Optimal Resilience for Erasure-Coded Byzantine Distributed Storage

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

Stefano Tessaro*
ETH Zurich
Department of Computer Science
CH-8092 Zurich, Switzerland
tessaros@student.ethz.ch

February 6, 2005

Abstract

We analyze the problem of efficient distributed storage of information in a message-passing environment where both less than one third of the servers, as well as an arbitrary number of clients, might exhibit Byzantine behavior, and where clients might access data concurrently. In particular, we provide a simulation of a multiple-writer multiple-reader atomic read/write register in this setting which uses erasure-coding for storage-efficiency and achieves optimal resilience. Additionally, we give the first implementation of non-skipping timestamps which provides optimal resilience and withstands Byzantine clients; it is based on threshold cryptography.

1 Introduction

Recent advances in the development of networked storage systems, such as Network-Attached Storage (NAS), Object Storage [14, 4], and Storage-Area Networks (SAN), combined with the increasing availability of fast networks, have made it very attractive to store large amounts of information in a *distributed storage system*. Such systems may use replication in order to enhance their security and fault-tolerance.

We consider a set of n *servers*, implementing the storage system itself, and a possibly unbounded set of *clients*, accessing the storage system for reading and writing data. Servers and clients communicate by exchanging messages over a fully connected *asynchronous* network. This model is suitable for heterogeneous and wide-area networks, and, furthermore, avoids timing assumptions, which may otherwise become a vulnerability of the system.

These servers are subject to failures, and the system has to be designed in order to tolerate them. Moreover, we do not want to limit ourselves to "benign" crashes of the *parties*, i.e., servers and clients, because a malicious entity might take control over some of them and launch a coordinated attack. For this reason, we consider *Byzantine failures* and assume that up to t servers and any number of clients may deviate from the protocol in an arbitrary way.

A distributed storage system needs to handle *concurrent* access by clients. A good abstraction for a concurrent storage system is a *multiple-writer multiple-reader read/write register* [19]. Such a register is a shared object which can be used by many clients in order to store and retrieve a value. A complete storage system can be modeled as an array of these registers. Thus, the problem of implementing a storage system can be formalized as the problem of simulating a multiple-writer multiple-reader read/write register by the servers. Despite the simplicity of such a register, it is not immediately clear how it should

*Work done at IBM Research, Zurich Research Laboratory.

behave if accessed concurrently. Lamport [19] has introduced three types of consistency conditions for registers: *safety*, *regularity*, and *atomicity*. Atomicity is the strongest one, requiring every execution to appear sequential, and it is the one considered here.

Previous fault-tolerant simulations of registers in a message-passing environment are based on *replication* of data [3, 21, 23, 5], where each server keeps a complete copy of the data. An approach that wastes less resources is based on *information dispersal* [25] and *erasure coding*. Here, the data is split into blocks such that each server stores exactly one block, and the information stored at the honest servers is enough to reconstruct the original data.

Most prior solutions for information dispersal in the Byzantine-failure model do not support concurrent access to the stored data. Only the recent protocol of Goodson et al. [15] addresses this question, but still allows malicious clients to write inconsistent data to the servers, and recovering from such inconsistent writes might be expensive. On the other hand, Cachin and Tessaro [9] define *verifiable information dispersal*, where the servers will detect if inconsistent information is stored. This avoids expensive operations for recovery, but their protocol does not allow for concurrent updates.

In this paper, we provide a new fault-tolerant simulation of an atomic register for data that is not self-verifying [23]. We give a definition of an atomic register simulation protocol in an asynchronous message-passing model, where both servers and clients are subject to Byzantine faults. We also give the first protocol for storage-efficient distributed simulation of a *multiple-writer multiple-reader read/write register* that provides *atomic semantics* and *optimal resilience*, i.e., tolerates the failure of up to one third of the servers and of an arbitrary number of clients [23]. It follows the "listeners' pattern" proposed by Martin et al. [23], but uses asynchronous verifiable information dispersal [9] and asynchronous reliable broadcast for tolerating Byzantine clients. Our protocol improves the storage and communication efficiencies of Martin et al.'s protocol for the simulation of atomic registers [23] and improves the resilience and the storage complexity of Goodson et al.'s solution for erasure-encoded storage [15], and avoids potentially expensive recovery operations. Like some of the previous work, our protocol uses interaction among the servers.

The challenge with using erasure coding in the concurrent setting is that no server stores the entire data, and in order to read correctly, a client must receive data blocks belonging to the same data item from multiple servers. A possible way for keeping track of multiple concurrently written versions of the data is provided by *(logical) timestamps*: Whenever a new data item is written, it receives a higher timestamp. Malicious parties, however, may be able to mount a denial-of-service attack by making timestamps arbitrarily large. Bazzi and Ding [5] considered this problem and solved it by introducing so-called *non-skipping timestamps*, where the value of every timestamp is bounded by the number of writes that have been executed previously in the system and where no timestamp value can be "skipped." We provide an improved implementation of non-skipping timestamps based on threshold signatures that withstands the Byzantine failure of clients and of up to one third of the servers. Our solution uses cryptographic digital signatures, but key management is much easier than in previous solutions: We require the single public key of the service to be stored at the clients, but no client keys at the servers.

1.1 Related work

Rabin's work [25] introduces the concept of information dispersal algorithms (IDA) for splitting large files, but does not address protocol aspects for implementing IDA in distributed systems.

IDA is extended by Krawczyk [18] using a technique called *distributed fingerprinting* in order to ensure the integrity of data in case of alterations of the stored blocks by malicious servers. The same idea is subsequently improved by Alon et al. [1, 2].

Garay et al. [13] propose an information dispersal scheme for *synchronous* networks that tolerates Byzantine server failures. Their model does not allow Byzantine clients, even though some attacks are tolerated. Because of its inherent synchrony, this protocol cannot be translated to an asynchronous network.

A solution for erasure-coded storage in an asynchronous network with robustness against Byzantine servers *and* clients has recently been proposed by Goodson et al. [15]. Their scheme is able to detect inconsistently written data only at read-time; the content of the storage system must then be rolled back to the last correctly written state. The major drawback of this approach is that retrieving data can be very inefficient in the case of several faulty write operations, and that consistency depends on a correct client. The protocol requires $t < \frac{n}{4}$ and ensures atomic semantics.

Cachin and Tessaro [9] introduce the concept of verifiable information dispersal in asynchronous networks, which guarantees that, once the storage a file has been accepted by the servers, the data is stored consistently. This is analogous to verifiability in secret sharing [11, 12, 24].

Among these cited works, only Goodson et al. [15] address the question of concurrent access to data for systems based on information dispersal and with Byzantine failures of clients and servers.

Many simulations of read/write registers in the message-passing model based on replication have been given, most of them considering only a bounded number of client processes or being restricted to crash failures (see e.g. the work of Attiya et al. [3]). *Phalanx* [21] is a practical system for survivable coordination that also provides a simulation of *safe* read/write registers based on replication and tolerates Byzantine failures of both clients and $t < \frac{n}{4}$ servers.

Martin et al. [23] proposed a replication-based simulation of an atomic register in the message-passing model, where $t < \frac{n}{3}$ servers might be Byzantine. A drawback of this solution is the ability of faulty servers to make timestamps as large as they wish. Recently, Bazzi and Ding [5] have improved this solution in order to implement non-skipping timestamps at the price of lower resilience, supporting the Byzantine failure of $t < \frac{n}{4}$ servers. Our protocol closes this gap and achieves both: optimal resilience and non-skipping timestamps. Furthermore, it is the first solution for the case where an arbitrary number of Byzantine clients may collude with the Byzantine-faulty servers.

1.2 Outline of this paper

Section 2 presents the model and introduces our tools, in particular, the system model, cryptographic primitives like threshold signatures, and information dispersal schemes. In Section 3, we define a simulation protocol for an atomic register, present a protocol that implements it, and prove it correct. Finally, we extend the protocol to provide non-skipping timestamps, and give a complexity analysis.

2 Preliminaries

2.1 System model

We use a model which is equivalent to the one of Cachin et al. [8]. The network consists of a set of *servers* $\{P_1, \dots, P_n\}$ and a set of *clients* $\{C_1, C_2, \dots\}$, which are all probabilistic interactive Turing machines (PITM) with running time bounded by a polynomial in a given security parameter κ. Servers and clients together are called *parties*. There is an *adversary*, which is a PITM with running time bounded by a polynomial in κ. Servers and clients can be controlled by the adversary. In this case, they are called *corrupted*, otherwise they are called *honest*. An adversary that controls up to t servers is called *t-limited*. We are not assuming any bounds on the number of clients that can be corrupted. The adversary is *static*, that is, it must choose the parties it corrupts before starting the protocol. Additionally, there is an initialization algorithm, which is run by some trusted party before the system actually starts.

Every pair of servers is linked by a *secure asynchronous channel* that provides privacy and authenticity with scheduling determined by the adversary. Moreover, every client and every server are linked by a secure asynchronous channel. We restrict the adversary such that every *run* of the system is *complete*, i.e., every message sent by an honest party and addressed to another honest party is delivered before the adversary terminates. We refer to this property when we say that a message is "eventually" delivered.

Whenever the adversary delivers a message to an honest party, this party is *activated*. In this case, the message is put in a so called input buffer, the party reads then the content of its buffer, performs some computation, and generates one or more response messages, which are written on the output tape of the party.

Protocols can be invoked either by the adversary, or by other protocols. Every protocol instance is identified by a unique string *ID*, called the *tag*, which is chosen arbitrarily by the adversary if it invokes the protocol, or which contains the tag of the calling protocol as a prefix if the protocol has been invoked by some other protocol. There may be several threads of execution for a given party, but only one of them is allowed to be active concurrently. When a party is activated, all threads are in *wait states*, which specify a condition defined on the received messages contained in the input buffer. If one or more threads are in a wait state whose condition is satisfied, one of these threads is scheduled (arbitrarily) and this thread runs until it reaches another wait state. This process continues until no more threads are in a wait state whose condition is satisfied. Then, the activation of the party is terminated and the control returns to the adversary.

The memory of each party consists of *local* and *global variables*. The former are used during the execution of a single thread, and erased at the end of the execution of the thread, whereas the latter are associated to a certain protocol instance and accessible to all threads of this instance.

We distinguish between *local events*, which are either *input actions* (that is, messages of the form $(\mathit{ID}, \mathtt{in}, \mathit{type}, \ldots)$) or *output actions* (messages of the form $(\mathit{ID}, \mathtt{out}, \mathit{type}, \ldots)$), and other *protocol messages*, which are ordinary protocol messages of the form $(\mathit{ID}, \mathit{type}, \ldots)$ to be delivered to other parties. All messages of this form that are generated by honest parties are said to be *associated* to the protocol instance *ID*.

The interaction between the adversary and the honest parties defines a logical sequence of events, which we use as implicit *global clock*. We refer to it by saying that an event takes takes place at a certain *point in time*.

We use the following syntax for specifying our protocols. To enter a wait state, a thread executes a command of the form **wait for** *condition*. There is a global implicit **wait for** statement that every protocol instance repeatedly executes: it matches any of the *conditions* given in the clauses of the form **upon** *condition block*.

The following complexity measures are used in the analysis of protocols. Complexities are always defined with respect to a single instance of a protocol.

- The *message complexity* of a protocol is defined as the number of messages associated to an instance of the protocol.

- The *communication complexity* of a given protocol is defined as the bit length of all messages associated to an instance of the protocol.

- The *storage complexity* of a protocol is defined as the size of the global variables associated to an instance of the protocol.

Finally, a function $\epsilon(\kappa)$ is called *negligible* if for all $c > 0$ there exists a κ_0 such that $\epsilon(\kappa) < \frac{1}{\kappa^c}$ for all $\kappa > \kappa_0$.

2.2 Cryptographic tools

We will make use of a *non-interactive threshold signature scheme*. A non-interactive (n,t)-threshold signature scheme $\mathcal{TSS}$ consists of the following algorithms:

- A *key generation algorithm* $\mathsf{generate}(\kappa, n, t)$ which returns a *public key* PK, as well as a *private key share* SK_j and a *local verification key* VK_j for each server P_j, where $j \in [1, n]$.

- A *signing algorithm* $\mathsf{sign}(m, \mathit{PK}, \mathit{SK}_j)$, where m is some message, which returns a *signature share* μ_j of server P_j on m.

- A *share verification algorithm* $\mathsf{verify\text{-}share}(m, \mu_j, PK, VK_j)$ that returns a boolean value. We say that a signature share μ_j from P_j on m is *valid* if $\mathsf{verify\text{-}share}(m, \mu_j, PK, VK_j) = \mathtt{true}$, and *invalid* otherwise.

- A *share combining algorithm* $\mathsf{combine}(m, \Sigma, PK, [VK_1, \ldots, VK_n])$, where Σ is a set of at least $t+1$ valid signature shares on m, which outputs a *signature* σ on m.

- A *signature verification algorithm* $\mathsf{verify}(m, \sigma, PK)$ which returns a boolean value. We say that a signature σ on m is *valid* if $\mathsf{verify}(m, \sigma, PK) = \mathtt{true}$, and *invalid* otherwise.

Assume the adversary plays the following game. Initially, a trusted dealer runs the key generation algorithm and gives to each server P_j the public key PK, all local verification keys $VK_1, \ldots, VK_n$, and its private key share SK_j. The adversary then decides which servers it corrupts. Subsequently, the adversary can submit messages to the honest servers, and each honest server answers by providing a signature share on the submitted message to the adversary. Finally, given at least $t+1$ signature shares for the same message, the adversary may combine them into a valid signature on the message.

We say that the scheme satisfies *robustness* if it is computationally infeasible for the adversary to produce $t+1$ valid signature shares such that the output of the share combining algorithm is not a valid signature. Moreover, the scheme satisfies *non-forgeability* if it is computationally infeasible for the adversary to output a valid signature on a message that was never submitted as a signing request to any honest server. A practical scheme satisfying these requirements (in the *random-oracle* model) has been proposed by Shoup [26].

Additionally, a *collision-resistant hash function* is a function $H : \{0,1\}^* \to \{0,1\}^h$ with the property that the adversary cannot generate two distinct strings x and x' with $H(x) = H(x')$, except with negligible probability. With a slight abuse of notation, we denote by $|H|$ the bit-size of the range of the hash function, that is, $|H| := h$. In practice, H could be implemented by SHA-1 (in this case, $|H| = 160$).

2.3 Information dispersal

Information dispersal has been introduced by Rabin [25], and is based on the concept of an *erasure code*. A (n,k)-erasure code $\mathcal{C}$ is given through an *encoding algorithm*, encode, and a *decoding algorithm*, decode, such that the following holds:

- Given a value[1] F, $\mathsf{encode}(F)$ produces a vector $[F_1, \ldots, F_n]$, where $|F_j| \approx \frac{|F|}{k}$ for all $j \in [1,n]$.

- Given a set of k pairs $\mathcal{A} := \{(j_1, F_{j_1}), \ldots, (j_k, F_{j_k})\}$, where $j_1, \ldots, j_k$ are distinct elements from $\{1, \ldots, n\}$, $\mathsf{decode}(\mathcal{A})$ produces a value F'.

Moreover, assume $[F_1, \ldots, F_n]$ is the vector produced by $\mathsf{encode}(F)$. Then, given any k components (or *blocks*) F_j, with the corresponding indices j, decode must reconstruct the original F. That is, every subset of k components of the encoded value is enough to reconstruct the value. For more details, the reader is referred to [25, 9, 6].

In the following, we will make use of a slightly modified version of the dispersal protocol in the AVID-RBC scheme of Cachin and Tessaro [9], called Disperse. (A review of *asynchronous* verifiable information dispersal is provided in Appendix A.) Our protocol makes use of an (n,k)-erasure code $\mathcal{C}$ for $k \leq n - t$ and of a collision-resistant hash function H.

Protocol Disperse is invoked at an honest client C_i through an input action (*ID*, `in`, `disperse`, F), containing a value F. In this case we say that client C_i *disperses* F. Assuming $[F_1, \ldots, F_n] := \mathsf{encode}(F)$, the protocol behaves also like an asynchronous reliable broadcast (see Appendix B) of the vector $\mathbf{D} := [D_1, \ldots, D_n]$, where $D_j := H(F_j)$ for $j \in [1,n]$. In particular, each honest server P_j

[1] Because of our strong bias toward data storage, we usually think of values as *files*.

outputs a message $(ID, \texttt{out}, \texttt{stored}, \mathbf{D}, i, F_j)$, where $\mathbf{D}$ is the vector delivered by reliable broadcast, i is the identifier of the client that started the dispersal, and F_j is an erasure-code block satisfying $H(F_j) = D_j$. In this case we say that P_j *completes the dispersal with* $[\mathbf{D}, i, F_j]$. Since the reliable broadcast provides agreement on the delivered value, all honest servers complete the dispersal with the same $\mathbf{D}$, and the following holds, except with negligible probability: There exists a value F' with encoding $[F'_1, \ldots, F'_n]$ such that $[H(F'_1), \ldots, H(F'_n)] = \mathbf{D}$ and, for each server P_j having completed the dispersal, $F_j = F'_j$. Moreover, if C_i is honest, then F' is the value F that it has originally dispersed (except with negligible probability).

The communication complexity of the Protocol Disperse is $\mathcal{O}\left(n|F| + n^3|H|\right)$. The $n^3|H|$ term can be reduced to $n^2 \log n|H|$ by using *hash trees* instead of hash vectors. For the sake of clarity, we will avoid using this optimization in the following, even though the reader should be aware of the fact that this optimization can be easily used in the proposed protocols in this paper, and we will indeed take advantage of this fact in the complexity results.

3 Byzantine simulation of atomic shared registers

In this section, we first define protocols for the simulation of multiple-writer multiple-reader atomic read/write registers (or "atomic register" for short) in the message passing model. After that, we give our information-dispersal-based simulation of an atomic register, analyze it, and improve it to provide non-skipping timestamps. Finally, we discuss the communication and storage complexities of our protocols.

3.1 Definitions

Recall that a *multiple-writer multiple-reader atomic read/write register* [19] is a concurrent object that supports a set of values $\mathcal{F}$ with an initial value $F_{init} \in \mathcal{F}$, and provides read and write operations, both of which can be invoked by an arbitrary number of clients. Every operation is required to eventually terminate. An implementation of such a register must be *wait-free*, i.e., ensure that every operation of an honest client terminates independently of the speed of other clients accessing the register. We assume all values F stored in the register have the same size $|F|$.

In the following, we want to define protocols for simulating a shared register with *atomic semantics* that can also be accessed by corrupted clients. Atomic semantics requires that for every execution, there exists a total order such that the view of the clients is consistent with an execution where the operations are executed sequentially according to the total order. Operations performed by Byzantine clients are not necessarily well-formed and could potentially modify the state of the register arbitrarily. For this reason, in a pure shared-memory model, there is no way to determine all operations altering the state of the register, since we have access only to what honest client observe.

The philosophy of our definition is to exploit the capabilities of the message-passing model in order to expose all operations that affect the state of the register. This concerns read and write operations invoked at honest clients, but also write operations on behalf of corrupted clients, which modify the value of the register. We capture them by requiring the servers to signal the completion of every write. Our approach guarantees that the view of the honest clients is always consistent, even though corrupted clients are active concurrently. Previous definitions of atomic semantics in a setting with Byzantine clients have been given by Malkhi, Reiter and Lynch [22] with weaker guarantees in a pure shared-memory model, and by Goodson et al. [15, 16], who adopt an ad-hoc approach without such an explicit signal.

A protocol Π executed by n servers $P_1, \ldots, P_n$ and an unbounded number of clients $C_1, C_2, \ldots$ for the simulation of an atomic register defines two types of operations available to clients: *write operations* and *read operations*. For notational convenience, we identify each operation by a bit string *oid*, called the *operation identifier*, which is chosen by the caller of the operation (that is, in our model, by the adversary) and must be unique in the system. We assume the servers simulate more than one register concurrently and identify every register by a tag *ID*.

A *write operation* (or *read operation*) for register *ID* is *invoked* at an honest client C_i with operation identifier *oid* when it receives an input action $(\mathit{ID}, \texttt{in}, \texttt{write}, \mathit{oid}, F)$ (or $(\mathit{ID}, \texttt{in}, \texttt{read}, \mathit{oid})$, respectively) from the adversary. In the first case we say the a client C_i *writes* F to register *ID* with operation identifier *oid*, and in the second case we say that client C_i *reads* from register *ID* with operation identifier *oid*. Whenever an operation is invoked at a client, it starts executing the operation until it generates an output action, and we say that the operation *terminates*. In particular, a read operation for *ID* with operation identifier *oid* generates an output action $(\mathit{ID}, \texttt{out}, \texttt{read}, \mathit{oid}, F)$; in this case, we say that client C_i *reads* value F from register *ID* with operation identifier *oid*, or that the read *returns* value F. A write operation for *ID* returns an output action $(\mathit{ID}, \texttt{out}, \texttt{ack}, \mathit{oid}, F)$; in this case we say C_i *has written* value F to register *ID* with operation identifier *oid*.

An honest server may *accept a write* to register *ID* with operation identifier *oid* by generating an output action $(\mathit{ID}, \texttt{out}, \texttt{write-accepted}, \mathit{oid})$. We say that a write to register *ID* with operation identifier *oid takes effect* if at least one honest server accepts the write with operation identifier *oid*. Every honest party must generate at most one output action for every tag *ID* and operation identifier *oid*.

Given an adversary A, let $\mathcal{R}_A^{\mathit{ID}}$ and $\mathcal{W}_A^{\mathit{ID}}$ be the set of operation identifiers of terminating read and write operations, respectively, which are invoked at honest clients with tag *ID* in a run of the system with adversary A. Let $\mathcal{E}_A^{\mathit{ID}}$ be the set of operation identifiers of write operations which take effect with tag *ID*. Note that these sets are random variables whose distributions depend on the coin tosses of the adversary A and of the honest parties. Moreover, $\mathcal{E}_A^{\mathit{ID}}$ might also contain identifiers of operations not invoked at honest clients, but performed by the adversary through corrupted clients.

We say that an invocation (or the termination of an operation) *takes place* at the point in time when the corresponding input (or output) action is delivered to (or generated by) the party. According to our model, no two events can take place at the same point in time.

Finally, we say that for two operations with identifiers $\mathit{oid}_1, \mathit{oid}_2 \in \mathcal{R}_A^{\mathit{ID}} \cup \mathcal{W}_A^{\mathit{ID}}$, the first operation *precedes* the second one in a run of the system if the termination of the first operation takes place at an earlier point in time than the invocation of the second one. Two operations are called *concurrent* if none of them precedes the other one. With a slight abuse of notation, we sometimes say that an operation identifier oid_1 precedes another operation identifier oid_2 if this holds for the corresponding operations.

The following definition captures the concept of an atomic register simulation protocol.

Definition 1. A protocol Π, providing the interface described above, is an *atomic register simulation protocol* if, for all t-limited adversaries A and all tags *ID*, the following properties hold, except with negligible probability:

Liveness: If an operation is invoked at an honest client C_i with tag *ID* and operation identifier *oid*, then the operation eventually terminates, that is, $\mathit{oid} \in \mathcal{R}_A^{\mathit{ID}} \cup \mathcal{W}_A^{\mathit{ID}}$. Moreover, every write to register *ID* invoked at an honest client eventually takes effect, that is, $\mathcal{W}_A^{\mathit{ID}} \subseteq \mathcal{E}_A^{\mathit{ID}}$.

Correctness: There exists a total order $<$ over $\mathcal{R}_A^{\mathit{ID}} \cup \mathcal{E}_A^{\mathit{ID}}$ such that

(i) for every pair $\mathit{oid}_1, \mathit{oid}_2 \in \mathcal{R}_A^{\mathit{ID}} \cup \mathcal{E}_A^{\mathit{ID}}$ such that $\mathit{oid}_1, \mathit{oid}_2 \in \mathcal{R}_A^{\mathit{ID}} \cup \mathcal{W}_A^{\mathit{ID}}$, if the operation with identifier oid_1 precedes the operation with identifier oid_2, then $\mathit{oid}_1 < \mathit{oid}_2$;

(ii) for every read operation with identifier $\mathit{oid}_r \in \mathcal{R}_A^{\mathit{ID}}$ returning some value F, let oid_w be the largest element of $\mathcal{E}_A^{\mathit{ID}}$ (according to $<$) such that $\mathit{oid}_w < \mathit{oid}_r$; then, every read operation with identifier $\mathit{oid}'_r \in \mathcal{R}_A^{\mathit{ID}}$, for which $\mathit{oid}_w < \mathit{oid}'_r < \mathit{oid}_r$, returns F; moreover, if $\mathit{oid}_w \in \mathcal{W}_A^{\mathit{ID}}$, then the write operation with identifier oid_w writes F.

In order to be formally correct, we would also have to take care of reading the initial value before any write has taken effect. We avoid to deal with this special case by assuming that for all tags *ID*, there is some write in the system that precedes all other operations and that writes $F_{\mathit{init}} \in \mathcal{F}$.

For an atomic register simulation protocol where values with a fixed size $|F|$ are stored, we define the *storage blow-up* as the ratio of the storage complexity of the protocol and $|F|$.

3.2 Simulation of an atomic register

In this section, we present an atomic register simulation protocol Atomic in the model of Section 2.1. The detailed description of the write and read operations is given in Figures 1 and 2, respectively.

Our protocol relies on Protocol $\mathsf{Disperse}$ as presented in Section 2.3, which makes use of a collision-resistant hash function H and of an (n, k)-erasure code $\mathcal{C}$ with encoding function encode and decoding function decode, respectively, where k satisfies $k \le n - t$. Our protocol also uses an asynchronous reliable broadcast protocol that tolerates Byzantine faults to disseminate a value among the servers, such as Bracha's protocol [7] (see Appendix B); its operations are denoted by *r-broadcast* and *r-deliver*, respectively, and *r-broadcast* may be executed by clients.

In our protocol, each value is written using a *timestamp*, which is an integer $ts \in \mathbb{N}$ acting as a version number for this value. Since it is possible that two writers use the same timestamp, we break ties by considering also the operation identifier, which is unique, and define the TIMESTAMP for a value being written with *oid* as $[ts, oid]$. TIMESTAMPS are ordered lexicographically, that is, given $[ts, oid]$ and $[ts', oid']$, we define

$$[ts, oid] <_{TS} [ts', oid'] \Leftrightarrow (ts < ts') \vee (ts = ts' \wedge oid <_{oid} oid'), \tag{1}$$

where operation identifiers are ordered according to some canonical order $<_{oid}$. Furthermore, one can define the relation $\le_{TS}$ in the usual way: for every two TIMESTAMPS TS, TS', we set $TS \le_{TS} TS' \Leftrightarrow (TS = TS') \vee (TS <_{TS} TS')$.

We now outline the key elements of Protocol Atomic. We start by describing the data structure maintained by every honest server, and then provide a brief explanation of the write and read operations.

Data stored by servers. A value F stored in the system is encoded with the (n, k)-erasure code $\mathcal{C}$ into a vector $[F_1, \dots, F_n]$. Every honest server P_j maintains a global variable F_c containing F_j for every tag *ID*. Note that at any point in time, distinct honest servers might store blocks of different values, as the system is asynchronous. Additionally, P_j stores $\mathbf{D}_c$, a vector consisting of the hashes of $F_1, \dots, F_n$, and a TIMESTAMP $[ts_c, oid_c]$ for the stored value. It also maintains a set $\mathcal{L}$, called the set of *listeners* [23], which contains at any point in time a set of tuples $[oid', TS', i']$, denoting the operation identifiers, TIMESTAMPS, and client identifiers of the concurrently executing read operations that it is aware of. Given F_{init}, let $[\tilde{F}_1, \dots, \tilde{F}_n]$ be equal to $\mathsf{encode}(F_{init})$. Initially, the variable F_c of server P_j is set to $\tilde{F}_j$, $\mathbf{D}_c$ is set to $[H(\tilde{F}_1), \dots, H(\tilde{F}_n)]$, and $[ts_c, oid_c]$ is set to $[0, \perp]$.

Write operations. A client C_i writing a value F to register *ID* with operation identifier *oid* first queries all servers for their most recent timestamps, and each server responds with ts_c. Once the client has received $n-t$ timestamps, it *r-broadcasts* the largest one to all servers and *disperses* F with Protocol $\mathsf{Disperse}$. When an honest server P_j has *r-delivered* a timestamp *ts* and *completed* the dispersal with vector $\mathbf{D}$, client identifier i, and block F_j, it increments the timestamp *ts*. Moreover, if $[ts_c, oid_c] <_{TS} [ts, oid]$, it replaces its stored values $[\mathbf{D}_c, F_c, ts_c, oid_c]$ by $[\mathbf{D}, F_j, ts, oid]$. In any case, the server checks for entries in $\mathcal{L}$ with TIMESTAMP smaller than $[ts, oid]$, and sends a `value` message to the corresponding clients with the new $\mathbf{D}$, F_j, and $[ts, oid]$. Finally, the server returns an acknowledgment message to the client. The client waits for $n - t$ such messages and terminates.

Read operations. A client C_i reading a value from register *ID* with operation identifier *oid* communicates its intention to read to the servers in a `read` message. Upon receipt of such a message, server P_j sends to C_i the vector $\mathbf{D}_c$, the block F_c, and the TIMESTAMP $[ts_c, oid_c]$ in a `value` message, unless server P_j has received a `read` message for register *ID* with identifier *oid* at an earlier time. P_j also adds the vector $[oid, [ts_c, oid_c], i]$ to $\mathcal{L}$.

C_i collects `value` messages from servers and stores them in a set $\mathcal{B}$. Such a `value` message could also have been caused by a concurrent write operation. Once C_i has received $n - t$ `value` messages

Protocol Atomic **for tag** *ID*

upon initialization: // Server P_j

$[\tilde{F}_1, \ldots, \tilde{F}_n] := \mathsf{encode}(F_{init})$

$\mathbf{D}_c := [H(\tilde{F}_1), \ldots, H(\tilde{F}_n)]$, $F_c := \tilde{F}_j$, $ts_c := 0$, $oid_c := \perp$, $\mathcal{L} := \emptyset$

upon *receiving* a message $(ID, \texttt{in}, \texttt{write}, oid, F)$: // Client C_i

for all $j \in [1, n]$ **do**

send $(ID, \texttt{get-ts}, oid)$ to P_j

wait for $n - t$ messages $(ID, \texttt{ts}, oid, ts_j)$ from distinct servers P_j

$ts := \max\{ts_j :$ a message $(ID, \texttt{ts}, oid, ts_j)$ has been received$\}$

disperse F using $\mathsf{Disperse}$ with tag $ID|disp.oid$ and *r-broadcast ts* with tag $ID|rbc.oid$

wait for $n - t$ messages $(ID, \texttt{ack}, oid)$ from distinct servers

output $(ID, \texttt{out}, \texttt{ack}, oid, F)$

upon *receiving* a message $(ID, \texttt{get-ts}, oid)$ from C_i: // Server P_j

send $(ID, \texttt{ts}, oid, ts_c)$ to C_i

upon *completing* $ID|disp.oid$ with $[\mathbf{D}, i, F_j]$ and *r-delivering ts* with tag $ID|rbc.oid$: // Server P_j

$ts := ts + 1$

if $[ts_c, oid_c] <_{TS} [ts, oid]$ **then**

$\mathbf{D}_c := \mathbf{D}$, $F_c := F_j$, $[ts_c, oid_c] := [ts, oid]$

for all $[oid', TS', i'] \in \mathcal{L}$ such that $TS' <_{TS} [ts, oid]$ **do**

send $(ID, \texttt{value}, oid', \mathbf{D}, F_j, [ts, oid])$ to $C_{i'}$

send $(ID, \texttt{ack}, oid)$ to C_i

output $(ID, \texttt{out}, \texttt{write-accepted}, oid)$

Figure 1: Protocol Atomic - initialization and write operation

Protocol Atomic **for tag** *ID*

upon *receiving* a message $(ID, \texttt{in}, \texttt{read}, oid)$: // Client C_i

$\mathcal{B} := \emptyset$

for all $j \in [1, n]$ **do**

send $(ID, \texttt{read}, oid)$ to P_j

repeat

wait for a message $(ID, \texttt{value}, oid, \mathbf{D}', F'_j, TS')$ from P_j such that $H(F'_j) = D'_j$

$\mathcal{B} := \mathcal{B} \cup \{[j, \mathbf{D}', F'_j, TS']\}$

until there exists a TIMESTAMP *TS*, a vector $\mathbf{D}$ and a set $\mathcal{S} \subseteq [1, n]$

such that $(|\mathcal{S}| = n - t) \wedge (\forall j \in \mathcal{S} : \exists F_j : [j, \mathbf{D}, F_j, TS] \in \mathcal{B})$

for all $j \in [1, n]$ **do**

send $(ID, \texttt{read-complete}, oid)$ to P_j

$F := \mathsf{decode}(\{(j, F_j) : j \in \mathcal{S}\})$

output $(ID, \texttt{out}, \texttt{read}, oid, F)$

upon *receiving* a message $(ID, \texttt{read}, oid)$ from C_i: // Server P_j

if $\mathcal{L}$ does not contain any entries $[oid, TS', i']$ for some TS' and i' **then**

$\mathcal{L} := \mathcal{L} \cup \{[oid, [ts_c, oid_c], i]\}$

send $(ID, \texttt{value}, oid, \mathbf{D}_c, F_c, [ts_c, oid_c])$ to C_i

upon *receiving* a message $(ID, \texttt{read-complete}, oid)$ from C_i: // Server P_j

remove from $\mathcal{L}$ all entries of the form $[oid, TS', i']$ for some TS', i'

Figure 2: Protocol Atomic - read operation

from distinct servers with the same TIMESTAMP and hash vector, then it stops collecting messages and decodes the received blocks to a value F. Before F is output, the client communicates the termination of its read to all servers in a `read-complete` message, in order to let them remove the corresponding entry from $\mathcal{L}$. Once a `read-complete` message has been received, the servers also stop responding to any `read` message with the same operation identifier.

The memory needed by the clients for storing $\mathcal{B}$ is not of interest in our model. In practice, however, one would use the elegant scheme of Martin et al. [23] that bounds the memory of the clients.

In the next section, we prove the following theorem.

Theorem 2. *Under the assumption that H is a collision-resistant hash function and $\mathcal{C}$ an (n,k)-erasure code, Protocol* Atomic *is an atomic register simulation protocol for $n > 3t$ and all $1 \leq k \leq n-t$.*

3.3 Analysis

The proof of Theorem 2 consists of two parts, corresponding to the liveness and correctness properties of an atomic register simulation protocol.

Let us first extend our terminology in order to handle TIMESTAMPS. We say that an honest server P_j *accepts a write to register ID with operation identifier oid and* TIMESTAMP $TS = [ts, oid]$ (or simply *with timestamp ts*) whenever P_j accepts the write to register *ID* with identifier *oid* after having *r-delivered* a timestamp $(ts-1)$. A read operation with operation identifier *oid* at an honest client C_i is said to *return a* TIMESTAMP *TS and a value F for tag ID* if the read returns F and the TIMESTAMPS of the corresponding blocks, from which F is decoded, are all *TS*. Furthermore, we say an honest client C_i *uses a* TIMESTAMP $TS = [ts, oid]$ in a write to register *ID* with operation identifier *oid* if it *r-broadcasts* a timestamp $(ts-1)$.

According to the agreement property of asynchronous reliable broadcast (see Appendix B), it is clear that if an honest server accepts a write to register *ID* with operation identifier *oid* and TIMESTAMP *TS*, and a distinct honest server accepts a write to register *ID* with operation identifier *oid* and TIMESTAMP TS', then $TS = TS'$. Therefore, we say that a write to register *ID* with operation identifier *oid takes effect with* TIMESTAMP *TS* (or *with timestamp ts*) if at least one honest server accepts it with TIMESTAMP *TS* (or with timestamp *ts*, respectively). Observe that no two write operations can take effect with the same TIMESTAMP since the operation identifier is part of the TIMESTAMP.

Liveness. It is easy to see that whenever a *write* operation is invoked at an honest client, then this operation also terminates, except with negligible probability: Since the client waits for $n-t$ `ts` messages and all honest servers eventually answer with a `ts` message, the client eventually receives enough timestamps, and moreover, according to the termination property of asynchronous verifiable information dispersal and the validity property of asynchronous reliable broadcast, all honest servers eventually send an acknowledgment message, except with negligible probability.

For *read* operations, note that since an honest client sends a `read` message to every server, every honest server eventually replies with a `value` message, unless the read operation has already terminated. Let $TS_{\max}$ be the largest TIMESTAMP contained in any `value` message sent by honest servers as a reply to the `read` messages for a particular operation identifier. By the properties of asynchronous reliable broadcast, all honest servers which have not sent a `value` message containing $TS_{\max}$ yet, eventually send a `value` message with $TS_{\max}$. The client receives these messages unless it has already terminated the read operation. Moreover, by the properties of Protocol Disperse, the hash vectors sent in the `value` messages with TIMESTAMP $TS_{\max}$ are the same, except with negligible probability. Thus, the read operation eventually terminates.

Correctness. In order to show correctness, we first need some technical lemmas, which are consequences of the implicit Byzantine quorum system [20] in the protocol.

Lemma 3. *Assume either a write operation to register ID has terminated at an honest client C_i and a* TIMESTAMP *TS has been used, or a read operation from register ID has terminated at an honest client C_i returning a* TIMESTAMP *TS. Then, if at a later time a read operation from register ID is invoked at an honest client C_ℓ, it does not return a* TIMESTAMP *smaller than TS.*

Proof. If such a write operation has terminated at an honest client C_i, it means C_i has received $n-t$ `ack` messages. At least $n-2t$ of these messages have been sent from honest servers, and by the agreement property of reliable broadcast, these honest servers all delivered the same timestamp. These honest servers send in every `value` message in the subsequent read operation a TIMESTAMP which is not smaller than *TS*. Analogously, if such a read operation has terminated, the honest client has received $n-t$ `value` messages from distinct servers with TIMESTAMP *TS*, and at least $n-2t$ of them have been sent by honest servers. Hence, none of these honest servers will send a TIMESTAMP smaller than *TS* in the subsequent read operation at C_ℓ. Since $n-t$ `value` messages from distinct servers and with the same TIMESTAMP are needed for the read to terminate, and at most $2t < n-t$ servers can send a TIMESTAMP smaller than *TS*, the lemma follows. □

Lemma 4. *Assume either a write operation to register ID has terminated at an honest client C_i and a* TIMESTAMP *TS has been used, or a read operation from register ID has terminated at an honest client C_i returning a* TIMESTAMP *TS. Then, if a write operation to register ID is invoked by an honest client C_ℓ at a later time, the write uses a* TIMESTAMP $TS' >_{TS} TS$.

Proof. Assume $TS = [ts, oid]$. With the same argument as in Lemma 3, at least $n-2t \geq t+1$ honest servers will send `ts` messages to C' in the subsequent write operation with a timestamp $ts' \geq ts$. In particular, in every set of $n-t$ `ts` messages received by C' at least one `ts` message must contain a timestamp $ts' \geq ts$. But this means that C' broadcasts a timestamp $ts'' \geq ts' \geq ts$, and thus uses a TIMESTAMP $TS' >_{TS} TS$. □

The following two lemmas state two additional important properties of TIMESTAMPS: a TIMESTAMP is connected to a unique value, except with negligible probability, and, moreover, in order for a TIMESTAMP to be read, it must have been written.

Lemma 5. *Assume an honest client reads a value F with* TIMESTAMP *TS, and some distinct honest client reads a value F' with the same* TIMESTAMP *TS. Then $F = F'$, except with negligible probability.*

Proof. Assume $TS = [ts, oid]$ and that indeed $F \neq F'$. Note that every honest server sends for a certain TIMESTAMP only a possible hash vector $\mathbf{D}$ in its `value` messages (since there is a unique instance of Disperse with tag $ID|disp.oid$). Moreover, `value` messages from $n-t$ distinct servers with the same TIMESTAMP *TS* and the same hash vector $\mathbf{D}$ are needed for both clients to terminate the read. But since any two sets of at least $n-t$ servers have at least one honest server in their intersection, both clients must have used the same hash vector $\mathbf{D}$ in the read operation. However, as two different values have been read according to our assumption, and only messages containing correct values according to the hash vector are accepted, the adversary must have found a collision for H. □

Lemma 6. *Assume a read operation from register ID at an honest client returns a* TIMESTAMP *TS. Then some write has taken effect with* TIMESTAMP *TS at an earlier time.*

Proof. If no such write ever takes effect, only corrupted servers can send this TIMESTAMP. But since there are at most t of them, an honest client never returns such a TIMESTAMP. □

For every adversary A and every tag *ID*, we can now construct an order $<$ over the operation identifiers in $\mathcal{R}_A^{ID} \cup \mathcal{E}_A^{ID}$ which ensures the correctness property.

- We order the operation identifiers in $\mathcal{E}_A^{ID}$ according to the TIMESTAMPS with which the corresponding writes take effect. Given two distinct operation identifiers oid_1 and oid_2 such that the corresponding writes take effect with TIMESTAMPS TS_1 and TS_2, we define

$$oid_1 < oid_2 \Leftrightarrow TS_1 \leq_{TS} TS_2. \tag{2}$$

- We order the operation identifiers in $\mathcal{R}_A^{ID}$ according to the returned TIMESTAMPS and the *precedes* relation established by the scheduler. In case two reads are concurrent and read the same timestamp, we break ties by using the canonical total order $<_{oid}$. Given two distinct operation identifiers oid_1 and oid_2 of reads at honest clients that return TIMESTAMPS TS_1 and TS_2, respectively, we define

$$\begin{aligned} oid_1 < oid_2 \Leftrightarrow & (TS_1 <_{TS} TS_2) \\ & \vee (TS_1 = TS_2 \wedge oid_1 \text{ precedes } oid_2) \\ & \vee (TS_1 = TS_2 \wedge (oid_2 \text{ does not precede } oid_1) \wedge oid_1 <_{oid} oid_2). \end{aligned} \tag{3}$$

- For every operation identifier oid_w of a write operation which takes effect with TIMESTAMP TS_w, and every operation identifier oid_r of a read operation which returns a TIMESTAMP TS_r, we define

$$oid_w < oid_r \quad \Leftrightarrow \quad TS_w \leq_{TS} TS_r \tag{4}$$
$$oid_r < oid_w \quad \Leftrightarrow \quad TS_r <_{TS} TS_w. \tag{5}$$

Observe that $<$ is a total order, since it is easy to verify that every pair of distinct oid_1, oid_2 satisfies either $oid_1 < oid_2$ or $oid_2 < oid_1$ from (2), (3), (4) and (5). In the following, for every two operation identifiers oid_1, oid_2, we say that $oid_1 \leq oid_2$ if either $oid_1 = oid_2$ or $oid_1 < oid_2$.

We are now ready to prove correctness. First observe that condition (i) is directly satisfied because of Lemmas 3 and 4, and the definition of the total order $<$. In order to prove (ii), assume a read operation with identifier oid_r returns a TIMESTAMP TS_r, and let oid_w be the largest identifier (according to the total order $<$) of a write which takes effect and such that $oid_w < oid_r$. According to Lemma 6, there is an operation identifier $oid'_w \in \mathcal{E}_A^{ID}$ such that a write operation with identifier oid'_w and TIMESTAMP TS_r takes effect. Note that $oid'_w < oid_r$, because of (4), and as oid_w is maximal, $oid'_w \leq oid_w < oid_r$. Additionally, it follows from (2) and (4) that the write operation with identifier oid_w also takes effect with TIMESTAMP TS_r, and this yields $oid'_w = oid_w$. Moreover, because of (3) and (4), every read operation with identifier oid'_r for $oid_w < oid'_r < oid_r$ returns TIMESTAMP TS_r, and from Lemma 5 we infer that these reads all return the same value except with negligible probability. Furthermore, if oid_w is the identifier of a write operation at an honest client, since no other write operation can use the same TIMESTAMP, it must have written F, except with negligible probability, because of the properties of the Disperse protocol and because H is collision-resistant.

3.4 Non-skipping timestamps

Protocol Atomic above uses client-generated timestamps to keep track of the order of the values written to the register. However, such timestamps are problematic since corrupted clients and servers may increase the timestamp value arbitrarily. This does not affect the liveness or the correctness of the protocol, but it opens a denial-of-service attack because the timestamps can waste memory at the honest servers.

Suppose the servers use a predefined amount of storage for the timestamps, bounded by a fixed polynomial in the security parameter. Then the adversary can cause overflows and harm the correctness of a protocol by setting them directly to the largest available value. Timestamps that are bounded by the number of writes that have already been executed have been called *non-skipping* by Bazzi and Ding [5]. They additionally ensure that whenever a value is written with a particular timestamp, every smaller timestamp has already been used at a previous point in time to write another value. And a fixed, polynomial-sized non-skipping timestamp value can accommodate any polynomial number of write operations.

In this section, we modify Protocol Atomic by using threshold signatures to implement non-skipping timestamps. Note that timestamps are not part of Definition 1. They are strictly related only to an implementation of an atomic register simulation protocol because such a protocol might be based on

Protocol AtomicNS **for tag** *ID*

upon initialization: // Server P_j

$[\tilde{F}_1, \ldots, \tilde{F}_n] := \mathsf{encode}(F_{init})$

$F_c := F_j$, $\mathbf{D}_c := [H(\tilde{F}_1), \ldots, H(\tilde{F}_n)]$, $ts_c := 0$, $oid_c := \perp$, $sig_c := \perp$, $\mathcal{L} := \emptyset$

upon *receiving* a message $(ID, \mathtt{in}, \mathtt{write}, oid, F)$: // Client C_i

for all $j \in [1, n]$ **do**

send $(ID, \mathtt{get\text{-}ts}, oid)$ to P_j

wait for $n - t$ messages $(ID, \mathtt{ts}, oid, ts_j, \mu_j)$ from distinct servers P_j with valid σ_j

$ts := \max \{ts_j :$ a message $(ID, \mathtt{ts}, oid, ts_j, \sigma_j)$ has been received$\}$

let σ be a signature corresponding to ts

disperse F using Disperse with tag $ID|disp.oid$ and *r-broadcast* $[ts, \sigma]$ with tag $ID|rbc.oid$

wait for $n - t$ messages $(ID, \mathtt{ack}, oid)$ from distinct servers

output $(ID, \mathtt{out}, \mathtt{ack}, oid, F)$

upon *receiving* a message $(ID, \mathtt{get\text{-}ts}, oid)$ from C_i: // Server P_j

send $(ID, \mathtt{ts}, oid, ts_c, sig_c)$ to C_i

upon *completing* $ID|disp.oid$ with $[\mathbf{D}, i, F_j]$ and *r-delivering* $[ts, \sigma]$ with tag $ID|rbc.oid$: // Server P_j

if $\mathsf{verify}([ID, ts], \sigma, PK) = \mathtt{true}$ **then**

$ts := ts + 1$

$\mu_j := \mathsf{sign}([ID, ts], PK, SK_j)$

for all $s \in [1, n]$ **do**

send $(ID, \mathtt{share}, oid, \mu_j)$ to P_s

wait for $n - t$ messages $(ID, \mathtt{share}, oid, \mu_s)$ from distinct servers P_s

with $\mathsf{verify\text{-}share}([ID, ts], \mu_s, PK, VK_s) = \mathtt{true}$

let Σ be the set of received valid signature shares

$\sigma := \mathsf{combine}([ID, ts], \Sigma, PK, [VK_1, \ldots, VK_n])$

if $[ts_c, oid_c] <_{TS} [ts, oid]$ **then**

$\mathbf{D}_c := \mathbf{D}$, $F_c := F_j$, $[ts_c, oid_c] := [ts, oid]$, $sig_c := \sigma$

for all $[oid', TS', i'] \in \mathcal{L}$ such that $TS' <_{TS} [ts, oid]$ **do**

send $(ID, \mathtt{value}, oid', \mathbf{D}, F_j, [ts, oid])$ to $C_{i'}$

send $(ID, \mathtt{ack}, oid, ts)$ to C_m

output $(ID, \mathtt{out}, \mathtt{write\text{-}accepted}, oid)$

Figure 3: Protocol AtomicNS - initialization and write operation

other techniques (e.g., atomic broadcast from the clients to the servers to serialize the operations [17, 10, 8]). For this reason, we refrain from formally defining non-skipping timestamps for atomic register simulations and rather show that the timestamps of our modified protocol are bounded.

We now describe our Protocol AtomicNS, in which the value of the timestamp in every accepted write is bounded by the number of writes to the register. The idea is to enforce non-skipping increments of the timestamp value by authenticating every timestamp with a threshold signature on $[ID, ts]$. Honest servers only accept, and subsequently increment, a timestamp if the client supplies a valid threshold signature. In order to increment the timestamp, the servers generate a new threshold signature for the timestamp by exchanging a round of messages containing signature shares. This ensures that the honest servers determine the growth of the timestamp.

Suppose a non-interactive (n, t)-threshold signature scheme $\mathcal{TSS}$ as in Section 2.2 is available. In particular, a trusted dealer initializes the system by generating a public key PK, secret keys SK_j and public verification keys VK_j for each server P_j. Server P_j receives PK, $VK_1, \ldots, VK_n$, and SK_j. Additionally, every server stores a global variable sig_c which is a threshold signature on ID and the current timestamp ts_c. This variable is initialized to $\perp$, and without loss of generality we assume $\perp$ is a valid signature for 0. Otherwise, the setup is the same as for Protocol Atomic.

When a client C_i writes a value F to register ID with operation identifier oid, and queries all servers in order to receive the most recent timestamps, it also receives a corresponding threshold signature σ_j from every P_j. Then it determines the largest timestamp *ts* and the accompanying signature σ, and *r-broadcasts* $[\mathit{ts}, \sigma]$. When an honest server P_j has *r-delivered* such a timestamp/signature pair $[\mathit{ts}, \sigma]$, verified that σ is valid, and has *completed* the dispersal, it sets the timestamp to $\mathit{ts} + 1$ and generates a new signature share on $[\mathit{ID}, \mathit{ts} + 1]$. Using a `share` message, it sends the signature share to all servers and then waits for enough signature shares from other servers in order to obtain a threshold signature σ on $[\mathit{ID}, \mathit{ts} + 1]$. Then, the server proceeds as before, but treats σ as a part of the timestamp, in particular, it also updates sig_c to σ when ts_c is updated. The details of the write operation in Protocol AtomicNS are given in Figure 3; the read operation is the same as in Protocol Atomic (Figure 2).

Clearly, AtomicNS is an atomic register simulation protocol for all $k \leq n - t$ and $n > 3t$, under the assumption that H is collision-resistant and $\mathcal{TSS}$ satisfies robustness. Liveness is satisfied since every honest server signs the same timestamp value and generates a valid signature share; thus, the additional round in which `share` messages are exchanged completes and every honest server obtains a valid threshold signature on $\mathit{ts} + 1$.

Lemma 7. *Assuming $\mathcal{TSS}$ satisfies non-forgeability, for every adversary A, the following holds in every run except with negligible probability: If some honest server has accepted a write operation to register ID with operation identifier oid and timestamp ts, then for all timestamps ts′ satisfying $0 \leq \mathit{ts}' < \mathit{ts}$, there exists an oid′ such that a write to register ID has taken effect with identifier oid′ and timestamp ts′.*

Proof. Towards a contradiction, assume there is an adversary and a tag *ID* such that in some run a write operation has been accepted by some honest server with timestamp *ts*, but for some $\mathit{ts}' < \mathit{ts}$, no write operation to register *ID* has taken effect. Let ts'' be the maximum timestamp with $\mathit{ts}' \leq \mathit{ts}'' < \mathit{ts}$ such that no write operation to register *ID* has been accepted with timestamp ts''.

Since a write operation has been accepted with timestamp $\mathit{ts}'' + 1$, some honest server has *r-delivered* a pair $[\mathit{ts}'', \sigma'']$ with a valid σ'' on $[\mathit{ID}, \mathit{ts}'']$. But by our assumption, no honest server has accepted a write operation with timestamps ts''. According to the protocol, this implies that no honest server has sent a `share` message containing a signature share on $[\mathit{ID}, \mathit{ts}'']$ and no honest server has generated such a share. Hence, σ'' must have been generated by the adversary, creating a forgery for the threshold signature scheme $\mathcal{TSS}$. □

Lemma 7 combined with the agreement property of reliable broadcast implies that every write operation takes effect with a unique timestamp, and it follows that the maximal value of a timestamp of register *ID* is bounded by the number of writes to register *ID*.

3.5 Complexity analysis

In the following, we determine the storage complexity of Protocol AtomicNS and analyze the communication and message complexities of isolated read and write operations. Every message is associated with either a read or a write operation according to the description in Figures 2 and 3. Since the complexity of a write operation depends on the number of concurrent reads, a meaningful complexity analysis is only possible by bounding the concurrency in the system. We therefore assume that at every honest server, the size of the set of listeners $\mathcal{L}$ is bounded by some value L. (Note that this violates the liveness of our protocol.)

According to our system model, the adversary is polynomial-time bounded, and, therefore, the number of scheduled messages, the number of distinct operations, and the number of clients actively taking part in a run of a protocol are all bounded by a polynomial in the security parameter. Hence, we may assume w.l.o.g. that the tags and operation identifiers are small and bounded by $\mathcal{O}(\log \kappa)$. Because the timestamps are non-skipping, the same holds for the size of the timestamps. Denote the maximal size of a threshold signature or a threshold signature share by $|S|$ and denote by $|H|$ the size of a hash value.

Complexity of write. The *message complexity* of a write operation is dominated by the complexity of the underlying dispersal and reliable broadcast protocols, which generate $\mathcal{O}(n^2)$ messages each. $\mathcal{O}(n^2)$ messages are also needed for the additional round of `share` messages. An honest server may also send up to L `value` messages to reading clients. Thus, the message complexity is $\mathcal{O}(n^2 + nL)$.

The communication complexity is dominated by the dispersal of the value and by the reliable broadcast of the timestamp and the corresponding signature. Again, there are up to L `value` messages of size $\mathcal{O}(\frac{|F|}{k} + n|H|)$ each which can be sent by every server. Hence, the communication complexity of a write operation is

$$\mathcal{O}\big(n(n+L)\frac{|F|}{k} + n^2(n+L)|H| + n^2|S|\big). \tag{6}$$

Using the maximal $k = n - t$, the communication complexity is $\mathcal{O}\big((n+L)|F| + n^2(n+L)|H| + n^2|S|\big)$. Note that using hash trees instead of hash vectors in the dispersal protocol according to [9], the $n^2(n+L)|H|$ term can be reduced to $n \log n(n+L)|H|$.

Complexity of read. Only $\mathcal{O}(n)$ *messages* are ever sent for a read operation. The *communication complexity* is dominated by the `value` messages. Again, for $k = n - t$, the communication complexity of a read operation is

$$\mathcal{O}(|F| + n^2|H|). \tag{7}$$

As already noted in the previous paragraph, this can be reduced to $\mathcal{O}(|F| + n \log n|H|)$ by using hash trees instead of hash vectors.

Storage complexity. For a particular *ID*, every server stores a vector $\mathbf{D}_c$, a block F_c, a threshold signature, and the set of listeners $\mathcal{L}$. This amounts to $n\frac{|F|}{k} + n^2|H| + n|S| + \mathcal{O}(nL\log\kappa)$ bits. Hence, in the optimal case $k = n - t$ the *storage complexity* is

$$\mathcal{O}\big(|F| + n^2|H| + n|S| + nL\log\kappa\big). \tag{8}$$

Under the reasonable assumption that $|F| \gg \max\{n|S|, n^2|H|, nL\log\kappa\}$, the storage blow-up is $\frac{n}{n-t} + o(1)$, which is nearly optimal. The use of hash trees can further reduce the $n^2|H|$ term to $n \log n|H|$.

Note that in practice, storage systems often execute write operations without any concurrent reads. In such an *optimistic case*, every honest server has $\mathcal{L} = \emptyset$ and no `value` messages are forwarded during the write operations. Moreover, each read operation returns the value written with the largest TIMESTAMP by a previous write operation.

References

[1] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. P. Stern, "Scalable secure storage when half the system is faulty," in *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP)* (U. Montanari, J. D. P. Rolim, and E. Welzl, eds.), vol. 1853 of *Lecture Notes in Computer Science*, pp. 576–587, Springer, 2000.

[2] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. P. Stern, "Addendum to scalable secure storage when half the system is faulty," *Information and Computation*, 2004. To appear.

[3] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM*, vol. 42, no. 1, pp. 124–142, 1995.

[4] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi, "Towards an object store," in *Proc. IEEE/NASA Conference on Mass Storage Systems & Technologies (MSST 2003)*, 2003.

[5] R. Bazzi and Y. Ding, "Non-skipping timestamps for Byzantine data storage systems," in *Proc. 18th International Conference on Distributed Computing (DISC 2004)* (R. Guerraoui, ed.), vol. 3274 of *Lecture Notes in Computer Science*, pp. 405–419, 2004.

[6] R. E. Blahut, *Theory and Practice of Error Control Codes*. Reading: Addison-Wesley, 1983.

[7] G. Bracha, "An asynchronous [(n - 1)/3]-resilient consensus protocol," in *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 154–162, 1984.

[8] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001. Full version available from *Cryptology ePrint Archive*, Report 2001/006, `http://eprint.iacr.org/`.

[9] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," Tech. Rep. RZ 3569, IBM Research, Dec. 2004.

[10] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, pp. 398–461, Nov. 2002.

[11] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, "Verifiable secret sharing and achieving simultaneity in the presence of faults," in *Proc. 26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 383–395, 1985.

[12] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *Proc. 28th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 427–437, 1987.

[13] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin, "Secure distributed storage and retrieval," *Theoretical Computer Science*, vol. 243, no. 1–2, pp. 363–389, 2000.

[14] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, pp. 37–45, Nov. 2000.

[15] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient Byzantine-tolerant erasure-coded storage," in *Proc. International Conference on Dependable Systems and Networks (DSN-2004)*, pp. 135–144, 2004.

[16] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "The safety and liveness properties of a protocol family for versatile survivable storage infrastructures," Tech. Rep. CMU-PDL-03-105, Parallel Data Laboratory, Carnegie Mellon University, Mar. 2004.

[17] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems* (S. J. Mullender, ed.), New York: ACM Press & Addison-Wesley, 1993. Expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[18] H. Krawczyk, "Distributed fingerprints and secure information dispersal," in *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 207–218, 1993.

[19] L. Lamport, "On interprocess communication. Part ii: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[20] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[21] D. Malkhi and M. K. Reiter, "An architecture for survivable coordination in large distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 187–202, 2000.

[22] D. Malkhi, M. Reiter, and N. Lynch, "A correctness condition for memory shared by byzantine processes." Manuscript, 1998.

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

[24] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology: CRYPTO '91* (J. Feigenbaum, ed.), vol. 576 of *Lecture Notes in Computer Science*, pp. 129–140, Springer, 1992.

[25] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM*, vol. 36, no. 2, pp. 335–348, 1989.

[26] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

A Review of asynchronous verifiable information dispersal

Information dispersal allows a client to store a value, usually called a *file*, in a distributed storage system. *Asynchronous verifiable information dispersal* has been introduced by Cachin and Tessaro [9] in the model of Section 2.1 and extends previous approaches by introducing the notion of *verifiability*: servers are always able to detect at the time of writing whether the information being stored is inconsistent.

An *asynchronous verifiable information dispersal (AVID) scheme* for a file F consists of two protocols:

The dispersal protocol: A client starts this protocol as it decides to store a certain file F in the storage system provided by the n servers. Some redundancy is added to the file, which is then split into n different blocks, each one being stored by one of the n servers.

The retrieval protocol: A client (not necessarily the same which has written the file F), wanting to retrieve file F, invokes this protocol in order to receive enough information from the servers to reconstruct the file F. Moreover, the retrieval protocol can be repeated as many times as necessary.

In the general definition of asynchronous verifiable information dispersal, we do not address the questions of concurrency and versioning. A file F can be written only once, but retrieved again and again. Since updates are not possible, concurrent reads and writes are not a problem. Stored files are indexed using the tag *ID* of the instance of the dispersal protocol which wrote them. Therefore, running the retrieval protocol for *ID* simply means retrieving the file stored with the instance of the dispersal protocol with tag *ID*.

We say that a client *disperses* a file F for *ID* if it starts the dispersal protocol with tag *ID* with a file F as an input, that is, it is activated through an input action $(ID, \texttt{in}, \texttt{disperse}, F)$. Furthermore, a server may *complete* the dispersal *ID* if it terminates the dispersal protocol for *ID* with some output of type `stored`, and it may *abort* the dispersal *ID* if it terminates the protocol with an output of type `abort`. However, a server might neither complete nor abort the dispersal. Finally, a client *reconstructs* a file F' for *ID*$'$ if it terminates the retrieval protocol for the file stored with tag *ID*$'$ with an output $(ID, \texttt{out}, \texttt{retrieved}, F')$.

The verifiability property requires that either all servers complete the dispersal or no server completes the dispersal. This ensures that the servers always store consistent data once enough honest servers have accepted. This is formalized in the following definition.

Definition 8. A (n, k)-*asynchronous verifiable information dispersal scheme* ($k \leq n$) is composed by a dispersal and a retrieval protocol which satisfy, for any t-limited adversary, any *ID*, and any client C_i starting the dispersal protocol for *ID*, the following conditions, except with negligible probability:

Termination: If the client C_i is honest, then all honest servers eventually complete the dispersal *ID*.

Agreement: If some honest server completes the dispersal *ID*, then all honest servers eventually complete the dispersal *ID*.

Availability: If k honest servers complete the dispersal *ID*, and an honest client C_j starts the retrieval protocol for *ID*, then it eventually reconstructs some file F'.

Correctness: If k honest servers complete the dispersal *ID*, there exists a fixed value G such that the following holds:

1. If C_i is honest and has dispersed a file F using *ID*, then $G = F$.
2. If an honest client C_j reconstructs F' for *ID*, then $G = F'$.

B Review of asynchronous reliable broadcast

Given the model introduced in Section 2.1, a protocol for *asynchronous reliable broadcast* is a protocol where a party (called a *dealer*) *r-broadcasts* a message m and all servers may *r-deliver* a value m'. Such a protocol satisfies the following properties:

Validity: If an honest dealer r-broadcasts a message m, some honest server eventually r-delivers m.

Agreement: If some honest server r-delivers a message m', then all honest servers eventually r-deliver m'.

Authenticity: Every honest server r-delivers at most one message m. Moreover, if the dealer is honest, m was previously r-broadcast by the dealer.

Note that in contrast to the usual definition of reliable broadcast, where the dealer belongs to the set of servers, the dealer is allowed to be a client in our context. This modification does not actually cause any problems, and existing protocols for reliable broadcast can be easily adapted in order to satisfy this new requirement.

The standard protocol for asynchronous reliable broadcast has been presented by Bracha [7]. When broadcasting a message m, this protocol has message complexity $\mathcal{O}(n^2)$ and communication complexity $\mathcal{O}(n^2|m|)$. Note that the message complexity is actually optimal, and we cannot expect to achieve anything better.

Bracha's protocol has been improved by Cachin et al. [8] using a hash function H, in order to reduce the communication complexity in an *optimistic setting*. That is, if messages among honest parties arrive in time and if the servers controlled by the adversary are not actively interfering with the execution of the protocol, the communication complexity is bounded by $\mathcal{O}\left(n|m| + n^2|H|\right)$, where $|H|$ is the size of the hash function output. On the other hand, in the worst case, that is, if the corrupted servers cheat actively and the network is slow, the communication complexity can be as high as $\mathcal{O}\left(n^2(|m| + |H|)\right)$, and no improvement with respect to Bracha's protocol is achieved.

Cachin and Tessaro [9] propose an asynchronous reliable broadcast protocol with communication complexity $\mathcal{O}(n|m| + n^2 \log n|H|)$.