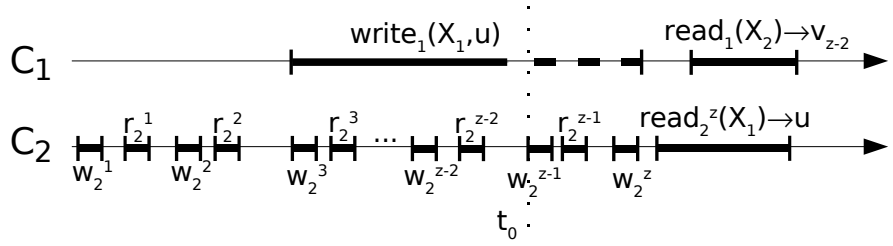

Figure 5.6: Execution γ, where S is faulty and simulates α to C_2 and β to C_1.

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-sequentially-consistent. Consider the sequential permutation π_2 required by the definition of fork sequential consistency, i.e., the view of C_2. As the real-time order of C_2's operations and the sequential specification of the registers must be preserved in π_2, and since r_2^1, ..., r_2^{z-1} return $\perp$ but r_2^z returns u, we conclude that w_1 must appear in π_2 and is located after r_2^{z-1} but before r_2^z. Because w_1 is one of C_1's operations, it also appears in π_1. By the no-join property, the sequence of operations preceding w_1 in π_2 must be the same as the sequence preceding w_1 in π_1. In particular, w_2^{z-1} and w_2^{z-2} appear in π_1 before w_1, and w_2^{z-2} precedes w_2^{z-1}. Since the real-time order of C_1's operations must be preserved in π_1, operation w_1 and, hence, also w_2^{z-1}, appears in π_1 before r_1. But since w_2^{z-1} writes v_{z-1} to X_2 and r_1 reads v_{z-2} from X_2, this violates the sequential specification of X_2 (v_{z-2} is written only by w_2^{z-2}). This contradicts the assumption that P guarantees fork sequential consistency in all executions. □

5.6 Comparing Forking and Causal Consistency Conditions

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

Theorem 35. *Every fork-linearizable history w.r.t. a functionality F 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' also precedes o 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 $\mathit{read}_k(X) \rightarrow v$. 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 to each other, in the sense that neither notion implies the other one. We consider a storage service functionality with multiple SWMR registers.

The next theorem shows that a fork-*-linearizable history may not be causally consistent with respect to functionalities with two registers.

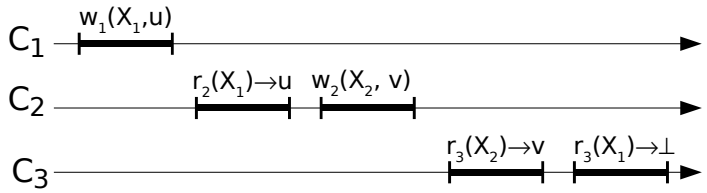

Figure 5.7: A fork-*-linearizable history that is not causally consistent.

Theorem 36. *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 5.7: Client C_1 executes $\mathit{write}_1(X_1, u)$, then client C_2 executes $\mathit{read}_2(X_1) \rightarrow u$, $\mathit{write}_2(X_2, v)$, and finally, client C_3 executes $\mathit{read}_3(X_2) \rightarrow$

v, *read*$_3(X_1) \rightarrow \perp$. Define the client views according to the definition of fork-*-linearizability as

$$\begin{aligned}
\pi_1 :&\ \ \textit{write}_1(X_1, u).\\
\pi_2 :&\ \ \textit{write}_1(X_1, u),\ \textit{read}_2(X_1) \rightarrow u,\ \textit{write}_2(X_2, v).\\
\pi_3 :&\ \ \textit{write}_2(X_2, v),\ \textit{read}_3(X_2) \rightarrow v,\ \textit{read}_3(X_1) \rightarrow \perp.
\end{aligned}$$

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: *write*$_1(X_1, u)$ causally precedes *write*$_2(X_2, v)$ which itself causally precedes *read*$_3(X_1) \rightarrow \perp$; thus, returning $\perp$ violates the sequential specification of a read/write register. □

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

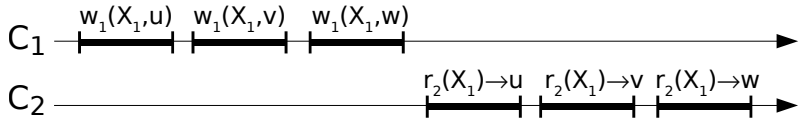

Figure 5.8: A causally consistent execution that is not fork-*-linearizable.

Theorem 37. *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 5.8: Client C_1 executes three write operations, *write*$_1(X_1, u)$, *write*$_1(X_1, v)$, and *write*$_1(X_1, w)$. After the last one completes, client C_2 executes three read operations, *read*$_2(X_1) \rightarrow u$, *read*$_2(X_1) \rightarrow v$, and *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:

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

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

5.7 Weak Fork-Linearizability

We introduce a new consistency notion, called *weak fork-linearizability*, which does not suffer from the availability problem inherent in all previously defined forking semantics, namely it permits wait-free protocols, and on the other hand it is not "too weak", and in fact, unlike fork-*-linearizability, weak fork-linearizability implies causal consistency. Our new notion of weak fork-linearizability is used in Chapters 6 and 7 as a building-block for providing higher-level and more intuitive semantics.

It is based on the notion of *weak* real-time order that removes the anomaly in fork-*-linearizability and allows the last operation of every client to violate real-time order. Let π be a sequence of events and let $\textit{lastops}(\pi)$ be a function of π returning the set containing the last operation from every client in π (if it exists), that is,

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

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 $\textit{lastops}(\pi)$ preserves the real-time order of σ. With these notions, we are now ready to state weak fork-linearizability.

Definition 14 (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 $o' \to_\sigma o$, 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 14 explicitly requires causal consistency; this is implied by fork-linearizability, as shown in Section 5.6. 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.

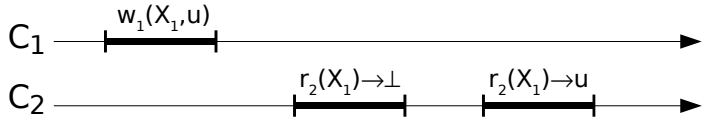

Figure 5.9: A weak fork-linearizable history that is not fork-linearizable.

Consider the following history, shown in Figure 5.9: Initially, X_1 contains $\perp$. Client C_1 executes $write_1(X_1, u)$, then client C_2 executes $read_2(X_1) \rightarrow \perp$ 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:

$$\pi_1 : \ write_1(X_1, u)$$
$$\pi_2 : \ read_2(X_1) \rightarrow \perp, \ write_1(X_1, u), \ read_2(X_1) \rightarrow u$$

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 5.5 and follows, intuitively, because the real-time order condition of weak fork-linearizability is less restrictive than the corresponding

condition of fork-*-linearizability. On the other hand, however, weak fork-linearizability requires causal consistency, whereas fork-*-linearizability does not.

Chapter 6

FAUST: Fail-Aware Untrusted Storage

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

In this chapter we introduce the abstraction of a *fail-aware untrusted service*, with meaningful semantics even when the storage 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. A preliminary version of this work was published in 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) [12].

[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.

6.1 Introduction

In this chapter, we tackle the challenge of providing meaningful service semantics with an untrusted (possibly Byzantine) service provider 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 [74] 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 [38], a basic condition that has proven to be important in various applications [6, 76]. 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 [29] 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 6.2, 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 of being 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 [59, 48, 15]. Forking semantics ensures 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 [59, 48].

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 [48, 49]. However, none of the existing forking consistency semantics allow for *wait-free* implementations; in previous protocols [48, 15] concurrent operations by different clients may block each other, even if the provider is correct. In fact, as we have shown in Section 5.5, this is not merely a shortcoming of a specific implementation, but rather inherent in the semantics. Specifically, no fork-linearizable storage protocol can be wait-free in all executions where the server is correct. Moreover, this limitation is inherent also in the other previously defined forking semantics, namely fork-*-linearizability (when adapted to our model with only one server) and fork-sequential-consistency. Fork-*-linearizability also permits a faulty server to violate causal consistency, as we show in Section 5.6. Thus, a new definition of the untrusted storage building block was needed, which will be strong enough to be useful for building fail-aware untrusted storage, and yet weak enough to allow for wait-free implementations.

In Section 5.7, we defined 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 6.3 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 6.4. 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.

In addition to the client-server communication channels, the FAUST protocol assumes a low-bandwidth communication channel among every pair of clients, which is 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 6.1.

Although this chapter focuses on fail-aware untrusted services that provide a data storage func-

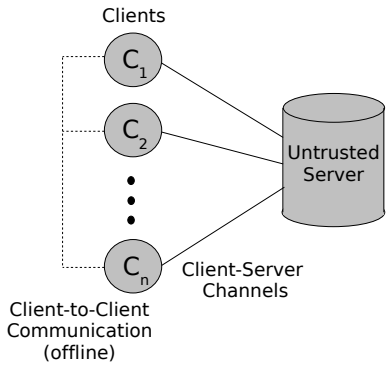

Figure 6.1: System architecture. Client-to-client communication may use offline message exchange.

tionality, we believe that the notion can be generalized to a variety of additional functionalities.

6.2 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 15 (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 $\mathit{stable}_i(W)$ 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 16 (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 that 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 6.2). 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.

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 6.3. The second step consists of extending the Byzantine emulation to a fail-aware storage protocol, as presented in Section 6.4.

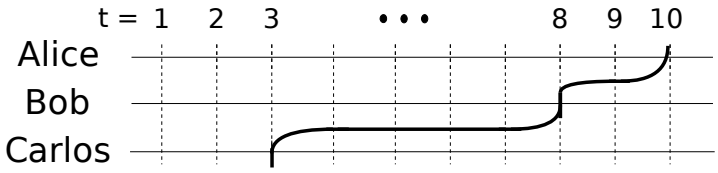

Figure 6.2: The stability cut of Alice indicated by the notification $stable_{Alice}([10, 8, 3])$. The values of t are the timestamps returned by the operations of Alice.

6.3 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 3, and the USTOR protocol for the server appears in Algorithm 4. 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 a **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

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

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 ar-

ray *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 17 (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, $(V_i, M_i) \stackrel{\cdot}{\leq} (V_j, M_j)$ 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 6.4, 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 $\mathit{MEM}[i]$ with the received values for every operation, but updates the register value in $\mathit{MEM}[i]$ 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 $\mathit{SVER}[c] = (V^c, M^c, \varphi^c)$. For a read operation from X_j, the reply also includes $\mathit{MEM}[j]$ and $\mathit{SVER}[j]$, 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

Algorithm 3 Untrusted storage protocol (USTOR). Code for client C_i, part 1.

```
101: notation
102:     Strings = {0,1}* ∪ {⊥}
103:     Clients = {1, ..., n}
104:     Opcodes = {READ, WRITE, ⊥}
105:     Invocations = Clients × Opcodes × Clients × Strings

106: state
107:     x̄_i ∈ Strings, initially ⊥                          // hash of most recently written value
108:     (V_i, M_i) ∈ ℕ_0^n × Strings^n, initially (0^n, ⊥^n)   // last version committed by C_i

109: operation write_i(x)                                     // write x to register X_i
110:     (···) ← writex_i(x)
111:     return OK

112: operation writex_i(x)                                    // extended write x to register X_i
113:     t ← V_i[i] + 1                                       // timestamp of the operation
114:     x̄_i ← H(x)
115:     τ ← sign(i, SUBMIT‖WRITE‖i‖t); δ ← sign(i, DATA‖t‖x̄_i)
116:     send message ⟨SUBMIT, t, (i, WRITE, i, τ), x, δ⟩ to S
117:     wait for receiving a message ⟨REPLY, c, (V^c, M^c, φ^c), L, P⟩ from S
118:     updateVersion(i, (c, V^c, M^c, φ^c), L, P)
119:     φ ← sign(i, COMMIT‖V_i‖M_i); ψ ← sign(i, PROOF‖M_i[i])
120:     send message ⟨COMMIT, V_i, M_i, φ, ψ⟩ to S
121:     return (V_i, M_i)

122: operation read_i(X_j)                                    // read from register X_j
123:     (x^j, ···) ← readx_i(X_j)
124:     return x^j

125: operation readx_i(X_j)                                   // extended read from register X_j
126:     t ← V_i[i] + 1                                       // timestamp of the operation
127:     τ ← sign(i, SUBMIT‖READ‖j‖t); δ ← sign(i, DATA‖t‖x̄_i))
128:     send message ⟨SUBMIT, t, (i, READ, j, τ), ⊥, δ⟩ to S
129:     wait for a message ⟨REPLY, c, (V^c, M^c, φ^c), (V^j, M^j, φ^j), (t^j, x^j, δ^j), L, P⟩ from S
130:     updateVersion(j, (c, V^c, M^c, φ^c), L, P)
131:     checkData(c, (V^c, M^c, φ^c), j, (V^j, M^j, φ^j), (t^j, x^j, δ^j))
132:     φ ← sign(i, COMMIT‖V_i‖M_i); ψ ← sign(i, PROOF‖M_i[i])
133:     send message ⟨COMMIT, V_i, M_i, φ, ψ⟩ to S
134:     return (x^j, V_i, M_i, V^j, M^j)
```

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

Algorithm 3 (cont.) Untrusted storage protocol (USTOR). Code for client C_i, part 2.

```
135: procedure updateVersion(j, (c, V^c, M^c, φ^c), L, P)
136:     if not ((V^c, M^c) = (0^n, ⊥^n) or verify(c, φ^c, COMMIT‖V^c‖M^c)) then output fail_i; halt
137:     if not ((V_i, M_i) ≤̇ (V^c, M^c) and V^c[i] = V_i[i]) then output fail_i; halt
138:     (V_i, M_i) ← (V^c, M^c)
139:     d ← M^c[c]
140:     for q = 1, ..., |L| do
141:         (k, oc, l, τ) ← L[q]
142:         if not (M_i[k] = ⊥ or verify(k, P[k], PROOF‖M_i[k])) then output fail_i; halt
143:         V_i[k] ← V_i[k] + 1
144:         if k = i or not verify(k, τ, SUBMIT‖oc‖l‖V_i[k]) then output fail_i; halt
145:         d ← H(d‖k)
146:         M_i[k] ← d
147:     V_i[i] = V_i[i] + 1
148:     M_i[i] ← H(d‖i)

149: procedure checkData(c, (V^c, M^c, φ^c), j, (V^j, M^j, φ^j), (t^j, x^j, δ^j))
150:     if not ((V^j, M^j) = (0^n, ⊥^n) or verify(j, φ^j, COMMIT‖V^j‖M^j)) then output fail_i; halt
151:     if not (t^j = 0 or verify(j, δ^j, DATA‖t^j‖H(x^j))) then output fail_i; halt
152:     if not ((V^j, M^j) ≤̇ (V^c, M^c) and t^j = V_i[j]) then output fail_i; halt
153:     if not (V^j[j] = t^j or V^j[j] = t^j − 1) then output fail_i; halt
```

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 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

Algorithm 4 Untrusted storage protocol (USTOR). Code for server.

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

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

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

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 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 38. *Protocol USTOR in Algorithms 3 and 4 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 Section 6.5. 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 the 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 49 in Section 6.5, 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 $o_r = \mathit{read}_j(X_k) \rightarrow v$ 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.

6.4 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 oper-

ation, 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 [59, 48, 15]. 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 6.3 illustrates the architecture of the FAUST protocol. Below we describe at a high level how FAUST achieves its goals, and refer to Algorithm 5 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

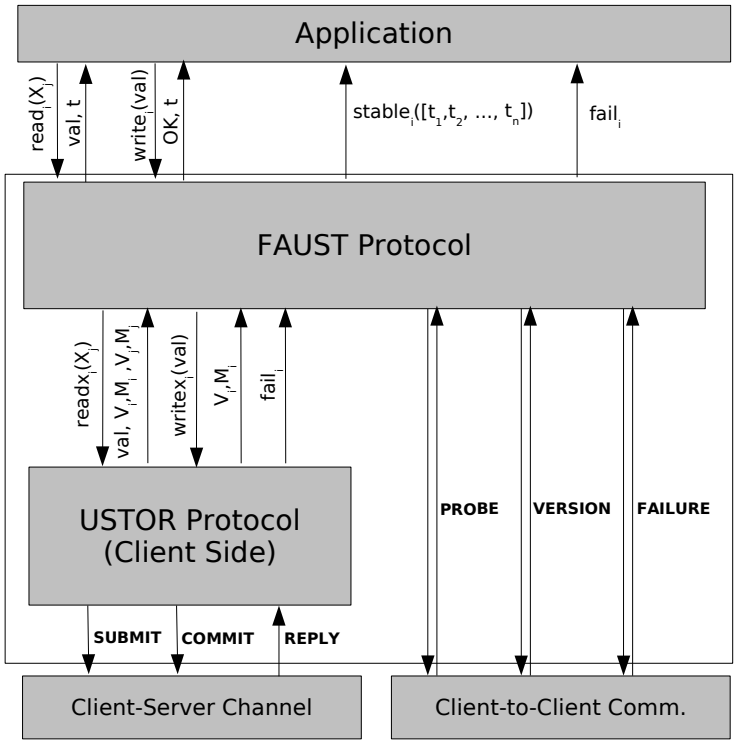

Figure 6.3: Architecture of the fail-aware untrusted storage protocol (FAUST).

VER$_i[j]$ when a *readx*$_i(X_j)$ 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 (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 16 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 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 $(V, M) = \mathit{VER}_j[\mathit{max}_j]$ is 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.

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 [59, 48], 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 $\mathit{USTOR.fail}_i$ 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

Algorithm 5 Fail-aware untrusted storage protocol (FAUST). Code for client C_i.

```
301: state
302:     k_i ∈ Clients, initially 0
303:     VER_i[j] ∈ ℕ_0^n × Strings^n, initially (0^n, ⊥^n), for j = 1,...,n   // biggest received from C_j
304:     max_i ∈ Clients, initially 1                                          // index of client with maximal version
305:     W_i ∈ ℕ_0^n, initially 0^n   // maximal timestamps of C_i's operations observed by different clients
306:     wchange_i ∈ {FALSE, TRUE}, initially TRUE   // indicates that W_i changed since last stable_i(W_i)
307:     execop_i ∈ {FALSE, TRUE}, initially FALSE   // indicates that a non-dummy operation is executing
308:     execdummy_i ∈ {FALSE, TRUE}, initially FALSE   // indicates that a dummy operation is executing
309:     T_i ∈ ℕ^n, initially 0^n                    // time when last updated version was received from C_j

310: operation write_i(x):
311:     execop_i ← TRUE
312:     invoke USTOR.writex_i(x)

313: upon completion of USTOR.writex_i
       with return value (V_i, M_i):
314:     execop_i ← FALSE
315:     update(i, (V_i, M_i))
316:     output (OK, V_i[i])

317: operation read_i(X_j):
318:     execop_i ← TRUE
319:     invoke USTOR.readx_i(X_j)

320: upon completion of USTOR.readx_i
       with return value (x, V_i, M_i, V_j, M_j):
321:     update(i, (V_i, M_i))
322:     update(j, (V_j, M_j))
323:     if execop_i then
324:         execop_i ← FALSE
325:         output (x, V_i[i])
326:     else
327:         execdummy_i ← FALSE

328: periodically:
329:     D ← {C_j | time() − T_i[j] > δ}
330:     send message ⟨PROBE⟩ to all C_j ∈ D
331:     if not execop_i and not execdummy_i then
332:         k_i ← k_i mod n + 1
333:         execdummy_i ← TRUE
334:         invoke USTOR.readx_i(k_i)

335: procedure update(j, (V, M)):
336:     if not ((V, M) ≤̇ VER_i[max_i] or
                VER_i[max_i] ≤̇ (V, M)) then
337:         fail()
338:     if VER_i[j] <̇ (V, M) then
339:         VER_i[j] ← (V, M)
340:         T_i[j] ← time()
341:         if VER_i[max_i] <̇ (V, M) then
342:             max_i ← j
343:         if W_i[j] < V[i] then
344:             W_i[j] ← V[i]
345:             wchange_i ← TRUE

346: upon wchange_i:
347:     wchange_i ← FALSE
348:     output stable_i(W_i)

349: upon receiving msg. ⟨PROBE⟩ from C_j:
350:     send message ⟨VERSION, VER_i[i]⟩ to C_j

351: upon receiving msg. ⟨VERSION,(V, M)⟩ from
       C_j:
352:     update(j, (V, M))

353: procedure fail():
354:     send message ⟨FAILURE⟩ to all clients
355:     output fail_i
356:     halt

357: upon receiving USTOR.fail_i or
       receiving a message ⟨FAILURE⟩ from C_j:
358:     fail()
```

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 39. *Protocol FAUST in Algorithm 5 implements a fail-aware untrusted storage service consisting of n SWMR registers.*

Proof overview. A proof of the theorem appears in Section 6.6; here we sketch its main ideas. Note that properties 1, 2, and 3 of Definition 16 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 $\mathit{VER}_i[i]$ is comparable to all other versions stored in VER_i. Additionally, we show (Lemma 55 in Section 6.6) 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 $\mathit{VER}_i[j]$.

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 $(V_i, M_i) \dot{<} \mathit{VER}_i[j]$ 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.

6.5 Analysis of the Weak Fork-Linearizable Untrusted Storage Protocol

This section is devoted to the proof of Theorem 38. We start with some lemmas that explain how the versions committed by clients should monotonically increase during the protocol execution.

Lemma 40 (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 41. *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 42. *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 41 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 43. *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 41 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 42 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 44. *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 45. *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 46. *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 45, o_k is contained in the view histories of o_i and o_j. Applying Lemma 44 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 sub-sequence of $\mathcal{VH}(o)$; the latter also includes all concurrent operations.

Lemma 47. *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 48. *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 46.

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 47, 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 42. 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 46 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 44. 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 46 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 45 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 46 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 49 (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 48, 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 38.

Theorem 50. *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 3 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 4, 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 45; 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 *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 49. 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 51. *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 51.1. *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 43, 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 42 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 42.

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 42 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 51.2. *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 51.1 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 51.1 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 39 in Section 6.6.

Claim 51.3. *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 51.4. *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 42, the statement follows immediately from Lemma 48.

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 51.2. 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 45 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 48 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 42.

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 42 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 51.5. *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 51.2.

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 3 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 $V_i[j]$ must have been invoked before. □

Claim 51.6. *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 51.1 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 51.4, 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 51.7. *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 51.4–51.7 establish that the protocol is weak fork-linearizable w.r.t. n SWMR registers. □

6.6 Analysis of the Fail-Aware Untrusted Storage Protocol

We prove Theorem 39, i.e., that protocol FAUST in Algorithm 5 satisfies Definition 16. 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 3 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 52 (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 42, $V_i[i]$ is larger than the timestamp of any preceding operation of C_i. □

Lemma 53 (Failure-detection accuracy). *If Algorithm 5 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 38 guarantees that Algorithm 3 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 48 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 16. The causal consistency property follows because weak fork-linearizability implies causal consistency.

Lemma 54 (Linearizability and wait-freedom with correct server, causality). *Let $\tilde{\sigma}$ be a fair execution of Algorithm 5 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 38. 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 51. 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 14. The invariant of sieving then implies that $\tilde{\sigma}|_F$ is also weak fork-linearizable w.r.t. F. □

Lemma 55. *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 42. 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, $(V^*, M^*) = \mathit{VER}_j[\mathit{max}_j]$. Consider the point in the execution when $\mathit{VER}_j[\mathit{max}_j] = (V^*, M^*)$ for the first time. If o_j completes before this point in time, then $(V_j, M_j) \dot{\leq} \mathit{VER}_j[\mathit{max}_j] = (V^*, M^*)$ 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 $\mathit{VER}_j[\mathit{max}_j]$ 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 49 (no-join) implies that (V_j, M_j) and (V^*, M^*) are comparable. □

Lemma 56. *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 $(V^*, M^*) = \mathit{VER}_i[j]$ 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 $(V_i, M_i) \dot{\leq} \mathit{VER}_i[\mathit{max}_i]$. For the same reasons, we have $(V^*, M^*) \dot{\leq} \mathit{VER}_i[\mathit{max}_i]$. Hence, Lemma 49 (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 $\mathit{stable}_i(W_i)$ occurs at C_i, then $W_i[j] \geq V_i[i]$. According to lines 343–345 in Algorithm 5, we have that $V^*[i] = W_i[j] \geq V_i[i]$. Then Lemma 45 implies that o_i appears in $\mathcal{VH}(o^*)$. By Lemma 48, 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 48, 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 55. In the first case, the lemma follows from Lemma 49 (no-join), and in the second case, the lemma follows by the transitivity of versions. □

Lemma 57 (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 51. 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 sub-sequence 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 51.4, 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 56 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 51.3 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 $\mathit{complete}(\tilde{\tau}')|_{C_j}$ is the same. By Claim 51.1, 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 56, 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 51.5 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 58 (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 $\mathit{stable}_i(W)$ 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 $\mathit{stable}_i(W)$ 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 $\mathit{VER}_j[\mathit{max}_j]$ eventually contains a version that is greater than or equal to (V_i, M_i). Second, we show that also $\mathit{VER}_i[j]$ 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 $\mathit{VER}_j[\mathit{max}_j]$ 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 $(V_i, M_i) \dot{\leq} \mathit{VER}_j[\mathit{max}_j]$ 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 $(V_i, M_i) \dot{\leq} \mathit{VER}_j[\mathit{max}_j]$ 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 $\mathit{VER}_i[j]$.

Suppose that after the first time when $(V_i, M_i) \dot{\leq} \mathit{VER}_j[\mathit{max}_j]$ 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 $\mathit{VER}_i[j]$. 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 43 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 42. Then Lemma 49 (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 $\mathit{VER}_i[j]$ is greater than or equal to (V_i, M_i).

To conclude the argument, note that when $\mathit{VER}_i[j]$ contains a version greater than or equal to (V_i, M_i) for the first time, then $\mathit{wchange}_i = \text{TRUE}$ and this triggers a $\mathit{stable}_i(W)$ notification with $W[j] \geq t$. □

Chapter 7

Venus: Verification for Untrusted Cloud Storage

This chapter presents Venus, a service for securing user interaction with untrusted cloud storage. Specifically, Venus guarantees integrity and consistency for applications accessing a key-based object store service, without requiring trusted components or changes to the storage provider. Venus completes all operations optimistically, guaranteeing data integrity. It then verifies operation consistency and notifies the application. Whenever either integrity or consistency is violated, Venus alerts the application. We implemented Venus and evaluated it with Amazon S3 commodity storage service. The evaluation shows that it adds no noticeable overhead to storage operations. A preliminary version of this work was published in the ACM Cloud Computing Security Workshop (CCSW) 2010 [71].

7.1 Introduction

In this chapter we present *Venus*, short for *VErificatioN for Untrusted Storage*. With Venus, a group of clients accessing a remote storage provider benefits from two guarantees: *integrity* and *consistency*. *Integrity* means that a data object read by any client has previously been written by some client; it protects against simple data modifications by the provider, whether inadvertent or caused by malicious attack. Note that a malicious provider might also try a "replay attack" and answer to a read operation with properly authenticated data from an older version of the object,

which has been superseded by a newer version. Venus restricts such behavior and guarantees that either the returned data is from the latest write operation to the object, ensuring that clients see atomic operations, or that the provider misbehavior is exposed. This is the *consistency* property of Venus, which allows multiple clients to access the stored data concurrently in a consistent fashion.

Venus notifies the clients whenever it detects a violation of integrity or consistency. Applications may handle this error in a variety of ways, such as switching to another service provider. Venus works transparently with simple object-based cloud storage interfaces, such that clients may continue to work with a commodity storage service of their choice without changing their applications.

During normal operation, clients of cloud storage should not have to communicate with each other. If clients did communicate, they could simply exchange the root value of a hash tree on the stored objects to obtain consistency. This, however, would introduce a prohibitive coordination overhead — clients should be able to execute operations in isolation, when the other clients are disconnected. But without client-to-client communication for every operation, a malicious service could simply ignore write operations by some clients and respond to other clients with outdated data, as we have demonstrated in Section 5.4. Previous solutions dealt with the problem using so-called "forking" semantics (in SUNDR [59, 48], and other proposals [15, 55, 11]). These solutions guarantee integrity, and by adding some extra out-of-band communication among the clients can also be used to achieve a related notion of consistency. However, they also incur a major drawback that hampers system availability. Specifically, even when the service functions correctly, all these protocols may sometimes block a client during an operation, requiring the client to wait for another client to finish, and do not guarantee that every client operation successfully completes. We have shown in Section 5.5 that this limitation is inherent.

Venus eliminates this problem by letting operations finish *optimistically* and establishing consistency later. When the service is correct, all client operations therefore terminate immediately and the service is "wait-free." When an operation returns optimistically, it is called *red*, and Venus guarantees integrity, but not yet consistency. If the storage service is indeed correct, Venus notifies the application later when a red operation is known to be consistent and thereby becomes *green*; in this sense, Venus is eventually consistent. Venus guarantees that the green operations of all clients are consistent, i.e., that they can be ordered in a single sequence of atomic operations. If some red operations are irreconcilable and so may never become green, Venus ensures that every client

eventually receives a failure notification.

Venus does not require any additional trusted components and relies only on the clients that are authorized to access the data. Venus allows a dynamic set of potentially disconnected clients. A subset of clients that are frequently online is designated as a *core set*; these clients manage the group membership and help to establish consistency. Venus assumes that clients are correct or may crash silently, but otherwise follow their specification, and that a majority of the clients in the core set is correct. The storage service usually functions properly, but may become subject to attacks or behave arbitrarily. Venus is asynchronous and never violates consistency or integrity due to timeouts, but relies on some synchrony assumptions for liveness. Clients may occasionally communicate with each other by email. Since this is conducted in the background, independently of storage operations, and only if a client suspects that the storage service is faulty, it does not affect the performance of Venus.

Venus implementation is comprised of a *client-side library* and a *verifier*. The client-side library overrides the interface of the storage service, extending it with eventual consistency and failure notifications. The verifier brokers consistency information and can be added to the storage service in a modular way; typically it will also run in the cloud, hosted by the same untrusted service that provides the storage. Internally, the verifier and the storage service might be replicated for fault tolerance and high availability. Note that using replication within the cloud does not solve the problem addressed by Venus, since from the client's perspective, the entire cloud is a single trust domain. We stress that Venus does not trust the verifier any more than the storage service — the two entities may collude arbitrarily against the clients, and separating them simply supports a layered implementation with commodity providers. Of course, the verifier could be run by a trusted third party, but it would be a much stronger assumption and existing protocols suffice for integrity and consistency in this model [8].

We have implemented Venus and deployed it using the commodity Amazon S3 cloud storage service[1]. Venus requires an additional message exchange between client and verifier for each operation, in addition to accessing the raw storage. We report on experiments using Venus connected to S3 and with a verifier deployed either on a remote server or on the same LAN as the clients. We compare the performance of storage access using Venus to that of the raw S3 service. Both the latency and the throughput of Venus closely match the performance of the raw S3 service. Specif-

[1] `http://aws.amazon.com/s3/`

ically, when the verifier is deployed on the local LAN, Venus' performance is identical to that of S3. When the verifier is deployed remotely, Venus adds a small overhead to latency compared to S3 (corresponding to one round of additional communication with the verifier) and achieves the same throughput. We have also tested Venus' capability to detect service misbehavior and present logs from such an experiment, where the clients communicate with each other and detect that the cloud storage provider has violated consistency (as simulated).

Contributions Our results demonstrate that data integrity and consistency for remote storage accessed by multiple clients can be obtained with insignificant overhead, no additional trusted components, and without interrupting normal operations. Specifically, Venus is the first practical decentralized algorithm that

- verifies cryptographic integrity and consistency of remotely stored data accessed by multiple clients without introducing trusted components,
- does not involve client-to-client coordination or introduce extra communication on the critical path of normal operations,
- provides simple semantics to clients, lets operations execute optimistically, but guarantees that either all operations eventually become consistent, or that every client is informed about the service failure, and
- is practically implemented on top of a commodity cloud storage service.

Venus may secure a variety of applications that currently use cloud storage, such as online collaboration, Internet backup, and document archiving. No less important is that Venus can encourage applications that require verifiable guarantees, and cannot afford to blindly trust services in the cloud, to consider taking advantage of what the cloud has to offer.

Comparison to FAUST. FAUST [12] (presented in Chapter 6) is an algorithm that implements the notion of weak fork-linearizability, which allows client operations to complete optimistically, as in Venus. It also provides notifications to clients, but they are different and less intuitive — FAUST issues *stability* notifications, where each notification includes a vector indicating the level of synchronization that a client has with every other client. This stability notion is not transitive and requires users to explicitly track the other clients in the system and to assess their relation to the data accessed by the operation. FAUST is therefore not easily amenable to dynamic changes in the

set of clients. Furthermore, global consistency in FAUST (among all clients) is guaranteed only if no client ever crashes. FAUST does not work with commodity storage – like other proposals it integrates storage operations with the consistency mechanism and moreover it does not allow multiple clients to modify the same object, which is the usual semantics of commodity storage services.

In contrast, indications in Venus simply specify the last operation of the client that has been verified to be globally consistent, which is easy to integrate with an application. Venus eliminates the need for clients to track one another, and enables dynamic client changes. Unlike the previous protocols [12, 15, 55], Venus allows all clients to modify the same shared object. Most importantly, the design of Venus is modular, so that it can be deployed with a commodity storage service.

Organization The remainder of the chapter is organized as follows: Section 7.2 presents the design of Venus, and Section 7.3 defines its semantics. The protocol for clients and the verifier is given in Section 7.4. Section 7.5 describes our implementation of Venus, and finally, Section 7.6 presents its evaluation.

7.2 System Model

Figure 7.1 depicts our system model, which includes a *storage service*, a generic commodity online service for storing and retrieving objects of arbitrary size, a *verifier*, which implements our consistency and verification functions and multiple *clients*. The storage service is used as is, without any modification. Usually the storage service and the verifier are hosted in the same cloud and will be correct; but they may become faulty or corrupted by an adversary, and they may collude together against the clients.

There are an arbitrary number of clients, which are subject to crash failures. Clients may be connected intermittently and are frequently offline. The *core set* of clients is a publicly known subset of the clients with a special role. These clients help detect consistency and failures (Section 7.4.4) and manage client membership (Section 7.4.6); to this end, clients occasionally communicate directly with clients from the core set. A quorum of the core set of clients must not crash (but may also be offline temporarily). Note that clients of cloud services, and especially users of cloud storage, do not operate continuously. Hence, clients should not depend on other clients

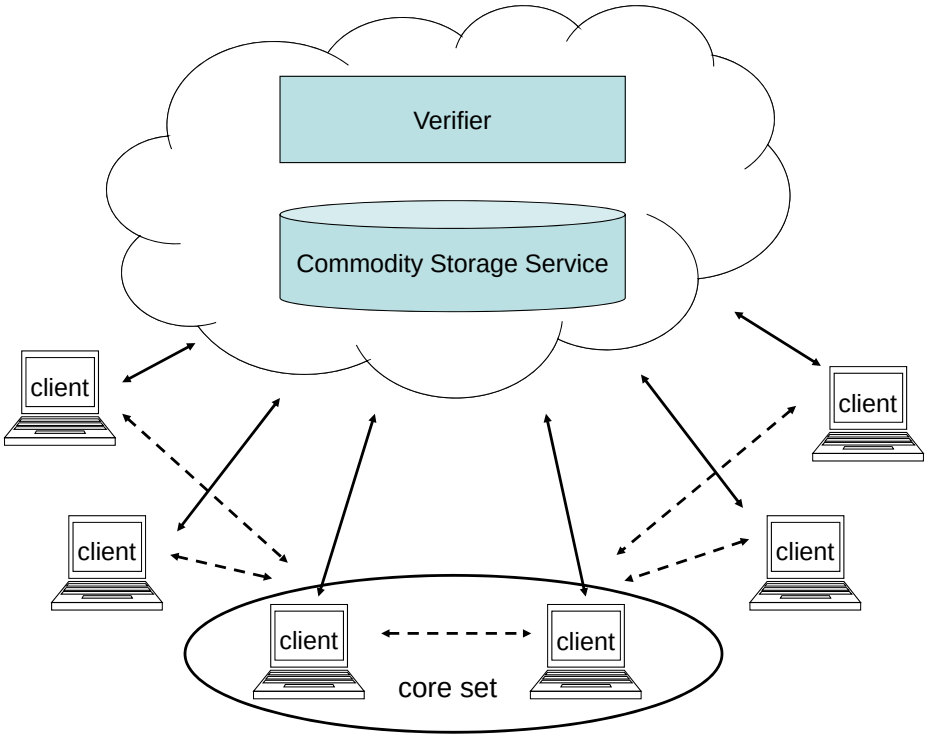

Figure 7.1: Venus Architecture.

for liveness of their operations. Indeed, every operation executed by a client in Venus proceeds independently of other clients and promptly completes, even if all other clients are offline.

Clients in Venus are honest and do not deviate from their specification (except for crashing). Note that tolerating malicious clients does not make a lot of sense, because every client may write to the shared storage. From the perspective of the correct clients, the worst potential damage by another client is to simply overwrite the storage with bogus information. Venus, just like commodity cloud storage, cannot perform application-specific validation of written data.

Venus clients are admitted by a member of the core set, as determined by the same access-control policy as the one used at the commodity storage interface. Clients are identified by a signature public key and an email address, bound together with a self-signed certificate. Every client knows initially at least the public keys of all clients in the core set.

Messages between clients and the verifier or the storage service are sent over reliable point-to-point channels. Client-to-client communication is conducted using digitally signed email messages; this allows clients to go temporarily offline or to operate behind firewalls and NATs. Clients rarely communicate directly with each other.

The storage service is assumed to have an interface for writing and reading data objects. The

write operation takes the identifier *obj* of the object and some data x as parameters and returns an acknowledgment. The *read* operation expects an object identifier *obj* and returns the data stored in the object. After a new object is successfully stored, clients are able to read it within a bounded period of time, though perhaps not immediately. We assume that this bound is known; in practice, it can be obtained dynamically[2]. The assumption that such time threshold exists reflects clients' expectation from any usable storage service. Inability to meet this expectation (e.g., due to an internal partition) can be perceived as a failure of the storage provider as far as clients are concerned. Venus makes several attempts to read the object, until this time bound is exceeded, at which time a failure notification is issued to clients.

7.3 Venus Interface and Semantics

Venus overrides the *write*(obj, x) and *read*(obj) operations for accessing an object identified by *obj* in the interface of the storage service. Venus does not allow partial updates of an object, the value x overwrites the value stored previously. If the object does not exist yet, it is created. For simplicity of presentation, we assume that each client executes operations sequentially.

Venus extends the return values of write and read operations by a local timestamp t, which increasing monotonically with the sequence of operations executed by the client. An operation o always completes optimistically, without waiting for other clients to complete their operations; at this point, we say that o is *red*, which means that the integrity of the operation has been checked, but its consistency is yet unverified.

A weak form of consistency is nevertheless guaranteed for all operations that become red. Namely, they ensure *causal consistency* [38], which means intuitively that all operations are consistent with respect to potential causality [42]. For example, a client never reads two causally related updates in the wrong order. In addition, it guarantees that a read operation never returns an outdated value, if the reader was already influenced by a more recent update. Causality has proven to be important in various applications, such as various collaborative tools and Web 2.0 applications [62, 76]. Although usually necessary for applications, causality is often insufficient. For example, it does not rule out replay attacks or prevent two clients from reading two different

[2]Amazon guarantees that S3 objects can be read immediately after they are created: `http://aws.typepad.com/aws/2009/12/aws-importexport-goes-global.html`

versions of an object.

Venus provides an asynchronous callback interface to a client, which issues periodic consistency and failure notifications. A consistency notification specifies a timestamp t that denotes the most recent *green* operation of the client, using the timestamp returned by operations. All operations of the client up to this operations have been verified to be consistent and are also green. Intuitively, all clients observe the green operations in the same order. More precisely, Venus ensures that there exists a global sequence of operations, including at least the green operations of all clients, in which the green operations appear according to their order of execution. Moreover, this sequence is *legal*, in the sense that every read operation returns the value written by the last write that precedes the read in the sequence, or an empty value if no such write exists. Note that the sequence might include some red operations, in addition to the green ones. This may happen, for instance, when a client starts to write and crashes during the operation, and a green read operation returns the written value.

Failure notifications indicate that the storage service or the verifier has violated its specification. Venus guarantees that every complete operation eventually becomes green, unless the client executing it crashes, or a failure is detected.

7.4 Protocol Description

Section 7.4.1 describes the interaction of Venus clients with the storage service. Section 7.4.2 describes *versions*, used by Venus to check consistency of operations. Section 7.4.3 presents the protocol between the clients and the verifier. Section 7.4.4 describes how clients collect information from other clients (either through the verifier or using client-to-client communication), and use it for eventual consistency and failure detection. Section 7.4.5 describes optimizations.

For simplicity, we first describe the protocol for a fixed set of clients $C_1, \ldots, C_n$, and relax this assumption later in Section 7.4.6. The algorithm uses several timeout parameters, which are introduced in Table 7.1. We have formally proven that Venus provides the properties defined in Section 7.3.

In what follows, we distinguish between objects provided by Venus and which can be read or written by applications, and objects which Venus creates on storage. The former are simply called objects, while the latter are called *low-level objects*. Every update made by the application to an

object *obj* managed with Venus creates a new low-level object at the storage service with a unique identifier, denoted p_x in the description below, and the verifier maintains a pointer to the last such update for every object managed by Venus. Clients periodically garbage-collect such low-level objects (see also Section 7.4.5).

R	Number of times an operation is retried on the storage service.
t_{dummy}	Frequency of dummy-read operations.
t_{send}	Time since last version observed from another client, before that client is contacted directly.
$t_{receive}$	Frequency of checking for new messages from other clients.

Table 7.1: Venus timeout parameters.

7.4.1 Overview of read and write operations

The protocol treats all objects in the same way; we therefore omit the object identifier in the sequel.

The general flow of read and write operations is presented in Figure 7.2. When a *write*(x) operation is invoked at a client C_i to update the object, the client calculates h_x, a cryptographic hash of x, and writes x to the storage service, creating a new low-level object with a unique path p_x, chosen by the client-side library. Using p_x as a handle, the written data can later be retrieved from storage. Notice that p_x identifies the low-level object created for this update, and it is different from the object identifier exported by Venus, which is not sent to storage. After the low-level write completes, C_i sends a SUBMIT message to the verifier including p_x and h_x, informing it about the write operation. C_i must wait before sending the SUBMIT message, since if C_i crashes before x is successfully stored, p_x would not be a valid handle and read operations receiving p_x from the verifier would fail when trying to retrieve x from the storage service. The verifier orders all SUBMIT messages, creating a global sequence $\mathcal{H}$ of operations.

When a *read* operation is invoked, the client first sends a SUBMIT message to the verifier, in order to retrieve a handle corresponding to the latest update written to the object. The verifier responds with a REPLY message including p_x and h_x from the latest update. The reader then contacts the storage service and reads the low-level object identified by p_x. In most cases, the data will be returned by the storage service. The reader then checks the integrity of the data by computing its hash and comparing it to h_x; if they are equal, it returns the data to the application. If the storage provider responds that no low-level object corresponds to p_x, the client re-executes

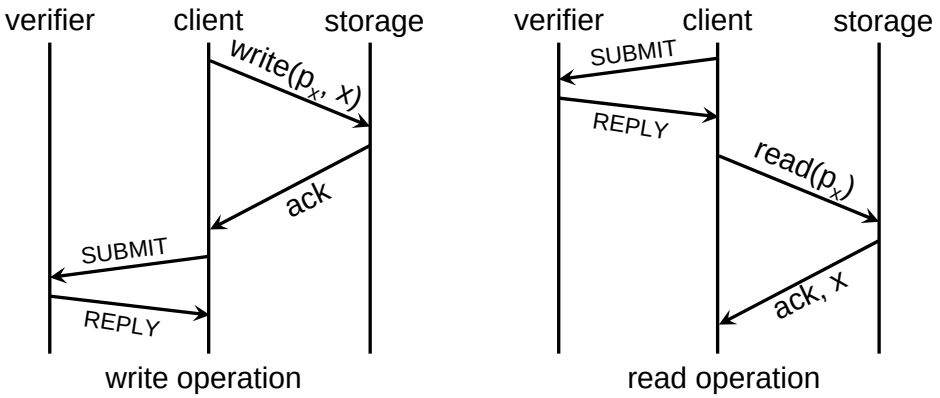

Figure 7.2: Operation flow.

the read. If the correct data can still not be read after R repetitions, the client announces a failure. Similarly, failure is announced if hashing the returned data does not result in h_x. Updates follow the same pattern: if the storage does not successfully complete the operation after R attempts, then the client considers it faulty.

Since the verifier might be faulty, a client must verify the integrity of all information sent by the verifier in REPLY messages. To this end, clients sign all information they send in SUBMIT messages. A more challenging problem, which we address in the next section, is verifying that p_x and h_x returned by the verifier correspond to the latest write operation, and in general, that the verifier orders the operations correctly.

7.4.2 From timestamps to versions

In order to check that the verifier constructs a correct sequence $\mathcal{H}$ of operations, our protocol requires the verifier to supply the *context* of each operation in the REPLY. The context of an operation o is the prefix of $\mathcal{H}$ up to o, as determined by the client that executes o. This information can be compactly represented using *versions* as follows.

Every operation executed by a client C_i has a local timestamp, returned to the application when the operation completes. The timestamp of the first operation is 1 and it is incremented for each subsequent operation. We denote the timestamp of an operation o by $ts(o)$. Before C_i completes o, it determines a vector-clock value $vc(o)$ representing the context of o; the j-th entry in $vc(o)$ contains the timestamp of the latest operation executed by client C_j in o's context.

In order to verify that operations are consistent with respect to each other, more information

about the context of each operation is needed. Specifically, the context is compactly represented by a version, as in previous works [59, 15, 12]. A *version*(o) is a pair composed of the vector-clock *version*$(o).vc$, which is identical to $vc(o)$, and a second vector, *version*$(o).vh$, where the i-th entry contains a cryptographic hash of the prefix of $\mathcal{H}$ up to o. This hash is computed by iteratively hashing all operations in the sequence with a cryptographic collision-resistant hash function. Suppose that o_j is the last operation of C_j in C_i's context, i.e., *version*$(o).vc[j] = ts(o_j)$. Then, the j-th entry in *version*$(o).vh$ contains a representation (in the form of a hash value) of the prefix of $\mathcal{H}$ up to o_j. Client C_i calculates *version*$(o).vh$ during the execution of o according to the information provided by the verifier in the REPLY message. Thus, if the verifier follows its protocol, then *version*$(o).vh[j]$ is equal to *version*$(o_j).vh[j]$.

For simplicity, we sometimes write $vc(o)$ and $vh(o)$ for *version*$(o).vc$ and *version*$(o).vh$, respectively. We define the following order (similarly to [59, 15, 12]), which determines whether o could have appeared before another operation o' in the same legal sequence of operations:

Order on versions: *version*$(o) \leq$ *version*(o') whenever both of the following conditions hold:

1. $vc(o) \leq vc(o')$, i.e., for every k, $vc(o)[k] \leq vc(o')[k]$.
2. For every k such that $vc(o)[k] = vc(o')[k]$, it holds that $vh(o)[k] = vh(o')[k]$.

The first condition checks that the context of o' includes at least all operations that appear in the context of o. Suppose that o_k is the last operation of C_k appearing both in the context of o and in that of o'. In this case, the second condition verifies that the prefix of $\mathcal{H}$ up to o_k is the same in the contexts of o and o'. We say that two versions are *comparable* when one of them is smaller than or equal to the other. The existence of incomparable versions indicates a fault of the verifier.

7.4.3 Operation details

Each client maintains a version corresponding to its most recently completed operation o_{prev}. Moreover, if o_{prev} is a read operation, the client keeps p_{prev} and h_{prev} retrieved by o_{prev} from the verifier. Note that client C_i does not know context and version of its current operation when it sends the SUBMIT message, as it only computes them after receiving the REPLY. Therefore, it sends the version of o_{prev} with the SUBMIT message its next operation to the verifier.

When sending the SUBMIT message for a READ operation o, C_i encloses a representation of o (including the timestamp $ts(o)$), the version o_{prev} of its previous operation as well as a signature on

$vh(o_{prev})[i]$. Such a signature is called a *proof* and authenticates the prefix of C_i's context of o_{prev}. If o is a write operation, the message also includes the tuple $(p_x, h_x, ts(o))$, where p_x is the handle and h_x is the hash of the data already written to the storage provider. Otherwise, if o is a read operation, and o_{prev} was also a read, the message includes $(p_{prev}, h_{prev}, ts(o_{prev}))$. All information in the SUBMIT message is signed (except for the proof, which is a signature by itself).

Recall that the verifier constructs the global sequence $\mathcal{H}$ of operations. It maintains an array *Ver*, in which the j-th entry holds the last version received from client C_j. Moreover, the verifier keeps the index of the client from which the maximal version was received in a variable c; in other words, *Ver*$[c]$ is the maximal version in *Ver*. We denote the operation with version *Ver*$[c]$ by o_c. The verifier also maintains a list *Pending*, containing the operations that follow o_c in $\mathcal{H}$. Hence, operations appear in *Pending* according to the order in which the verifier received them from clients (in SUBMIT messages). Furthermore, a variable *Proofs* contains an array of proofs from SUBMIT messages. Using this array, clients will be able to verify their consistency with C_j up to C_j's previous operation, before they agree to include C_j's next operation in their context.

Finally, the verifier stores an array *Paths* containing the tuple $(p_x, h_x, ts(o))$ received most recently from every client. Notice that if the last operation of a client C_j is a *write*, then this tuple is included in the SUBMIT message and the verifier updates *Paths*$[j]$ when it receives the SUBMIT. On the other hand, the SUBMIT message of a *read* operation does not contain the handle and the hash; the verifier updates *Paths*$[j]$ only when it receives the next SUBMIT message from C_j. The verifier processes every SUBMIT message atomically, updating all state variables together, before processing the next SUBMIT message.

After processing a SUBMIT message, the verifier sends a REPLY message that includes c, *version*(o_c), *Pending*, *Proofs* (only those entries in *Proofs* which correspond to clients executing operations in *Pending*), and for a read operation also a tuple (p_x, h_x, t_x) with a handle, hash, and timestamp as follows. If there are write operations in *Pending*, then the verifier takes (p_x, h_x, t_x) from the entry in *Paths* corresponding to the client executing the last write in *Pending*. Otherwise, if there are no writes in *Pending*, then it uses the tuple (p_x, h_x, t_x) stored in *Paths*$[c]$.

When C_i receives the REPLY message for its operation o, it verifies the signatures on all information in the message, and then performs the following checks:

1. The maximal version sent by the verifier, *version*(o_c), is at least as big as the version corresponding to C_i's previous operation, *version*(o_{prev}).

```
1:  function compute-version-and-check-pending(o)
2:      (vc, vh) ← version(o_c)
3:      histHash ← vh[c]
4:      for q = 1, ..., |Pending| :                          // traverse pending ops
5:          let C_j be the client executing Pending[q]
6:          vc[j] ← vc[j] + 1
7:          histHash ← hash(histHash||Pending[q])
8:          vh[j] ← histHash
9:          perform checks 4, 5, and 6 (see text below)
10:     version(o) = (vc, vh)
11:     return version(o)
```

Figure 7.3: Computing the version of an operation.

2. The timestamp $ts(o_{prev})$ of C_i's previous operation is equal to $vc(o_c)[i]$, as o_{prev} should be the last operation that appears in the context of o_c.
3. If o is a read operation, then (p_x, h_x, t_x) indeed corresponds to the last write operation in *Pending*, or to o_c if there are no write operations in *Pending*. This can be checked by comparing t_x to the timestamp of the appropriate operation in *Pending* or to $ts(o_c)$, respectively.

Next, C_i computes *version*(o), by invoking the function shown in Figure 7.3, to represent o's context based on the prefix of the history up to o_c (represented by *version*(o_c)), and on the sequence of operators in *Pending*. The following additional checks require traversing *Pending*, and are therefore performed during the computation of *version*(o), which iterates over all operations in *Pending*:

4. There is at most one operation of every client in *Pending*, and no operation of C_i, that is, the verifier does not inlcude too many operations in *Pending*.
5. For every operation o by client C_j in *Pending*, the timestamp $ts(o)$ is equal to $vc(o_c)[j] + 1$, that is, o is indeed the next operation executed by C_j after the one appearing in the context of o_c.
6. For every client C_j that has an operation in *Pending*, *Proofs*$[j]$ is a valid signature by C_j on $vh(o_c)[j]$. That is, the context of o_c includes and properly extends the context of the previous operation of C_j, as represented by the hash $vh(o_c)[j]$ and the signature *Proofs*$[j]$.

If one of the checks fails, the application is notified and a failure message is sent to the core set clients, as described in Section 7.4.4.

7.4.4 Detecting consistency and failures

An application of Venus registers for two types of callback notifications: consistency notifications, which indicate that some operations have become green and are known to be consistent, and failure notifications, issued when a failure of the storage service or the verifier has been detected. Below we describe the additional mechanisms employed by the clients for issuing such notifications, including client-to-client communication.

Each client C_i maintains an array *CVer*. For every client C_j in the core set, *CVer*$[j]$ holds the biggest version of C_j known to C_i. The entries in *CVer* might be outdated, for instance, when C_i has been offline for a while, and more importantly, *CVer*$[j]$ might not correspond to an operation actually executed by C_j, as we explain next. Together with each entry of *CVer*, the client keeps the local time of its last update to the entry.

Every time a client C_i completes an operation o, it calculates *version*(o) and stores it in *CVer*$[i]$. To decide whether its own operations are globally consistent, C_i must also collect versions from other clients. More precisely, it needs to obtain the versions from a majority quorum of clients in the core set. Usually, these versions arrive via the verifier, but they can also be obtained using direct client-to-client communication.

To obtain another client's version via the verifier, C_i piggybacks a VERSION-REQUEST message with every SUBMIT message that it sends. The VERSION-REQUEST message includes the identifier k of some client in the core set. In response, the verifier includes *Version*$[k]$ with the REPLY message. When C_i receives the REPLY message, it updates *CVer*$[k]$ if the received version is bigger than the old one (of course, the signature on the received version must be verified first). Whenever C_i executes an operation, it requests the version of another client from the core set in the VERSION-REQUEST message, going in round-robin over all clients in the core set. When no application-invoked operations are in progress, the client also periodically (every t_{dummy} time units) issues a dummy-read operation, to which it also attaches VERSION-REQUEST messages. The dummy-read operations are identical to application-invoked reads, except that they do not access the storage service after processing the REPLY message. A dummy-read operation invoked by C_i causes an update to *Version*$[i]$ at the verifier, even though no operation is invoked by the application at C_i. Thus, clients that repeatedly request the version of C_i from the verifier see an increasing sequence of versions of C_i.

It is possible, however, that C_k goes offline or crashes, in which case C_i will not see a new

version from C_k and will not update *CVer*$[k]$. Moreover, a faulty verifier could be hiding C_k's new versions from C_i. To client C_i these two situations look the same. In order to make progress faced with this dilemma, C_i contacts C_k directly whenever *CVer*$[k]$ does not change for a predefined time period t_{send}. More precisely, C_i sends the maximal version in *CVer* to C_k, asking C_k to respond with a similar message. When C_k is online, it checks for new messages from other clients every $t_{receive}$ time units, and thus, if C_k has not permanently crashed, it will eventually receive this message and check that the version is comparable to the maximum version in its array *CVer*. If no errors are found, C_k responds to C_i with the maximal version from *CVer*, as demonstrated in Figure 7.4(a). Notice that this maximal version does not necessarily correspond to an operation executed by C_i. All client-to-client messages use email and are digitally signed to prevent attacks from the network.

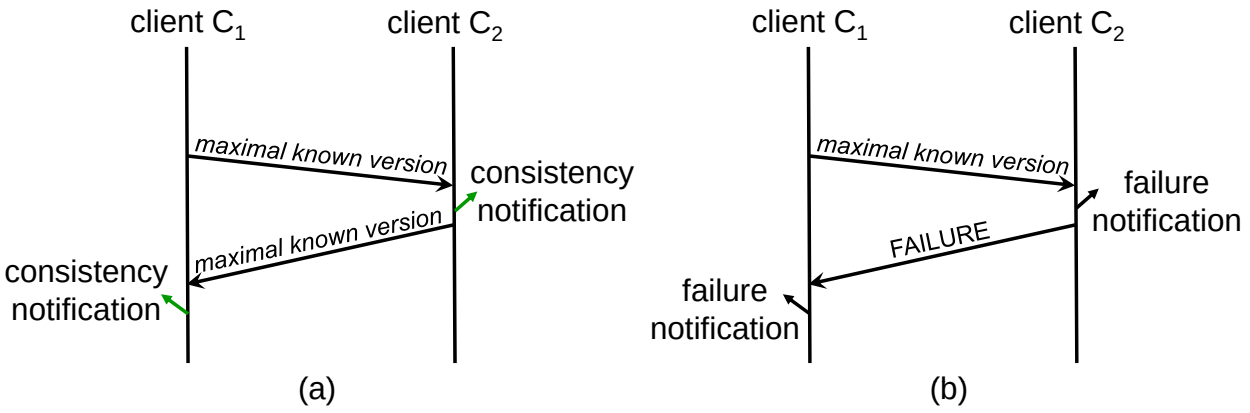

Figure 7.4: Consistency checks using client-to-client communication. In (a) the checks pass, which leads to a response message and consistency notifications. In (b) one of the checks fails and C_2 broadcasts a FAILURE message.

When a client C_i receives a version directly from C_k, it makes sure the received version is comparable with the maximal version in its array *CVer*. If the received version is bigger than *CVer*$[k]$, then C_i updates the entry.

Whenever an entry in *CVer* is updated, C_i checks whether additional operations become green, which can be determined from *CVer* as explained next. If this is the case, Venus notifies the application and outputs the timestamp of the latest green operation. To check if an operation o becomes green, C_i invokes the function in Figure 7.5, which computes a *consistency set* $\mathcal{C}(o)$ of o. If $\mathcal{C}(o)$ contains a majority quorum of the clients in the core set, the function returns *green*, indicating that o is now known to be consistent.

```
1: function check-consistency(o)
2:     C(o) ← ∅
3:     for each client C_k in the core set:
4:         if CVer[k].vc[i] ≥ ts(o) then
5:             add k to C(o)
6:     if C(o) contains a quorum of the core set then
7:         return green
8:     else
9:         return red
```

Figure 7.5: Checking whether o is green.

C_i starts with the latest application-invoked (non-dummy) red operation o, going over its red operations in reverse order of their execution, until the first application-invoked red operation o is encountered that becomes green. If such an operation o is found, C_i notifies the application that all operations with timestamps smaller than or equal to $ts(o)$ are now green.

If at any point a check made by the client fails, it broadcasts a failure message to all core set clients; when receiving such message for the first time, a core set client forwards this message to all other core set clients. When detecting a failure or receiving a failure message, a client notifies its application and ceases to execute application-invoked and dummy operations. After becoming aware of a failure, a core set client responds with a failure message to any received version message, as demonstrated in Figure 7.4(b).

7.4.5 Optimizations and garbage collection

Access to the storage service consumes the bulk of execution time for every operation. Since this time cannot be reduced by our application, we focus on overlapping as much of the computation as possible with the access to storage.

For a read operation, as soon as a REPLY message is received, the client immediately starts reading from the storage service, and concurrently makes all checks required to complete its current operation. In addition, the client prepares (and signs) the information about the current operation that will be submitted with its next operation (notice that this information does not depend on the data returned by the storage service).

A write operation is more difficult to parallelize, since a SUBMIT message cannot be sent to the verifier before the write to the storage service completes. This is due to the possibility that a SUBMIT message reaches the verifier but the writer crashes before the data is successfully written

to the storage service, creating a dangling pointer at the verifier. If this happens, no later read operation will be able to complete successfully.

We avoid this problem by proceeding with the write optimistically, without changing the state of the client or verifier. Specifically, while the client awaits the completion of its write to the storage, it sends a DUMMY-SUBMIT message to the verifier, as shown in Figure 7.6. Unlike a normal SUBMIT, this message is empty and thus cannot be misused by the verifier, e.g., by presenting it to a reader as in the scenario described above. When receiving a DUMMY-SUBMIT message, the verifier responds with a REPLY message identical to the one it would send for a real SUBMIT message (notice that a REPLY message for a write operation does not depend on the contents of the SUBMIT message). The writer then optimistically makes all necessary checks, calculations and signatures. When storing the data is complete, the client sends a SUBMIT message to the verifier. If the REPLY message has not changed, pre-computed information can be used, and otherwise, the client re-executes the checks and computations for the newly received information.

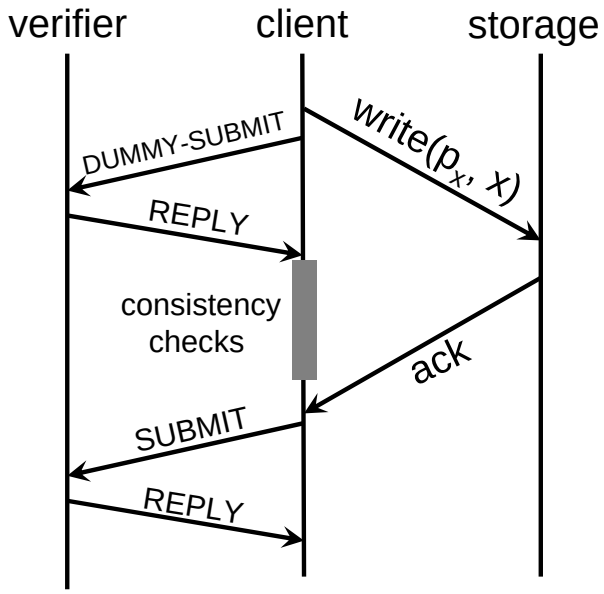

Figure 7.6: Speculative write execution.

Venus creates a new low-level object at the storage provider for every write operation of the application. In fact, this is exactly how updates are implemented by most cloud storage providers, which do not distinguish between overwriting an existing object and creating a new one. This creates the need for garbage collection. We have observed, however, that with Amazon S3 the cost of storing multiple low-level objects for a long period of time is typically much smaller than

the cost of actually uploading them (which is anyway necessary for updates), thus eager garbage collection will not significantly reduce storage costs. In Venus, each client periodically garbage-collects low-level objects on storage corresponding to its outdated updates.

7.4.6 Joining the system

We have described Venus for a static set of clients so far, but in fact, Venus supports dynamic client joins. In order to allow for client joins, clients must have globally unique identifiers. In our implementation these are their unique email addresses. All arrays maintained by the clients and by the verifier, including the vector clock and the vector of hashes in versions, are now associative arrays, mapping a client identifier to the corresponding value. Clients may also leave Venus silently but the system keeps their entries in versions.

The verifier must not accept requests from clients for which it does not have a public key signed by some client in the core set. As mentioned in Section 7.2, every client wishing to join the system knows the core set of clients and their public keys. To join the system, a new client C_i sends a JOIN message, including its public key, to some client in the core set; if the client does not get a response it periodically repeats the process until it gets a successful response. When receiving a JOIN request from C_i, a client C_j in the core set checks whether C_i can be permitted access to the service using the externally defined access policy, which permits a client to access Venus if and only if the client may also access the object at the storage service. If access to C_i is granted, C_j still needs to verify that C_i controls the public key in the JOIN message. To this end, C_j asks the joining client to sign a nonce under the supplied public key, as shown in Figure 7.7.

If the signature returned by C_j is valid, then C_j signs C_i's public key and sends it to the verifier. After the verifier has acknowledged its receipt, C_j sends a final acknowledgment to C_i, and from this time on, C_i may invoke read and write operations in Venus.

The verifier informs a client C_i about clients that are yet unknown to C_i, by including their signed public keys in REPLY messages to C_i. In order to conclude what information C_i is missing, the verifier inspects $\mathit{version}(o_{prev})$ received from C_i in the SUBMIT message, where it can see which client identifiers correspond to values in the associative arrays. A client receiving a REPLY message extracts all public keys from the message and verifies that the signature on each key was made by a client from the core set. Then, it processes the REPLY message as usual. If at any time

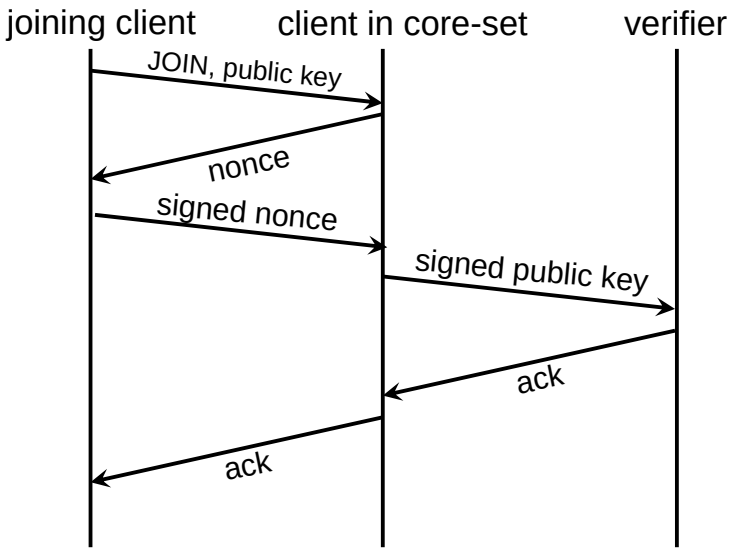

Figure 7.7: Flow of a join operation.

some information is received from the verifier, but a public key needed to verify this information is missing, then C_i concludes that the verifier is faulty and notifies its application and the other clients accordingly.

7.5 Implementation

We implemented Venus in Python 2.6.3, with Amazon S3 as the storage service. Clients communicate with S3 using HTTP. Communication with the verifier uses direct TCP connections or HTTP connections; the latter allow for simpler traversal of firewalls.

Client-to-client communication is implemented by automated emails. This allows our system to handle offline clients, as well as clients behind firewalls or NATs. Clients communicate with their email provider using SMTP and IMAP for sending and receiving emails, respectively. Clients are identified by their email addresses.

For signatures we used GnuPG. Specifically, we used 1024-bit DSA signatures. Each client has a local key-ring where it stores the public keys corresponding to clients in our system. Initially the key-ring stores only the keys of the clients in the core set, and additional keys are added as they are received from the verifier, signed by some client in the core set. We use SHA-1 for hashing.

Venus does not access the versioning support of Amazon S3, which was announced only recently, and relies on the basic key-value store functionality.

Log of Client #1: venusclient1@gmail.com

09:26:38: initializing client venusclient1@gmail.com
09:26:43: executing dummy-read with <REQUEST-VERSION, venusclient2@gmail.com>
-----"----: no update to CVersions[venusclient2@gmail.com]
09:26:45: received email from client venusclient2@gmail.com. Signature OK
-----"----: failure detected: venusclient2@gmail.com sent an incomparable version
-----"----: notifying other clients and shutting down...

Log of Client #2: venusclient2@gmail.com

09:26:30: initializing client venusclient2@gmail.com
09:26:35: executing dummy-read with <REQUEST-VERSION, venusclient1@gmail.com>
-----"----: no update to CVersions[venusclient1@gmail.com]
09:26:40: executing dummy-read with <REQUEST-VERSION, venusclient1@gmail.com>
-----"----: no update to CVersions[venusclient1@gmail.com]
-----"----: sending version to client venusclient1@gmail.com, requesting response
09:26:45: executing dummy-read with <REQUEST-VERSION, venusclient1@gmail.com>
-----"----: no update to CVersions[venusclient1@gmail.com]
09:26:49: received email from client venusclient1@gmail.com. Signature OK
-----"----: failure reported by client venusclient1@gmail.com
-----"----: notifying other clients and shutting down...

Figure 7.8: Client logs from detecting a simulated "split-brain" attack, where the verifier hides each client's operations from the other clients. System parameters were set to $t_{dummy} = 5sec.$, $t_{send} = 10sec.$, and $t_{receive} = 5sec.$ There are two clients in the system, which also form the core set. After 10 seconds, client $\#2$ does not observe a new version corresponding to client $\#1$ and contacts it directly. Client $\#1$ receives this email, and finds the version in the email to be incomparable to its own latest version, as its own version does not reflect any operations by client $\#2$. The client replies reporting of an error, both clients notify their applications and halt.

To evaluate how Venus detects service violations of the storage service and the verifier, we simulated some attacks. Here we demonstrate one such scenario, where we simulate a "split-brain" attack by the verifier, in a system with two clients. Specifically, the verifier conceals operations of each client from the other one. Figure 7.8 shows the logs of both clients as generated by the Venus client-side library. We observe that one email exchange suffices to detect the inconsistency.

7.6 Evaluation

We report on measurements obtained with Venus for clients deployed at the Technion (Haifa, Israel), Amazon S3 with the US Standard Region as the storage service, and with the verifier

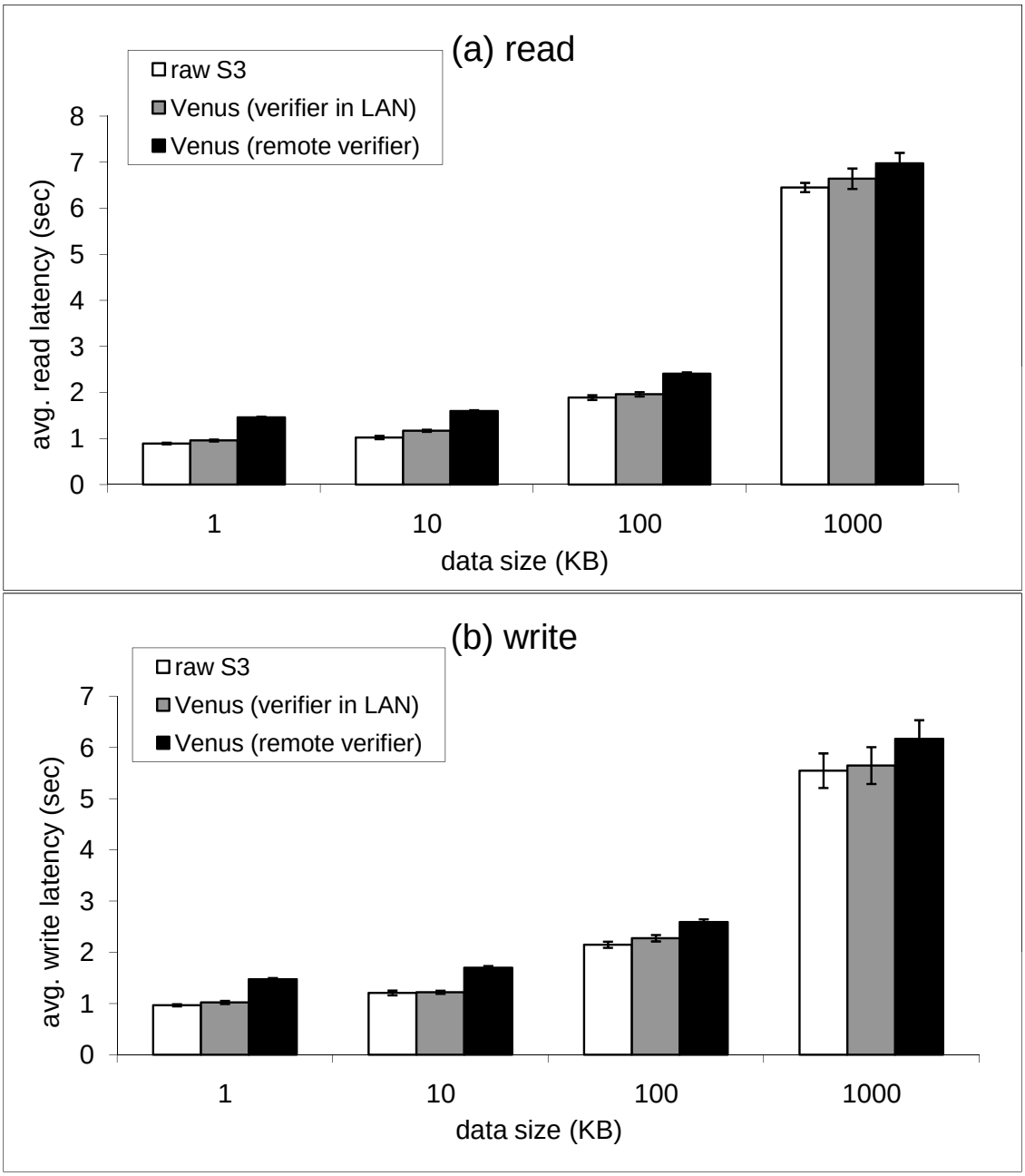

Figure 7.9: Average latency of a read and write operations, with 95% confidence intervals. The overhead is negligible when the verifier is the same LAN as the client. The overhead for WAN is constant.

deployed at MIT (Cambridge, USA) and locally at the Technion.

The clients in our experiments run on two IBM 8677 Blade Center chassis, each with 14 JS20 PPC64 blades. We dedicate 25 blades to the clients, each blade having 2 PPC970FX cores (2.2 GHz), 4GB of RAM and 2 BroadCom BCM5704S NICs. When deployed locally, the verifier runs

on a separate HS21 XM blade, Intel QuadCore Xeon E5420 with 2.5GHz, 16GB of RAM and two BroadCom NetXtreme II BCM5708S NICs. In this setting the verifier is connected to the clients by a 1Gb Ethernet.

When run remotely at MIT, the verifier is hosted on a shared Intel Xeon CPU 2.40GHz machine with 2GB RAM. In this case, clients contact the verifier using HTTP, for tunneling through a firewall, and the requests reach the Venus verifier redirected by a CGI script on a web server.

All machines run the Linux 2.6 operating system.

7.6.1 Operation latency

We examine the overhead Venus introduces for a client executing operations, compared to direct, unverified access to S3, which we denote here by "raw S3."

Figure 7.9 shows the average operation latency for a single client executing operations (since there is a single client in this experiment, operations become green immediately upon completing). The latencies are shown for raw S3, with the verifier in the same LAN as the client at the Technion, and with the remote verifier at MIT. Each measurement is an average of the latencies of 300 operations, with the 95% confidence intervals shown. We measure the average latency for different sizes of the data being read or written, namely 1KB, 10KB, 100KB and 1000KB.

Figure 7.9 shows that the latency for accessing raw S3 is very high, in the orders of seconds. Many users have previously reported similar measurements[3,4]. The large confidence intervals for 1000KB stem from a high variance in the latency (also previously reported by S3 users) of accessing big objects on S3. The variance did not decrease when an average of 1000 operations was taken.

The graphs show that the overhead of using Venus compared to using Amazon S3 directly depends on the location of the verifier. When the verifier is local, the overhead is negligible. When it is located far from the clients, the overhead is constant (450-550 ms.) for all measured data sizes. It stems from one two-way message exchange between the client and verifier, which takes two round-trip times in practice, one for establishing a TCP connection and another one for the message itself. Although we designed the verifier and the clients to support persistent HTTP connections, we found that the connection remained open only between each client and a

[3] http://bob.pythonmac.org/archives/2006/12/06/cachefly-vs-amazon-s3/

[4] http://developer.amazonwebservices.com/connect/message.jspa?messageID=93072

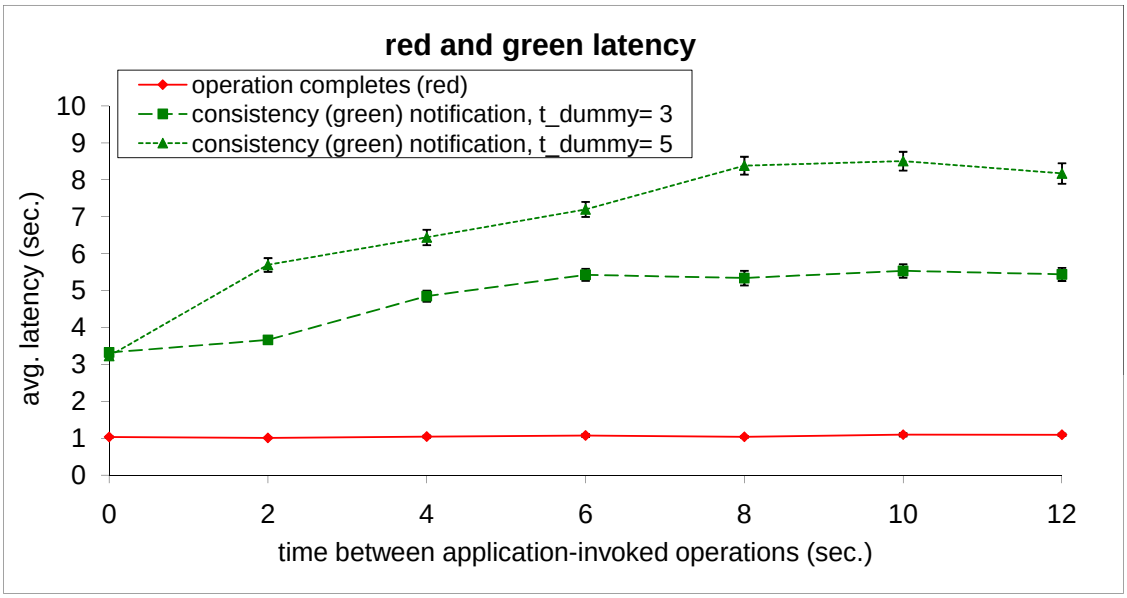

Figure 7.10: Average latency for operations with multiple clients to become red and green respectively.

local proxy, and was closed and re-opened between intermediate nodes in the message route. We suspect the redirecting web server does not support keeping HTTP connections open.

We next measure the operation latency with multiple clients and a local verifier. Specifically, we run 10 clients, 3 of which are the core set. Half of the clients perform read operations, and half of them perform writes; each client executes 50 operations. The size of the data in this experiment is 4KB. Figure 7.10 shows the average time for an operation to complete, i.e., to become red, as well as the time until it becomes green, with t_{dummy} set to 3 sec., or to 5 sec. Client-to-client communication was disabled for these experiments.

One can observe that as the time between user-invoked operations increases, the average latency of green notifications initially grows as well, because versions advance at a slower rate, until the dummy-read mechanism kicks in and ensures steady progress. Of course the time it takes for an operation to complete, i.e., to become red, is not affected by the frequency of invocations.

7.6.2 Verifier

Knowing that the overhead of our algorithm at the client-side is small, we proceed to test the verifier's scalability and throughput. Since our goal here is to test the verifier under high load, we perform this stress test with a synthetic multi-client program, which simulates many clients to the server. The simulated clients only do as much as is needed to flood the verifier with plausible

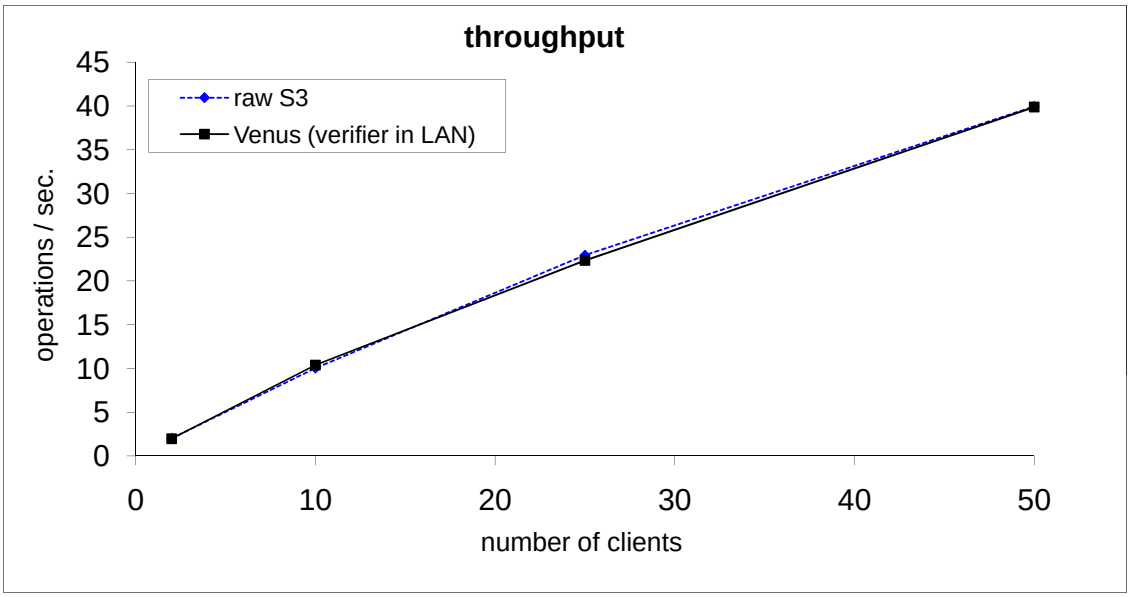

Figure 7.11: Average throughput with multiple clients.

requests.

Amazon S3 does not support pipelining HTTP operation requests, and thus, an operation of a client on S3 has to end before that client can invoke another operation. Consequently, the throughput for clients accessing raw S3 can be expected to be the number of client threads divided by the average operation latency. In order to avoid side effects caused by contention for processing and I/O resources, we do not run more than 2 client threads per each of our 25 dual-core machines, and therefore measure throughput with up to 50 client threads. As Venus clients access Amazon S3 for each application-invoked operation, our throughput cannot exceed that of raw S3, for a given number of clients. Our measurements show that the throughput of Venus is almost identical to that of raw S3, as can be seen in Figure 7.11.

Chapter 8

Conclusions

This thesis dealt with providing consistency, integrity and availability for clients collaborating through unreliable storage servers or devices. We tackled two areas where such problems arise – distributed storage and cloud storage.

For distributed storage, we defined a dynamic R/W storage problem, including an explicit liveness condition stated in terms of user interface and independent of a particular solution. The definition captures a dynamically changing resilience requirement, corresponding to reconfiguration operations invoked by users. Our approach easily carries to other problems, and allows for cleanly extending static problems to the dynamic setting. We presented DynaStore, which is the first algorithm we are aware of to solve the atomic R/W storage problem in a dynamic setting without consensus or stronger primitives. In fact, we assumed a completely asynchronous model where fault-tolerant consensus is impossible even if no reconfigurations occur. This implies that atomic R/W storage is weaker than consensus, not only in static settings as was previously known, but also in dynamic ones. Our result thus refutes a common belief, manifested in the design of all previous dynamic storage systems, which used agreement to handle configuration changes. Our main goal in DynaStore was to prove feasibility; in a recent follow-up work [72] we studied the performance tradeoffs between consensus-based solutions and consensus-free ones.

In the second part of this thesis 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 studied previously defined consistency conditions, and proved that tradi-

tional strong semantics cannot be guaranteed with an untrusted remote server. We then showed that all previously defined weaker, so called "forking", semantics inherently rule out wait-free implementations, i.e., although they protect the client when the server is faulty they hamper service availability in the common case, when the server is correct. 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 that provides weak fork-linearizable semantics with untrusted storage.

We then presented a higher-level abstraction of a fail-aware untrusted service. This notion generalizes the concepts of eventual consistency and fail-awareness to account for Byzantine faults. We realized this new abstraction in the context of an online storage service. We designed FAUST, a fail-aware untrusted storage protocol using our weak fork-linearizable protocol as an underlying layer. FAUST guarantees linearizability and wait-freedom when the server is correct, provides accurate and complete consistency and failure notifications, and ensures causality at all times.

Finally, we presented Venus, a practical service that guarantees integrity and consistency to users of untrusted cloud storage. Venus can be deployed transparently with commodity online storage and does not require any additional trusted components. Unlike previous solutions, Venus offers simple semantics and never aborts or blocks client operations when the storage is correct. We implemented Venus and evaluated it with Amazon S3. The evaluation demonstrates that Venus has insignificant overhead and can therefore be used by applications that require cryptographic integrity and consistency guarantees while using online cloud storage.

Bibliography

[1] M. Abd-El-Malek. et al. ursa minor: versatile cluster-based storage. In *FAST*, 2005.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[3] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *PODC*, pages 17–25, 2009.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. ACM CCS*, pages 598–609, 2007.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

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

[7] K. Birman, G. Chockler, and R. van Renesse. Towards a cloud computing research agenda. *SIGACT News*, 40(2), June 2009.

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

[9] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. Cryptology ePrint Archive, Report 2008/175, 2008. `http://eprint.iacr.org/`.

[10] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *CCS*, pages 187–198, 2009.

[11] C. Cachin and M. Geisler. Integrity protection for revision control. In *ACNS*, pages 382–399, 2009.

[12] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *DSN*, pages 494–503, 2009.

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

[14] C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *SIGACT News*, 40(2):81–86, 2009.

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

[16] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, 1996.

[17] G. Chockler, S. Gilbert, V. C. Gramoli, P. M. . Musial, and A. A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[18] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[19] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, 2001.

[20] G. Chockler, D. Malkhi, and D. Dolev. A data-centric approach for scalable state machine replication. In FuDiCo, LNCS Volume 2584, 2002.

[21] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, pages 189–204, 2007.

[22] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 1990.

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

[24] D. Davcev and W. Burkhard. Consistency and recovery control for replicated files. In *10th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 87–96, 1985.

[25] 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 *SOSP*, pages 205–220, 2007.

[26] C. Delporte, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 338–346, 2004.

[27] A. El Abbadi and S. Dani. A dynamic accessibility protocol for replicated databases. *Data and Knowledge Engineering*, 6:319–332, 1991.

[28] B. Englert and A. A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 454, Washington, DC, USA, 2000. IEEE Computer Society.

[29] 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.

[30] R. Friedman, M. Raynal, and C. Travers. Two abstractions for implementing atomic objects in dynamic systems. In *OPODIS*, pages 73–87, 2005.

[31] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the 17th Intl. Symp. on Distributed Computing (DISC)*, pages 259–268, June 2003.

[32] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003.

[33] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, pages 175–188, 2007.

[34] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, pages 73–86, 2007.

[35] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst.*, 4(1):32–53, 1986.

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

[37] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[38] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *ICDCS*, pages 302–309, 1990.

[39] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *CCS*, pages 584–597, 2007.

[40] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

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

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

[43] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[44] L. Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[45] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[46] L. Lamport, D. Malkhi, and L. Zhou. Brief announcement: Vertical paxos and primary-backup replication. In 28th ACM Symposium on Principles of Distributed Computing (PODC), August 2009. Full version appears as Microsoft Technical Report MSR-TR-2009-63, May 2009.

[47] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

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

[49] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[50] N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *In Symposium on Fault-Tolerant Computing*, pages 272–281. IEEE, 1997.

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

[52] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *DISC*, 2002.

[53] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, pages 105–120, 2004.

[54] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. OSDI*, 2000.

[55] M. Majuntke, D. Dobre, M. Serafini, and N. Suri. Abortable fork-linearizable storage. In *OPODIS*, pages 255–269, 2009.

[56] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, pages 569–578, 1997.

[57] T. Marian, M. Balakrishnan, K. Birman, and R. van Renesse. Tempest: Soft state replication in the service tier. In *DSN*, pages 227–236, 2008.

[58] J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

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

[60] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[61] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In *DISC*, pages 164–178, 2000.

[62] A. Milani. Causal consistency in static and dynamic distributed systems. PhD Thesis, "La Sapienza" Università di Roma, 2006.

[63] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, 2006.

[64] A. Oprea and M. K. Reiter. On consistency of encrypted files. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 254–268, 2006.

[65] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM CCS*, pages 437–448, 2008.

[66] J. Paris and D. Long. Efficient dynamic voting algorithms. In *13th International Conference on Very Large Data Bases (VLDB)*, pages 268–275, 1988.

[67] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, 2003.

[68] R. Rodrigues and B. Liskov. Reconfigurable byzantine-fault-tolerant atomic memory. In *Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, St. John's, Newfoundland, Canada, July 2004. Brief Announcement.

[69] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[70] H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *Proceedings of Asiacrypt 2008*, volume 5350 of *LNCS*, pages 90–107. Springer-Verlag, Dec. 2008.

[71] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *the ACM Cloud Computing Security Workshop (CCSW)*, 2010.

[72] A. Shraer, J.-P. Martin, D. Malkhi, and I. Keidar. Data-centric reconfiguration with network-attached disks. In *the 4th ACM SIGOPS/SIGACT Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, July 2010.

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

[74] 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*, pages 172–182, 1995.

[75] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI 04)*, December 2004.

[76] J. Yang, H. Wang, N. Gu, Y. Liu, C. Wang, and Q. Zhang. Lock-free consistency control for Web 2.0 applications. In *WWW*, pages 725–734, 2008.

[77] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symp. on Principles of Distributed Computing (PODC-16)*, pages 63–71, August 1997.

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

נוספים הדורשים רמת אבטחה חזקה יותר מזו הניתנת על ידי הענן עצמו ושלא יכולים לבטוח בענן באופן עיוור, לשקול להתחיל ליהנות מהיתרונות שנובעים משימוש בענן.

בפרק 4 אנו מגדירים את מודל האחסון הלא אמין. בפרק 5 אנו מוכיחים שבלתי אפשרי להבטיח ללקוחות קונסיסטנטיות חזקה של מידע הנשמר על שרת לא אמין, וחוקרים סמנטיקות חלשות יותר שכן ניתן להבטיח במצבים כאלה, גם כאשר השרת תקול [18]. אנו מזהים מגבלה חשובה בכל הפתרונות הקודמים: הם מקריבים את זמינות המערכת במצב הרגיל, כאשר שרת האחסון תקין. למשל, אם לקוח אחד נופל בזמן שהוא מעדכן מידע שמור, אף לקוח לא יכול יותר לקרוא את המידע. אנו מוכיחים שבעיה זו אינהרנטית לכל הבטחות הקונסיסטנטיות שהוגדרו למודל זה [14, 15, 18].

בפרק 6 אנו מציגים את FAUST [14], שירות שעושה שימוש בטכניקות ממערכות מבוזרות ומקריפטוגרפיה ומבטיח סמנטיקה ברורה ללקוחות גם כאשר ספק האחסון תקול. במצב הרגיל, כאשר שרת האחסון תקין, FAUST מבטיח ללקוח גם קונסיסטנטיות חזקה וגם זמינות מירבית של המידע.

למרות כל זאת, FAUST ועבודות קודמות לא ניתנות לשימוש עם שירותי אחסון ענן קיימים: מערכות אלו דורשות הרצת פרוטוקולים מורכבים בין הלקוחות לשירות האחסון ואילו שירותי האחסון המוצעים כיום כוללים בדרך כלל ממשק פשוט לקריאה וכתיבה של המידע. כדי לפתור בעיה זו, פיתחנו וממשנו את Venus [78], המוצג בפרק 7. Venus הוא שירות הבודק את השלמות והקונסיסטנטיות של מידע המאוחסן בענן. Venus ניתן לשימוש עם שירותי אחסון ענן קיימים, ואינו מצריך שינויים כלשהם בשירותים אלה. בנוסף וחשוב לא פחות, Venus מציע ללקוח סמנטיקה פשוטה בהרבה מ-FAUST ומהפתרונות הקודמים, מה שמוסיף עוד יותר לשימושיותו. Venus יכול להתמודד עם תקלות רבות החל משיבוש פשוט של המידע וכלה בתקלות ביזאנטיות בענן. השתמשנו ב-Venus עם שירות אחסון הענן Amazon S3 ואנו מראים ש-Venus הינו scalable ולא מוסיף תקורה רבה.

תקציר תזה

מערכות אחסון ארגוניות הינן מערכי דיסקים (כלומר "ארונות אחסון") מונוליתיים והן כיום בשימוש נרחב בסביבות ארגוניות. מערכות אלה מורכבות מחומרה ייעודית יקרה ומספקות אמינות גבוהה גם במצבים קיצוניים ובלתי סבירים. למרות זאת, ישנם מספר חסרונות חשובים למערכות כאלה. קודם כל, ריכוז כל נתוני החברה בארון אחסון בודד (או מספר קטן של ארונות כאלה) גורם לעיתים קרובות ליצירת צווארי בקבוק בפורטי הקלט/פלט והבקרים של מערכת האחסון. עניין נוסף הוא מוגבלות ההרחבה של ארונות האחסון, ויצרנים רבים של מערכות אחסון מציעים פתרונות אחסון נפרדים לגמרי עבור גדלים שונים של הארגון הלקוח (למשל IBM ו-HP מציעים מערכות "entry level", "midrange" ו-"high-end"). בנוסף לכך, מערכות אלו יקרות מאד.

ארכיטקטורות אחסון מבוזרות מציעות אלטרנטיבה זולה למערכות אחסון ארגוניות והן ניתנות להרחבה בקלות. מערכות מבוזרות אלה מורכבות מהרבה שרתים (או התקני אחסון) לא אמינים והאמינות מסופקת על ידי שיכפול המידע. אלטרנטיבה פופולארית נוספת הינה אחסון ב-"ענן" (cloud), המוצע באינטרנט על ידי גופים רבים. תזה זו עוסקת בשתי בעיות אמינות בפתרונות אלו. הבעיה הראשונה קשורה לשינויי קונפיגורציה במערכות אחסון מבוזרות: מספרם הרב של השרתים הבלתי אמינים דורש תמיכה בשינויים דינאמיים כלומר בהוצאת שרתים תקולים מהמערכת והכנסת שרתים חדשים למערכת. הבעיה השנייה היא חוסר אמינות בשירותי אחסון מרוחק (למשל אחסון "ענן"), כלומר קיים צורך לספק אמינות ללקוח המשתמש באחסון ענן אשר אינו אמין ואף עלול להיות תקול באופן כלשהו.

למרות שאלגוריתמי שכפול במערכות מבוזרות נלמדו באופן נרחב בעבר, שכפול לבדו מספק עמידות לנפילות שהינה מוגבלת מטבעה – במערכת אסינכרונית אין אפשרות להתמודד עם נפילות של יותר ממיעוט משתתפים השומרים עותק של המידע [6]. שינוי קונפיגורציה, כלומר שינוי אוסף המחשבים השומרים עותק של המידע, מגביר את עמידות המערכת לנפילות: נניח שהמידע משוכפל ב-5 שרתים במערכת אסינכרונית. בהתחלה נוכל לעמוד בנפילת שני שרתים בלבד. ברור שעל ידי הוספת שרתים למערכת נוכל לעמוד בנפילות נוספות. אך מעניין לשים לב שגם על ידי הוצאה בלבד של שרתים מהמערכת נוכל להגביר את עמידות המערכת לנפילות. לדוגמא: אם שני שרתים קורסים ואנו מוציאים אותם מהמערכת, המערכת שלנו כוללת כעת שלושה שרתים. אחד מהם (כלומר מיעוט) יכול לקרוס, כלומר נוכל לעמוד בשלוש נפילות בסך הכול ונעקוף את "מחסום המיעוט". ברור שהנפילות צריכות לקרות בהדרגה וניתן להתיר נפילות נוספות רק לאחר שרוב השרתים במערכת החדשה שומרים עותק של המידע.

כדי לשמור על אמינות כאשר שינויים כדוגמת אלו קורים ולהימנע מ"פיצול" המערכת (למשל מצב בו לקוחות שונים יעבדו עם אוסף שרתים שונה), הכרחי להבטיח תיאום של השינויים, למשל כאשר מספר שינויי קונפיגורציה קורים במקביל. הפתרונות הקיימים כיום הינם ריכוזיים או משתמשים באלגוריתמי סנכרון חזקים (למשל הסכמה מבוזרת - קונצנזוס) בין השרתים כדי להסכים על כל שינוי במערכת. למעשה, האמונה הרווחת הייתה ששינויי קונפיגורציה דורשים שימוש בקונצנזוס ולכן (כמו קונצנזוס) אינם ניתנים לביצוע במערכות אסינכרוניות. בפרק 3 אנו מפריכים אמונה זו ומציגים את DynaStore [3], אלגוריתם אחסון אסינכרוני מבוזר לחלוטין המאפשר שינויי קונפיגורציה בנוסף לקריאה וכתיבה של המידע השמור.

אלטרנטיבה נוספת למערכות אחסון ארגוני היא אחסון ב-"ענן" והיא מוצעת כיום על ידי חברות רבות באינטרנט. אחסון בענן ושירותי ענן נוספים מאפשרים למשתמשים לשתף פעולה ולגשת למידע מכל מקום. בשונה ממערכות אחסון ארגוני, שירותי ענן מאפשרים ללקוחות לקנות משאבים כאשר אלו נדרשים (on demand) , ולא לשלם מראש כמו במערכות אחסון ארגוניות. כלומר, הלקוח משלם בכל רגע נתון רק עבור המשאבים שבשימוש ברגע זה. למרות זאת, כפי שאנו מסבירים בפרק 4, אמינות שירותי ענן מעוררת דאגה רבה אצל המשתמשים [17]. חשוב למשל להבטיח את שלמות המידע שנשמר בענן ולוודא שלקוחות המשתפים מידע יראו תמונה קונסיסטנטית של המידע. מעט מאד מחקרים נעשו בעבר על סוגיות אלו. בנוסף, התעשייה מתמקדת באבטחת הספק, כלומר הענן, ולא בהגנת הלקוח מתקלות אפשריות של הענן.

בעבודה זו אנו מפתחים כלים וסמנטיקה המאפשרים ללקוחות המשתמשים בשירותי אחסון בענן לעקוב אחר שירות האחסון אותו הם מקבלים, ולוודא שהענן מתנהג כצפוי. עבודה זו מאפשרת למגוון יישומים אשר כבר משתמשים בענן ליהנות מאבטחה מוגברת. חשוב לא פחות, היא יכולה לעודד יישומים

המחקר נעשה בהנחיית פרופ' עידית קידר בפקולטה להנדסת חשמל – טכניון

אני מודה לטכניון – מכון טכנולוגי לישראל על התמיכה הכספית הנדיבה בהשתלמותי

שיתוף נתונים אמין בעזרת שרתי אחסון לא אמינים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

אלכסנדר שרייר

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
אלול תש"ע חיפה ספטמבר 2010

שיתוף נתונים אמין בעזרת שרתי אחסון לא אמינים

אלכסנדר שרייר