Separating Data and Control: Asynchronous BFT Storage with $2t + 1$ Data Replicas

Christian Cachin[1], Dan Dobre[2], Marko Vukolić[3]

[1] IBM Research - Zurich, Switzerland, cca@zurich.ibm.com
[2] Work done at NEC Labs Europe, Germany, dan@dobre.net
[3] Eurécom, France, vukolic@eurecom.fr

Abstract. The overhead of Byzantine fault tolerant (BFT) storage is a primary concern that prevents its adoption in practice. The cost stems from the need to maintain at least $3t + 1$ copies of the data at different storage replicas in the asynchronous model, so that t Byzantine replica faults can be tolerated. This paper presents *MDStore*, the first fully asynchronous BFT storage protocol that reduces the number of replicas that store the payload data to as few as $2t + 1$ and maintains metadata at $3t + 1$ replicas on (possibly) different servers. At the heart of *MDStore* lies a metadata service built upon a new abstraction called "timestamped storage." Timestamped storage allows for conditional writes (facilitating the implementation of the metadata service) and has consensus number one (making it implementable with wait-free semantics in an asynchronous system despite faults). In addition to its low replication overhead, *MDStore* offers strong guarantees by emulating a multi-writer multi-reader atomic register, providing wait-free termination, and tolerating any number of Byzantine readers and crash-faulty writers.

1 Introduction

Byzantine fault-tolerant (BFT) protocols are notoriously costly to deploy. Their overhead stems from the extra resources that must be installed compared to systems that tolerate less severe faults, such as crashes. For example, in the asynchronous communication model, BFT storage protocols that emulate a simple register abstraction need at least $N > 3t$ server replicas so that t faults can be tolerated [32]. This stands in contrast to the required number of replicas when only server crashes are tolerated, where $2t + 1$ replicas suffice. Such crash-tolerant systems based on quorums [34] are in production use today, in cloud-storage systems and other contexts. But the additional cost of handling Byzantine faults compared to crashes represents one of the main concerns for the adoption of BFT systems in practice.

In this paper, we show that the gap between crash-tolerance and Byzantine-tolerance in distributed storage can be reduced significantly. By separating the functions that handle metadata from those that store the payload data, the number of expensive servers with large storage capacity can be reduced to $N > 2t$ while tolerating Byzantine faults. We introduce protocol *MDStore*, which emulates a storage register abstraction in an asynchronous message-passing model; it requires only $N > 2t$ *storage replicas* that store payload data (of which t may be Byzantine) and $M > 3f$ *metadata replicas*

that maintain short control information (of which f may be Byzantine). Storage and metadata replicas may be separated physically or co-hosted on the same servers.

Despite achieving lower replication cost, *MDStore* does not sacrifice other desirable features: *MDStore* implements a multi-writer multi-reader (MWMR) *atomic* register [21, 24] with *wait-free* semantics [20], tolerates any number of Byzantine readers and crash-faulty writers, and works *without any synchrony assumption*. Compared to other BFT storage protocols that reduce the number of storage replicas to $3t$ or less [11, 12, 22, 33], *MDStore* is the first one that achieves this without trusted hardware components. Moreover, because *MDStore* is fully asynchronous and does not employ a consensus primitive, it fundamentally differs from other related systems that separate the control plane from the data plane for providing, e.g., consensus [19], state-machine replication [26, 37], and distributed storage [2] — these are all subject to the FLP impossibility result [17] and require partial synchrony [15].

Protocol *MDStore* has a modular architecture. The clients exchange metadata about the stored data through a *metadata service (MDS)*. The metadata related to a stored value v consists of a cryptographic hash of v, a logical timestamp, and pointers to $t + 1$ among the N storage replicas that store v. Our MDS implementation contains an array of simple read/write registers with safe semantics for the hash values and a novel *timestamped storage* function for the other metadata. Timestamped storage offers conditional operations to multiple readers and writers, is linearizable, and has wait-free semantics. The storage replicas, on the other hand, simply store data values associated to timestamps.

The timestamped storage function is very similar to a classical atomic register [24], except that it also exposes a timestamp associated with the stored value. This permits the clients to execute conditional writes, i.e., write operations that take effect depending on the timestamp value. Interestingly, despite its support of conditional writes, timestamped storage has consensus number equal to one [20], and this paves the way for a wait-free BFT distributed implementation of the MDS in the asynchronous model. We show how to realize the MDS for *MDStore* from asynchronous BFT safe [1, 18, 31] and atomic [3, 9, 13, 30] single-writer storage protocols using $M > 3f$ metadata replicas.

In a preliminary version of this work [6], we also show why the number N of storage replicas cannot be reduced to $2t$ or less, even when only crashes are tolerated. Furthermore, we argue that cryptographic techniques, in particular, collision-free hash functions, appear to be necessary for any BFT storage emulation that uses $3t$ or fewer replicas.

The rest of the paper is organized as follows. The next section further discusses the relation of *MDStore* to other work; Section 3 introduces the system model and definitions. In Section 4, protocol *MDStore* is presented with an overview, pseudocode, an example execution, and a formal correctness argument.

2 Related work

The formal study of *registers* as abstractions for concurrently accessed read/write storage starts with Lamport's classical paper [24]; this work also introduced safe, regular, and atomic consistency properties. Martin et al. [32] establish a tight lower bound of

$3t + 1$ replicas for any register implementation that tolerates t Byzantine replicas in an asynchronous system. Their bound applies even to a single-writer single-reader safe register, where the reader and the writer may only fail by crashing. In this paper, we refine our understanding of this bound by logically separating the replicas into storage replicas and metadata replicas. Protocol *MDStore* shows that the lower bound of $3t + 1$ replicas [32] applies only to metadata replicas that exercise a control function. The number of storage replicas, which take care of storing the data, can be lowered to $2t + 1$ in the presence of t Byzantine faults, assuming cryptographic techniques.

Some elements of *MDStore* are similar to mechanisms in Farsite [2], a virtual file service that tolerates some Byzantine nodes, and Hybris [14], a recent hybrid cloud storage system. In particular, Farsite and Hybris separate metadata from data, they store cryptographic hashes and maintain directory information in a metadata service, and they both use only $2t+1$ storage replicas that are subject to Byzantine faults. However, unlike *MDStore*, the metadata services of Farsite and Hybris are based on a generic service implemented by a replicated state machine. Hence, Farsite and Hybris are subject to the FLP impossibility result [17] and require at least partial synchrony [15], whereas *MDStore* is asynchronous. The replication mechanism in Farsite assumes there is a single writer and uses read/write locks for concurrency control. On the other hand, Hybris is not wait-free as it only provides reads that are live in the presence of finitely many concurrent writes (so-called *FW-termination* [1]). Protocol *MDStore*, in contrast, supports multiple concurrent writers, offers atomic semantics, and provides wait-free termination without resorting to locks.

Many practical storage systems separate data and control for reasons related to performance and modularity [36]. In an asynchronous model where nodes are subject to crashes, several replicated storage systems have divided the control path for metadata from the data path for bulk data [10, 16, 35]. Interestingly, on a conceptual level, this separation does not pay off with crash-faulty replicas, as it does not allow to lower the number of storage replicas to below $2t+1$. These related systems all require $2t+1$ storage replicas. It can be shown that this is inherent: $2t + 1$ storage replicas are necessary, even with a fault-free metadata service [6].

In the context of state-machine replication and the consensus problem, separating data from control functions is a well-known technique. Lamport's Paxos consensus algorithm [25, 26] introduces three roles for the participant processes and distinguishes between proposers, acceptors, and learners. The lower bound of $3t + 1$ replicas for partially synchronous BFT consensus only applies to the acceptors but not to proposers or learners [27]. For example, there is a partially synchronous BFT consensus protocol in which any number of proposers and learners may be Byzantine [19]. Yin et al. [37] separate the agreement function from an execution component in a BFT system for generic state-machine replication, with $3t + 1$ replicas needed for agreement and $2t + 1$ replicas for storing state and executing commands. However, just like Farsite [2] and Hybris [14], these designs are fundamentally different from the principle underlying *MDStore*. As these are based on consensus, they are subject to the impossibility of consensus in asynchronous systems [17]; therefore, they rely on stronger timing assumptions [15].

3 System model and definitions

System model. We consider an asynchronous distributed system of *process abstractions* that communicate with each other. There are at least four kinds of processes: (1) a set $\mathcal{M} = \{m_1, \ldots, m_M\}$ of M *metadata replicas* that act as servers for (small) metadata, (2) a set $\mathcal{S} = \{s_1, \ldots, s_N\}$ of N *storage replicas* that store (large) values, (3) a set $\mathcal{W}$ of *writers* and (4) a set $\mathcal{R}$ of *readers*. The readers and writers together form the set $\mathcal{C}$ of *clients*, which run operations on the storage service. The set $\mathcal{R} = \mathcal{M} \cup \mathcal{S}$ denotes all *replicas*, which provide the storage service. Clients are disjoint from replicas. Processes may be *correct*, *benign*, or *Byzantine*, as defined later.

The processes interact asynchronously by exchanging events. A protocol specifies a collection of algorithms with instructions for all processes; equivalently, a distributed algorithm can be seen as a collection of deterministic automata, where each process is assigned an automaton. An *execution* of an algorithm is an infinite sequence of the *steps* taken by the correct and benign processes according to their algorithms, together with the actions of the Byzantine processes. More formal descriptions appear in the literature [7, 29].

A process may fail by *crashing* or by exhibiting *Byzantine* faults. A *benign* process executes its algorithm until it crashes and takes no further steps. A *Byzantine* process may perform arbitrary actions, such as sending arbitrary messages or changing its state in an arbitrary manner (NR-arbitrary faults). We assume an *adversary* that coordinates the Byzantine processes and controls the scheduling of events.

All writers are benign (they are correct or may crash), readers may be Byzantine, up to f metadata replicas are Byzantine, where $M > 3f$, and up to t storage replicas are Byzantine, where $N > 2t$. Processes that do not fail are called *correct*.

Channels. We assume that every process can communicate with every other process over point-to-point perfect asynchronous communication channels with FIFO order [7]. Perfect channels guarantee reliable communication among correct processes, i.e., that every message sent from a correct process is eventually delivered to a correct receiver exactly once. In an actual implementation, the channels between clients and replicas are *authenticated* in the sense that the adversary cannot modify or insert messages on the channels. Using point-to-point channels and a message-authentication code (MAC) [23], such authenticated channels can be implemented easily.

Notation. Protocols are presented in a modular way using an event-based notation [7]. A process exposes an *interface* to other processes, which defines the events that it exposes. Processes are specified either through abstract *properties* or via an *implementation*. A process may react to a received event by doing computation and triggering further events. Every process is named by an identifier. Events are qualified by the process identifier to which the event belongs and may take parameters. An event *Sample* of a process *m* with a parameter x is denoted by $\langle$ *m*-*Sample* $\mid x \rangle$.

Objects and histories. An *object* is a special type of process for which every input event (called an *invocation* in this context) triggers exactly one output event (called a

response). Every such pair of invocation and response define an *operation* of the object. An operation *completes* when its response occurs.

A *history* σ of an execution of an object O consists of the sequence of invocations and responses of O occurring in σ. An operation is called *complete* in a history if it has a matching response. 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. If o precedes o' then o' *follows* o. 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.

An execution is *well-formed* if the events at every object are 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 triggered events to be consumed (see the standard literature for a formal definition [28]).

Registers. A *read/write register r* is an object that stores a value from a domain $\mathcal{V}$ and supports exactly two operations, for writing and reading the value. More precisely:

- A *Write* operation to *r* is triggered by an invocation $\langle$ *r-Write* $\mid v \rangle$ that takes a value $v \in \mathcal{V}$ as parameter and terminates by generating a response $\langle$ *r-WriteAck* $\rangle$ with no parameter.
- A *Read* operation from *r* is triggered by an invocation $\langle$ *r-Read* $\rangle$ with no parameter; the register signals that the read operation completes by triggering a response $\langle$ *r-ReadVal* $\mid v \rangle$, which contains a parameter $v \in \mathcal{V}$.

The behavior of a register is given through its sequential specification, which requires that every *r-Read* operation returns the value written by the last preceding *r-Write* operation in the execution, or the special symbol $\perp \notin \mathcal{V}$ if no such operation exists. For simplicity, we will assume that every distinct value is written only once.

In this work, there are multiple readers and writers for the emulated storage, but only readers may invoke *Read* operations and only writers may invoke *Write* operations on the emulated register. Such a register is also called a *multi-writer multi-reader (MWMR) register* (we will also use a *single-writer* variant, abbreviated *SWMR*). Furthermore, we assume that all clients invoke a *well-formed* sequence of operations.

Consistency and availability. Recall that clients interact with an object O through its operations, defined in terms of an invocation and a response event of O. We say that a client c *executes* an operation between the corresponding invocation and response events. When accessed concurrently by multiple processes, executions of objects considered in this work are *linearizable*, that is, the object appears to execute all operations *atomically*.

More formally, a sequence of events π is called a *view* of a history σ at a client c w.r.t. an object O whenever:

1. π is a sequential permutation of some subsequence of complete operations in σ;
2. all complete operations executed by c appear in π; and
3. π satisfies the sequential specification of O.

Definition 1 (Linearizability [21]). *A history σ is linearizable w.r.t. an object O if there exists a sequence of events π such that:*

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

The goal of this work is to describe a protocol that emulates a linearizable register abstraction among the clients; such a register is also called *atomic*. Some of the clients may crash and some replicas may be Byzantine, but every client operation should terminate in all cases, irrespective of how other clients and replica behave.

Definition 2 (Wait-freedom [20]). *A protocol is called* wait-free *if every operation invoked by a correct client eventually completes.*

Cryptography. We make use of cryptographic hash functions. One can imagine that these are implemented by a distributed oracle accessible to all processes [7]. A hash function H maps an input value x of arbitrary length (e.g., represented as a bit string) to a short, unique representation in a small domain (e.g., a bit string of fixed length). We use a *collision-free* hash function; this property means that no process, not even a Byzantine process, can find two distinct values x and x' such that $H(x) = H(x')$.

4 Protocol *MDStore*

MDStore emulates a MWMR atomic wait-free register. Our implementation of *MDStore* is modular. We begin this section by specifying an abstract metadata service (MDS). Then we given an overview of *MDStore*, which uses the MDS abstraction and $N > 2t$ storage replicas, describe its implementation, and illustrate it through a sample execution. Subsequently we discuss possible implementations of the MDS in a distributed system from $M > 3f$ metadata replicas. Finally, we argue why *MDStore* provides a wait-free atomic register.

4.1 Timestamped storage and the metadata service

The metadata service used by *MDStore* is assumed to be a wait-free abstraction provided by a correct process. The MDS comprises two independent functions: the first is a storage abstraction called *timestamped storage*, which resembles a register object with a versioned interface and a particular sequential specification; the second one models an *array of registers* for storing hash values associated to timestamps.

The specification of the MDS appears in Alg. 1. The timestamped storage function is accessed through the *MDS-WriteTs* and *MDS-ReadMax* operations and maintains a timestamp *ts* and a value *data*. In order to write a timestamped value, a client supplies a write-timestamp *wts* and a data value v. The MDS stores (wts, v) in its state $(ts, data)$ if and only if $wts \geq ts$. In a read operation for the timestamped value, the MDS returns the stored *ts* and *data*.

In the specification of timestamped storage it is critical that the guard for a *MDS-WriteTs* operation to "take effect" requires *wts* to be *greater than or equal to* the stored *ts*. With this condition, timestamped storage has consensus number *one* [20] and can be implemented from simple atomic registers, as discussed later in Section 4.4.

In contrast, Cachin et al. [8] define a "replica" object that is the same as the timestamped storage function, except that the guard for the conditional write requires the write-timestamp to be *strictly greater* than the stored timestamp; this object, however, is much more powerful and more difficult to implement, as it has an infinite consensus number [8].

The second function of the MDS stores an array of independent hash values associated with timestamps. The operations *MDS-WriteHash* and *MDS-ReadHash* implement these in the canonical way.

Algorithm 1. Timestamped-storage metadata service *MDS*.

```
1: Types
2:     TS = ℕ0 × (C ∪ {⊥}), with fields num and c        // ts = (ts.num, ts.c) for ts ∈ TS

3: State
4:     ts ∈ TS, initially (0, ⊥)                          // Timestamp of stored value
5:     data ∈ Σ*, initially ⊥                             // Stored metadata associated with ts
6:     hashes[ts] ∈ Σ*, initially ⊥, for ts ∈ TS          // Hash values associated to timestamps

7: upon ⟨ MDS-WriteTs | wts, v ⟩ do
8:     if wts ≥ ts then
9:         (ts, data) ← (wts, v)
10:    invoke ⟨ MDS-WriteTsAck ⟩

11: upon ⟨ MDS-ReadMax ⟩ do
12:    invoke ⟨ MDS-ReadMaxVal | ts, data ⟩

13: upon ⟨ MDS-WriteHash | ts, h ⟩ do
14:    hashes[ts] ← h
15:    invoke ⟨ MDS-WriteHashAck | ts ⟩

16: upon ⟨ MDS-ReadHash | ts ⟩ do
17:    invoke ⟨ MDS-ReadHashVal | ts, hashes[ts] ⟩
```

4.2 Description

Protocol *MDStore* operates similar to related algorithms and associates an increasing timestamp, chosen by the writer, to every written value. It employs the MDS for storing metadata of two kinds according to the previous section. First, the timestamped storage function of the MDS maintains the *authoritative* timestamp *ts*, i.e., the one of the most recently written value; it also acts as a *directory* by pointing to a set of $t + 1$ storage replicas that store the value associated with *ts*. This resembles the role of metadata in Farsite [2] and LDR [16]. The second function of the MDS permits to store hash values associated with timestamps, and writers in *MDStore* store the hash of a written value

there, indexed by the timestamp. The hash ensures the integrity of the value towards readers, as a majority of the storage replicas may be Byzantine. Every client may write to and read from the MDS, but the hash values for a particular timestamp is written only once by a single client.

A timestamp *ts* in *MDStore* (see also Alg. 1) is a classical multi-writer timestamp [5, 7], consisting of a pair (num, c), where *num* is an integer and c is a client identifier (of the writer). The latter serves to break ties. Comparison of timestamps uses lexicographic ordering such that $ts_1 > ts_2$ if and only if $ts_1.num > ts_2.num$ or $ts_1.num = ts_2.num$ and $ts_1.c > ts_2.c$.

The pseudocode for clients is given in Alg. 2 and the pseudocode for storage replicas appears in Alg. 3. At a high level, a *r-Write* operation that writes value v to register r proceeds as follows (Alg. 2): (1) the writer c_w invokes *MDS-ReadMax* and obtains the latest timestamp *ts* from the MDS (line 22); (2) it produces a write-timestamp *wts* by incrementing *ts* and writes the hash of v to the MDS under *wts* (lines 23–25); (3) c_w now invokes s_i-*Write* on all storage replicas s_i for $i \in [1, N]$ with *wts* and v, and waits for a set Q of $t + 1$ replicas to acknowledge the write (lines 26–30); (4) c_w writes (wts, Q) to MDS with the timestamped storage function (line 31); (5) c_w now invokes s_i-*Commit* on all storage replicas with parameter *wts*, such that they may garbage collect the stored values associated to timestamps smaller than *ts* (lines 32–33); and, finally, (6) the writer resets its internal state (lines 34–35). In response to a s_i-*Write* operation, a storage replica saves the written value indexed by the write-timestamp, as long as the write-timestamp exceeds the most recently committed timestamp at s_i. This means that a storage replica may store multiple values at one time.

On the other hand, when a reader c_r invokes *r-Read*, it first obtains the authoritative metadata $(ts, replicas)$ from the MDS, where *replicas* denotes the $t + 1$ storage replicas which have stored the value and acknowledged it to the writer (Alg. 2, line 38). The reader then invokes s_i-*Read* with parameter $rts = ts$ on s_i for $i \in replicas$ (lines 42–44). The storage replica s_i responds with the value indexed by the timestamp *rts* supplied by c_r; however, if s_i has already committed a higher timestamp than *rts* and thus deleted the corresponding value, then it advances the timestamp to the committed timestamp and responds with that value (lines 61–64, Alg. 3). Hence, the reader c_r obtains a value associated to timestamp *rts* or to a higher one.

Since clients cannot trust replicas, the reader *validates* the value received through s_i-*ReadVal* from the replica. To this end, c_r consults the MDS and verifies that the hash of the value v with timestamp *ts* received from the replica matches the hash stored at the MDS as follows (lines 45–51): (1) c_r retrieves the hash value h' corresponding to *ts* from the MDS; (2) c_r will check that $H(v) = h'$ (line 51); (3) if *ts* (which was obtained from s_i) is higher than *rts* (which the reader requested) due to a concurrent write operation, then c_r validates *ts* by retrieving the authoritative metadata with the currently highest timestamp $\overline{ts}$ from the MDS and by checking that *ts* lies between *rts* and $\overline{ts}$ (lines 47–51).

As a side remark to Alg. 2, the values *data* and *data*$'$ obtained in lines 22 and 48, respectively, are ignored.

Intuitively, the register emulation preserves safety because the MDS stores an authoritative hash of the value stored by the (Byzantine) storage replicas. Furthermore,

Algorithm 2. Protocol *MDStore*, atomic register instance r for client c.

State
  $wts, rts \in TS$, initially $(0, \perp)$ // Timestamp of written and read value, resp.
  $Q \in 2^{\mathbb{N}}$, initially $\emptyset$ // Storage replicas that have acknowledged write

upon $\langle$ *r-Write* | v $\rangle$ **do**
  invoke $\langle$ *MDS-ReadMax* $\rangle$; **wait for** $\langle$ *MDS-ReadMaxVal* | *ts*, *data* $\rangle$
  $wts \leftarrow (ts.num + 1, c)$
  invoke $\langle$ *MDS-WriteHash* | *wts*, $H(v)$ $\rangle$
  wait for $\langle$ *MDS-WriteHashAck* | ts' $\rangle$ **such that** $ts' = wts$
  forall $i \in [1, N]$ **do**
    invoke $\langle$ s_i*-Write* | *wts*, v $\rangle$

upon $\langle$ s_i*-WriteAck* | *ts* $\rangle$ **such that** $ts = wts$ **do**
  $Q \leftarrow Q \cup \{i\}$
  if $|Q| > t$ **then**
    invoke $\langle$ *MDS-WriteTs* | *wts*, Q $\rangle$; **wait for** $\langle$ *MDS-WriteTsAck* $\rangle$
    forall $i \in [1, N]$ **do**
      invoke $\langle$ s_i*-Commit* | *wts* $\rangle$
    $wts \leftarrow (0, \perp)$
    $Q \leftarrow \emptyset$
    invoke $\langle$ *r-WriteAck* $\rangle$

upon $\langle$ *r-Read* $\rangle$ **do**
  invoke $\langle$ *MDS-ReadMax* $\rangle$; **wait for** $\langle$ *MDS-ReadMaxVal* | *ts*, *replicas* $\rangle$
  if $ts = (0, \perp)$ **then**
    invoke $\langle$ *r-ReadVal* | $\perp$ $\rangle$
  $rts \leftarrow ts$
  forall $i \in$ *replicas* **do**
    invoke $\langle$ s_i*-Read* | *rts* $\rangle$

upon $\langle$ s_i*-ReadVal* | *ts*, v $\rangle$ **do**
  invoke $\langle$ *MDS-ReadHash* | *ts* $\rangle$
  wait for $\langle$ *MDS-ReadHashVal* | ts', h' $\rangle$ **such that** $ts' = ts$
  if $ts > rts$ **then**
    invoke $\langle$ *MDS-ReadMax* $\rangle$; **wait for** $\langle$ *MDS-ReadMaxVal* | $\overline{ts}, data'$ $\rangle$
  else
    $\overline{ts} \leftarrow rts$
  if $rts \leq ts \leq \overline{ts} \wedge H(v) = h'$ **then**
    $rts \leftarrow (0, \perp)$
    invoke $\langle$ *r-ReadVal* | v $\rangle$

Algorithm 3. Protocol *MDStore*, implementation of storage replica s_i.

```
54: State
55:     ts ∈ TS, initially (0, ⊥)                          // Committed timestamp
56:     values[ts] ∈ V, initially ⊥, for ts ∈ TS           // Map of stored values

57: upon ⟨ s_i-Write | wts, v ⟩ do
58:     if wts > ts then
59:         values[wts] ← v
60:     invoke ⟨ s_i-WriteAck | wts ⟩

61: upon ⟨ s_i-Read | rts ⟩ do
62:     if rts < ts then
63:         rts ← ts
64:     invoke ⟨ s_i-ReadVal | rts, values[rts] ⟩

65: upon ⟨ s_i-Commit | cts ⟩ do
66:     if cts > ts ∧ values[cts] ≠ ⊥ then
67:         ts ← cts
68:         forall freets ∈ TS such that freets < ts do
69:             values[freets] ← ⊥
70:     invoke ⟨ s_i-CommitAck | cts ⟩
```

client operations are linearizable because of the atomic operations on the MDS primitive. For showing liveness and that the emulation is wait-free, note that the writer never blocks, assuming a wait-free MDS abstraction. Moreover, the timestamp *ts* obtained by the reader together with v is higher and therefore "more recent" than the timestamp *rts*, which the reader initially requested, due to the protocol logic at the storage replicas. The range check $rts \leq ts \leq \overline{ts}$ by the reader ensures that *ts* is also permitted with respect to the authoritative timestamp $\overline{ts}$. The formal analysis appears in Section 4.5.

4.3 Illustration

We illustrate *MDStore* using an execution σ, depicted in Figure 1. In σ, we assume $t = 1$ and $N = 3$ storage replicas. Replica s_1 does not receive any message due to asynchrony in a timely manner, whereas replica s_3 is Byzantine.

The execution starts with a complete operation $o_{w,1} = r\text{-}Write(v_1)$ that writes (ts_1, v_1) to the storage replicas s_2 and s_3; the timestamp ts_1 is a pair $(1, w_1)$ that the writer w_1 generated in line 23 during $o_{w,1}$. The operation $o_{w,1}$ is not contained in the figure, only the state of the MDS upon completion of $o_{w,1}$ is shown.

The initial write $o_{w,1}$ is followed by two concurrent operations shown in Figure 1: first, $o_{w,2} = r\text{-}Write(v_2)$ by a writer w_2, and, second, $o_r = r\text{-}Read$ by a reader r_1. Upon invoking $o_{w,2}$, writer w_2 in Step ① (referring to the numbers in Fig. 1) first invokes *MDS-ReadMax* on the MDS (line 22). When the MDS responds, the writer w_2 obtains the highest timestamp $ts_1 = (1, w_1)$. Then, w_2 computes the timestamp of its operation

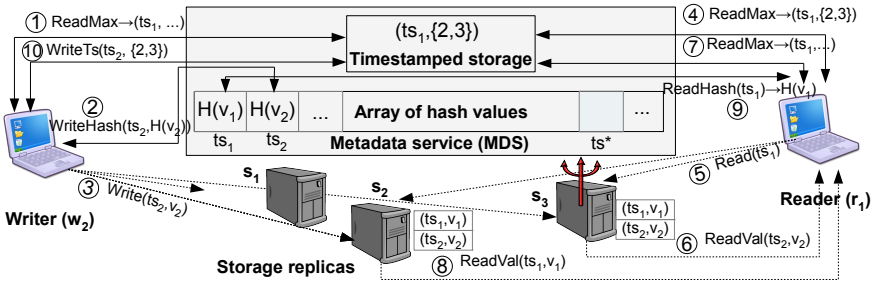

Fig. 1. An execution of *MDStore* with a concurrent *r-Write* and *r-Read* operation.

as $ts_2 = (2, w_2)$ (line 23) and invokes *MDS-WriteHash* with ts_2 and $H(v_2)$ in Step ② (line 24). Notice that the hash is written to the MDS *before* the write $o_{w,2}$ is exposed to other clients via the timestamp through the MDS; this will prevent a Byzantine storage replica from forging values with a given timestamp. Eventually, the MDS responds and w_2 then invokes s_i-*Write*(ts_2, v_2) on the storage replicas for $i = 1, \ldots, 3$ in Step ③ (lines 26–27). The messages carrying these operations are received only by the storage replicas s_2 and s_3 (but recall that s_3 is Byzantine). Since s_2 is correct, it stores v_2 in $values[ts_2]$ (line 59). At this point in the execution, the writer w_2 stalls, waiting for two s_i-*WriteAck* replies from the storage replicas.

Concurrently with $o_{w,2}$, a reader r_1 invokes $o_r =$ *r-Read*. The reader first queries the MDS through a *MDS-ReadMax* operation in Step ④ to determine the latest timestamp *rts* and the set *replicas*, which store the corresponding value (line 38). The MDS responds such that $rts = ts_1$ and $replicas = \{2, 3\}$. Next, in Step ⑤, r_1 invokes s_i-*Read*(ts_1) on the storage replicas s_2 and s_3 (lines 42–44). According to the algorithm, a storage replica responds to this with the value that it stores under ts_1 or under its committed timestamp *cts*, and not necessarily with the value from *data* with the highest timestamp at the replica; for instance, at this time in σ, for replica s_2, it holds $cts = ts_1$ since no s_2-*Commit*(ts_2) has been invoked yet. However, the Byzantine replica s_3 could mount a sophisticated attack and include (ts_2, v_2) in its s_3-*ReadVal* response, see Step ⑥. Although value v_2 is in fact being written concurrently, it would be wrong for r_1 to return v_2, since readers do not write back data in *MDStore* and the write of v_2 is not yet complete — this may violate atomicity. For preventing this attack, the reader subsequently invokes *MDS-ReadMax* again to determine whether ts_2 (or a higher timestamp) has become authoritative meanwhile, in Step ⑦ (lines 47–48). Since this is not the case here, client r_1 discards the response from s_3 (after the test in line 51) and waits for an additional reply (this will arrive from s_2).

An alternative attack by the Byzantine replica s_3 could be to make up a value v^* with a large timestamp, say $ts^* = (100, w_2)$. In this case, r_1 would also check with the MDS whether ts^* or a higher timestamp has been written (just like in Step ⑦). Moreover, r_1 would check the integrity of the value reported by s_3 by retrieving the hash at ts^* from the MDS and by checking if it matches the hash of v^* (lines 45–51). As the hash function is collision-free and the MDS is correct, this check will fail.

Returning to σ, in Step ⑧, s_2 eventually responds to r_1 with the pair (ts_1, v_1) (lines 61–64). According to the protocol, r_1 successfully verifies the integrity of v_1 after obtaining the hash value at ts_1 from the MDS in Step ⑨ (lines 45–51), and the *r-Read* of r_1 returns v_1.

Eventually, the writer w_2 in $o_{w,2}$ receives two s_i-*WriteAck* responses from replicas s_2 and s_3. Then, it invokes *MDS-WriteTs* with ts_2 and the set $\{2, 3\}$ in Step ⑩ (line 31). Note that the write of v_2 only "takes effect" at this point in time; in other words, the linearization point of $o_{w,2}$ coincides with the linearization point of the *MDS-WriteTs* operation with ts_2, and it is safe subsequently for readers to read v_2 from *r*.

Finally, the writer invokes s_i-*Commit* on all storage replicas, so as to allow them to garbage collect stale data (lines 32–33). Storage replicas update their local variable *ts*, which determines the value that they will send to a reader, only upon processing this s_i-*Commit* operation (lines 65–70).

Let us point out that *MDStore* uses timestamped storage at the MDS as a way to avoid storing an entire history of values at the storage replicas. One could not achieve this saving if the MDS would only expose a standard read/write register interface, since this would allow that a stored value is overwritten by a value with a lower timestamp. Given the implementation of storage replicas (notably lines 57–60) and our goal of avoiding to store entire histories, such an overwrite might cause inconsistent states between the MDS and the storage replicas.

4.4 Implementation of the metadata service

We show how to implement the MDS abstraction with existing asynchronous BFT storage protocols that rely on $M > 3f$ metadata replicas. In order to qualify for the implementation, such a BFT protocol should also tolerate an arbitrary number of Byzantine readers, permit multiple benign writers (which may crash), and, ideally, make no cryptographic assumptions. Recall that the MDS has two completely independent functions, providing the timestamped storage and the array of hash values. Hence, we will implement them through different components.

First, the wait-free atomic timestamped storage function can be implemented as a straightforward extension of the classical SWMR to MWMR transformation on atomic storage objects (e.g., [7, page 163]). In this transformation, there is one SWMR storage object per writer and every writer maintains a timestamp/value pair in "its" storage object, after first reading and incrementing the highest timestamp found in any other storage object. In our extension, the reader determines the timestamp/value pair with the highest timestamp among the SWMR storage objects as usual, and simply returns also the timestamp together with the value. This implementation may be realized from existing SWMR atomic wait-free storage (using $M > 3f$ replicas); some permit a computationally unbounded adversary [3,13], whereas others assume cryptography, that is, they tolerate only a computationally bounded adversary [9, 30].

Second, the function related to the hash values consists simply of an array of SWMR safe storage objects. These may be directly implemented from the protocols with atomic semantics mentioned above. Furthermore, as one may relax the consistency guarantee for them to safe semantics, one might also employ protocols with weaker semantics,

such as (1) SWMR safe wait-free storage [1] or (2) its regular variant, both without cryptographic assumptions [18], or (3) regular storage with digital signatures [31].

Finally, note that more efficient, direct, implementations of the *MDStore* metadata service can be obtained easily, but these are beyond the scope of this paper.

4.5 Analysis

In this section we prove that protocol *MDStore* in Alg. 2–3 emulates an atomic MWMR register and is wait-free.

We define the *timestamp of an operation* o on the register as follows: If o is *r-Write*, then its timestamp is the value of variable *wts* after the assignment in line 23; otherwise, if o is *r-Read*, its timestamp is the value of variable *ts* obtained through s_i*-ReadVal* (line 44) at the time when o returns by invoking *r-ReadVal*.

Lemma 1 (Monotonicity of timestamped storage). *Consider the timestamped storage function of the MDS and suppose an operation o_r = MDS-ReadMax returns (ts', v'). If o_r follows an operation o_w = MDS-WriteTs(ts, v) or an operation o'_r = MDS-ReadMax that returns (ts, v) then $ts' \geq ts$.*

Proof. This follows directly from the sequential specification of timestamped storage in Alg. 1. □

Lemma 2 (Sandwich). *Let o_r be a complete r-Read operation with timestamp ts, let rts denote the timestamp returned by the MDS in line 38 and let rts′ denote the timestamp returned by the MDS in line 48. Then $rts \leq ts \leq rts'$.*

Proof. According to the definition of the operation timestamp, the timestamp of o_r is the value of the variable *ts* at line 53. Consider the test that $rts \leq ts \leq \overline{ts}$ in line 51. According to the algorithm, if $ts > rts$, then the variable $\overline{ts}$ contains rts'. □

Lemma 3 (Partial Order). *Let o and o' be two operations with timestamps ts and ts′, respectively, such that o precedes o'. Then $ts \leq ts'$ and if o' is a r-Write operation, then $ts < ts'$.*

Proof. Suppose o is a *r-Read* operation. Then its timestamp is either equal to *rts*, which is returned by *MDS-ReadMax* in line 38, or *ts* is not larger than $\overline{ts}$, which is returned by *MDS-ReadMax* in line 48. On the other hand, if o is a *r-Write* operation, its timestamp is written to the MDS through *MDS-WriteTs*. Hence, at the time when o completes, the monotonicity of the timestamped storage (Lemma 1) implies that any subsequent *MDS-ReadMax* operation returns a timestamp that is at least as large as *ts*.

In the following we consider operation o' that follows o and distinguish two cases:

1. Suppose o' is a *r-Read* operation. Then its timestamp ts' is at least as large as the timestamp *rts*, which is returned by *MDS-ReadMax* in line 38, and the lemma follows.
2. Otherwise, o' is a *r-Write* operation. Then its timestamp $ts' = wts$ is computed in line 23 from the timestamp returned by *MDS-ReadMax* by incrementing its first component. Hence *wts* and the timestamp of o' are strictly larger than the timestamp returned by *MDS-ReadMax* and, hence, also strictly larger than *ts*.

Lemma 4 (Unique writes). *If o and o' are two r-Write operations with timestamps ts and ts$'$, respectively, then ts $\neq$ ts$'$.*

Proof. If o and o' are executed by different clients, then the two timestamps differ in their second component. If o and o' are executed by the same client, then the client executed them sequentially. By Lemma 3, it follows *ts* $\neq$ *ts*$'$.

Lemma 5 (Integrity). *Let o_r be a r-Read with timestamp ts_r that returns a value $v \neq \perp$. Then there exists a unique r-Write operation o_w that writes v such that its timestamp ts_w is equal to ts_r. Furthermore o_w does not follow after o_r.*

Proof. Since o_r returns v and has timestamp ts_r, the reader receives a s_i-*ReadVal* response containing ts_r and v from one of the storage replicas. Suppose for the purpose of contradiction that v was never written. Then, then by the collision resistance of H, the check in line 51 fails and o_r does not return v. Therefore, we conclude that some *r-Write* operation o_w has invoked s_i-*Write*(ts_r, v) on a storage replica in line 27. Since this timestamp ts_r is equal to variable *wts* and the timestamp ts_w of o_w, it follows that $ts_w = ts_r$. Finally, by Lemma 4, no other *r-Write* operation has the same timestamp, which completes the proof.

Theorem 1 (Linearizability). *Every execution of protocol MDStore is linearizable.*

Proof. Let σ be the history of any execution of *MDStore*. By Lemma 5 the timestamp of a *r-Read* operation has either been written by some *r-Write* operation or the *r-Read* operation returns $\perp$.

We first construct σ' from σ by completing all operations of the form *r-Write*(v) such that v has been returned by some complete *r-Read*. Then we construct a sequential permutation π of σ' by ordering all operations in σ', excluding the *r-Read* operations that returned $\perp$, according to their timestamps and by placing all *r-Read* operations that did not return $\perp$ immediately after the *r-Write* operation with the same timestamp. The *r-Read* operations that returned $\perp$ are placed at the beginning of π. Note that (concurrent) *r-Read* operations with the same timestamp may appear in any order, whereas all other *r-Read* operations appear in the same order as in σ'.

To prove that π preserves the sequential specification of a MWMR register we must show that every *r-Read* returns the value written by the latest *r-Write* that precedes it in π, or the initial value $\perp$ if there is no preceding *r-Write* in π. Let o_r be a *r-Read* operation returning a value v. If $v = \perp$, then by construction o_r is ordered before any *r-Write* in π.

Otherwise, $v \neq \perp$, and by Lemma 5, there exists a *r-Write*(v) operation with the same timestamp ts_r. In this case, this write is placed in π before o_r by construction. According to Lemma 4, every other *r-Write* in π has a different timestamp and, therefore, appears in π either before *r-Write*(v) or after o_r.

It remains to show that π preserves real-time order of σ. Consider two complete operations *o* and *o*$'$ in σ' such that *o* precedes *o*$'$ with timestamps *ts* and *ts*$'$, respectively. Lemma 3 implies that *ts*$' \geq$ *ts*. If *ts*$' \geq$ *ts*, then *o*$'$ follows *o* in π by construction. Otherwise *ts*$' =$ *ts* and Lemma 3 implies that *o*$'$ is a *r-Read* operation. If *o* is a *r-Write* operation, then *o*$'$ appears after *o* since we placed every *r-Read* after the *r-Write* with

the same timestamp. Otherwise, if o is a *r-Read*, then it appears in π before o', as it does in σ'.

Theorem 2 (Wait-freedom). *Every execution of protocol MDStore is wait-free.*

Proof. Since the MDS abstraction used by Alg. 2 is wait-free, every operation invoked on the MDS eventually completes. It remains to show that no *r-Write* always fails the test in line 30 and that no *r-Read* operation permanently fails the check of line 51 and never returns a value.

For a *r-Write* operation o_w, the condition in line 30 is eventually satisfied because there is a time after which all correct storage replicas have responded with s_i-*WriteAck* and because there are more than t correct replicas, from the assumption $N > 2t$.

On the other hand, let o_r be a *r-Read* operation and suppose for the sake of contradiction that the condition in line 51 is never satisfied — therefore, o_r never returns. Let s_i be a correct storage replica with $i \in \mathit{replicas}$. Since the reader has previously invoked s_i-*Read* on s_i during o_r, it eventually receives a s_i-*ReadVal*(ts, v) in response.

If *ts* satisfies the clause $\mathit{rts} \leq \mathit{ts} \leq \overline{\mathit{ts}}$ in line 51, then the second clause of the condition, $H(v) = h'$, is also true because s_i is correct, and o_r would return. Therefore, we continue the argument assuming that $\mathit{ts} < \mathit{rts}$ or that $\mathit{ts} > \overline{\mathit{ts}}$. Recall that the reader requested timestamp *rts* in s_i-*Read*. If $\mathit{ts} < \mathit{rts}$, then s_i has replied with a smaller timestamp than *rts*, which is not possible according to the algorithm for a replica (lines 62–64). Otherwise, if $\mathit{ts} > \overline{\mathit{ts}}$, then by Lemma 2, it holds $\mathit{ts} > \mathit{rts}$, and therefore s_i has replied from its committed timestamp variable; to avoid confusion, we call this value ts^* and note that $\mathit{ts}^* = \mathit{ts}$. According to the replica code, line 67 is the only place where its committed timestamp variable may change. Furthermore, if the replica sets this variable to ts^*, then there exists a *r-Write* operation o_w^* that committed with timestamp ts^*. According to the *r-Write* code, o_w^* commits only after invoking *MDS-WriteTs* containing timestamp ts^*. Hence, if $\mathit{ts} > \overline{\mathit{ts}}$, then o_r invokes *MDS-ReadMax* in line 48 and does so after the corresponding *r-Write* wrote ts^* to the MDS. According to Lemma 1, the reader obtains from the MDS in line 48 a timestamp $\overline{\mathit{ts}}$ that is a least as large as ts^*. This implies that $\overline{\mathit{ts}} \geq \mathit{ts}^* = \mathit{ts}$, which contradicts the assumption that $\mathit{ts} > \overline{\mathit{ts}}$, and the result follows.

5 Conclusion

This paper has explored how to separate the maintenance of metadata from the storage of bulk-data in distributed storage. It introduces *MDStore*, the first fully asynchronous wait-free BFT storage protocol that reduces the number of replicas that store bulk data to as few as $2t + 1$, with t Byzantine faults. Recent work shows that the same approach also improves erasure-coded protocols for distributed storage that tolerate Byzantine faults [4], reducing the storage overhead even further.

Acknowledgment

We thank Elli Androulaki, Alessandro Sorniotti, and Nikola Knežević for inspiring discussions about this work. This work is supported in part by the EU CLOUDSPACES (FP7-317555) and SECCRIT (FP7-312758) projects.

References

[1] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] A. Adya, W. J. Bolosky, M. Castro, et al. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symp. Operating Systems Design and Implementation (OSDI)*, 2002.

[3] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions. In A. Pelc, editor, *Proc. 21th International Conference on Distributed Computing (DISC)*, volume 4731 of *Lecture Notes in Computer Science*, pages 7–19. Springer, 2007.

[4] E. Androulaki, C. Cachin, D. Dobre, and M. Vukolić. Erasure-coded Byzantine storage with separate metadata. Report arXiv:1402.4958, CoRR, 2014.

[5] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, London, 1998.

[6] C. Cachin, D. Dobre, and M. Vukolić. BFT storage with $2t + 1$ data replicas. Report arXiv:1305.4868, CoRR, 2013.

[7] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[8] C. Cachin, B. Junker, and A. Sorniotti. On limitations of using cloud storage for data replication. Proc. 6th Workshop on Recent Advances in Intrusion Tolerance and reSilience (WRAITS 2012), 2012.

[9] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 115–124, 2006.

[10] B. Cho and M. K. Aguilera. Surviving congestion in geo-distributed storage systems. In *Proc. USENIX Annual Technical Conference*, pages 439–451, 2012.

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

[12] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proc. 23rd Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2004.

[13] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić. PoWerStore: Proofs of writing for efficient and robust storage. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2013.

[14] D. Dobre, P. Viotti, and M. Vukolić. Hybris: Consistency hardening in robust hybrid cloud storage. Research Report RR-13-291, Eurécom, 2013.

[15] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[16] R. Fan and N. A. Lynch. Efficient replication of large data objects. In F. E. Fich, editor, *Proc. 17th International Conference on Distributed Computing (DISC)*, volume 2848 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2003.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[18] R. Guerraoui and M. Vukolić. How fast can a very robust read be? In *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 248–257, 2006.

[19] R. Guerraoui and M. Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

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

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

[22] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proc. 7th European Conference on Computer Systems (EuroSys)*, pages 295–308, Apr. 2012.

[23] J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.

[24] L. Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 86–101, 1986.

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

[26] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

[27] L. Lamport. Lower bounds for asynchronous consensus. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*. Springer, 2003.

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

[29] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, Sept. 1989.

[30] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th Symposium on Reliable Distributed Systems (SRDS)*, 1998.

[31] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

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

[33] G. S. Veronese, M. Correia, A. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan. 2011.

[34] M. Vukolić. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

[35] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proc. USENIX Annual Technical Conference*, pages 413–424, 2012.

[36] J. Wilkes, C. Hoover, B. Keer, P. Mehra, and A. Veitch. *Storage, Data, and Information Systems*. HP Laboratories, 2008.

[37] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault-tolerant services. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–268, 2003.