Asynchronous BFT Storage with $2t + 1$ Data Replicas

Christian Cachin
IBM Research - Zurich
cca@zurich.ibm.com

Dan Dobre
NEC Labs Europe
dan.dobre@neclab.eu

Marko Vukolić
EURECOM
marko.vukolic@eurecom.fr

Abstract

The cost of Byzantine Fault Tolerant (BFT) storage is the main concern preventing its adoption in practice. This cost stems from the need to maintain at least $3t + 1$ replicas in different storage servers in the asynchronous model, so that t Byzantine replica faults can be tolerated.

In this paper, we present *MDStore*, the first fully asynchronous read/write BFT storage protocol that reduces the number of data replicas to as few as $2t + 1$, maintaining $3t + 1$ replicas of metadata at (possibly) different servers. At the heart of *MDStore* store is its metadata service that is built upon a new abstraction we call *timestamped storage*. Timestamped storage both allows for conditional writes (facilitating the implementation of a metadata service) and has consensus number one (making it implementable wait-free in an asynchronous system despite faults). In addition to its low data replication factor, *MDStore* offers very strong guarantees implementing multi-writer multi-reader atomic wait-free semantics and tolerating any number of Byzantine readers and crash-faulty writers.

We further show that *MDStore* data replication overhead is optimal; namely, we prove a lower bound of $2t + 1$ on the number of data replicas that applies even to crash-tolerant storage with a fault-free metadata service oracle. Finally, we prove that separating data from metadata for reducing the cost of BFT storage is not possible without cryptographic assumptions. However, our *MDStore* protocol uses only lightweight cryptographic hash functions.

1 Introduction

Byzantine Fault Tolerant (BFT) protocols are notoriously costly to deploy. This cost stems from the fact that, in many applications, tolerating Byzantine faults requires more resources than tolerating less severe faults, such as crashes. For example, in the asynchronous communication model, BFT read/write storage protocols [23] are shown to require at least $3t + 1$ replicas in different storage servers so that t Byzantine server faults can be tolerated [30]. This is to be contrasted with the requirement for $2t + 1$ replicas in the asynchronous crash model for protocols used in production cloud-storage systems. This gap between crash tolerance and BFT is one of the main concerns for practical adoption of BFT systems.

In this paper we show that this gap may in fact be significantly smaller. Namely, we present *MDStore*, a novel asynchronous message-passing read/write storage emulation that reduces the number of *data replicas* to only $2t + 1$, maintaining $3t_M + 1$ *metadata replicas* at (possibly) different servers. Here, t and t_M are thresholds on the number of Byzantine data and metadata replicas, respectively. To achieve lower replication cost, *MDStore* does not sacrifice other functionalities. Namely, *MDStore* implements multi-writer multi-reader (MWMR) atomic wait-free storage [18, 23] that tolerates any number of Byzantine readers and crash-faulty writers. *MDStore* is the first asynchronous BFT storage protocol that does not assume any trusted components to reduce its resource cost (unlike [9, 10, 21, 31]). Moreover, being a fully asynchronous read/write storage protocol, *MDStore* is fundamentally different from the existing consensus [17], state-machine replication (SMR) [25, 34] and SMR-based storage protocols [2], which employ the similar separation of control and data planes and which are all subject to the FLP impossibility result [15] and require partial synchrony [13].

MDStore has modular architecture: a client reads and writes metadata (which consists of a hash of the value, timestamp and pointers to data replicas that store a value) through the abstraction of a *metadata service*. Our implementation of metadata service consists of an array of SWMR safe wait-free storage objects [23] and a novel MWMR atomic wait-free storage object variant, which we call *timestamped storage*. In an array of safe storage, indexed by timestamps, *MDStore* stores hashes of data values, whereas in atomic timestamped storage, *MDStore* stores pointers to $t + 1$ (out of $2t + 1$) data replicas storing the most recent value. On the other hand, data replicas simply store timestamp/value pairs.

Our timestamped storage object is very similar to classical atomic [23] (or linearizable [20]) read/write storage, except that it also exposes a timestamp attached to the stored values to clients, allowing for conditional writes, i.e., writes that take effect conditional on a timestamp value. Interestingly, despite its support of conditional writes, timestamped storage has consensus number [18] equal to one, which makes an implementation of the metadata service possible in the asynchronous model despite faults. Indeed, we show that the *MDStore* metadata service can be implemented from simple asynchronous BFT SWMR safe [1, 16, 28] and SWMR atomic [3, 7, 11, 29] storage protocols using $3t + 1$ replicas for tolerating t faults; in the context of *MDStore*, these replicas are exactly the $3t_M + 1$ *metadata replicas*.

Complementing the *MDStore* protocol, this paper also establishes lower bounds on the number of data replicas that are needed for asynchronous storage implementations with logically separated metadata. In more detail:

- We prove that at least $2t + 1$ data replicas are necessary for implementations that leverage a metadata service, even if data replicas can fail only by crashing. This shows not only that *MDStore* is optimally resilient, but also that it incurs no additional data replication cost compared to crash-tolerant storage. The lower bound of $2t + 1$ has a very broad scope: it applies already to obstruction-free [19] single-writer single-reader safe storage [23] (and can be extended to eventual consistency [32]). Moreover, for the purpose of the lower bound, we define a metadata service very loosely as a fault-free oracle that provides arbitrary functionality with the single limitation that it cannot store or forward data values, roughly speaking. We believe that this definition of a metadata service is of independent interest.

- We show that reducing the cost of BFT storage by separating metadata and data requires to limit the computational power of a Byzantine adversary. In the practically relevant case of a bounded adversary that cannot subvert collision resistance of cryptographic hash functions, *MDStore* shows that $2t + 1$ data replicas are sufficient. However, with an unbounded adversary, we show that one needs $3t + 1$ data replicas, despite the metadata service oracle.

The rest of the paper is organized as follows. In Section 2 we introduce the system model and preliminary definitions. Section 3 presents *MDStore*. In Section 4 we prove our lower bounds on the number of data replicas and Section 5 discusses related work. Finally, Section 6 concludes the paper with an outlook to future work. The correctness proof of *MDStore* is postponed to Appendix A.

2 System model and definitions

Processes. The distributed system we consider consists of four sets of processes: (i) a set *metadata replicas* of size M containing processes $\{m_1, ..., m_M\}$, (ii) a set of D *data replicas* containing processes $\{d_1, ..., d_D\}$, (iii) a set of W *writers* containing processes $\{w_1, ..., w_W\}$; and (iv) a set *readers* of size R containing processes $\{r_1, ..., r_R\}$. The set *clients* is the union of *writers* and *readers*. Similarly, the set *replicas* denotes the union of *data replicas* and *metadata replicas*. Clients are disjoint from replicas, but *writers* and *readers* may intersect, just like *metadata replicas* and *data replicas*. Clients are either *benign* or *Byzantine*, as defined later.

We model distributed algorithm A for set of processes Π as a collection of deterministic automata, where A_p is the automata assigned to process $p \in \Pi$. The computation of benign processes proceeds in *steps* of A. For space constraints, we omit the details of this model and refer to the literature [27].

Channels. We assume that every process can communicate with every other process using point-to-point perfect asynchronous communication channels [5]. In short, perfect channels guarantee reliable communication: i.e., if neither process at the end of a communication channel is faulty, every sent message is eventually delivered to the receiver exactly once.[1] For presentation simplicity, we also assume a global clock, which, however, is not accessible to processes who perform local computations and communicate asynchronously.

Adversary. A *Byzantine* process p does not follow A_p and may perform arbitrary *actions*, such as (i) sending arbitrary messages or (ii) changing its state in an arbitrary manner. We assume an *adversary* that can coordinate Byzantine processes and make them collude.

We use a deterministic model for a cryptographic hash function. A hash function maps a bit string of arbitrary length to a short, unique representation of fixed length and consists of a distributed oracle accessible to all processes. The hash oracle exports a single operation H; its invocation takes a bit string x as parameter and returns an integer as the response. The oracle maintains a list L of all x that have been invoked so far. When the invocation contains $x \in L$, then H responds with the position of x in L; otherwise, H appends x to the end of L and returns its position. This ideal implementation models only collision resistance, i.e., that it is infeasible even for an unbounded adversary to produce two different inputs x and x' such that $H(x) = H(x')$.

In the following, unless explicitly specified differently, we use this model of a hash function. In our context this is equivalent to assuming that the adversary is computationally *bounded*, i.e., that it cannot

[1] Perfect channels are simply implemented from lossy channels using retransmission mechanisms [5].

break cryptographic hash functions. Alternatively, we speak of an *unbounded* adversary when no such hash function is available. This terminology matches the traditional names and formalizations of cryptographic hash functions [22].

Finally we assume that channels that relate benign processes are *authenticated*, i.e., that the adversary cannot (undetectably) insert messages in these channels. In practice, authenticated communication can be implemented easily from point-to-point channels with a message-authentication code (MAC) [22].

Executions and faults. Given any algorithm A, an *execution* of A is an infinite sequence of steps of A taken by benign processes, and actions of Byzantine processes. A *partial execution* is a finite prefix of some execution. A (partial) execution ex' *extends* some (partial) execution ex if ex is a prefix of ex'. We say that a benign process p is *correct* in an execution ex if p takes an infinite number of steps of A in ex. Otherwise a benign process p is *crash-faulty*. We say that a *crash-faulty* process p *crashes* at step sp in an execution, if sp is the last step of p in that execution.

All writers in our model are benign and any number of them can be crash-faulty. Moreover, any number of readers can be Byzantine. Unless stated differently, we assume that up to t (resp., t_M) data (resp., metadata) replicas can be Byzantine; all other replicas are correct. Unless stated differently, we assume $D = 2t + 1$ and $M = 3t_M + 1$.

Storage object. A storage abstraction is a shared READ/WRITE object. Its sequential specification consists of a shared variable x with two operations: WRITE(v), which takes a value v from domain V, stores v in x and returns special value $ok \notin V$, and READ(), which returns the value of x. We assume that the initial value of x is a special value $\perp \notin V$.

We assume that each client invokes at most one operation at a time (i.e., does not invoke the next operation until it receives the response for the current operation). Only writers invoke WRITE operations, whereas any client can invoke READ operations. When we talk about SWSR storage (single-writer single-reader), we assume that the writer and the reader are distinct process. Otherwise, we assume MWMR storage with $W \geq 1$ and $R \geq 1$.

For presentation simplicity, we do not explicitly model the initial state of processes nor the invocations and responses of the operations of the implemented storage object. We assume that the algorithm A initializes the processes in executions and determines the invocations and responses of operations. We say that p *invokes* an operation *op* at step sp when A modifies the state of a process p in step sp to start *op*; similarly, *op completes* at the step of A when the response of *op* is received.

We say that a READ/WRITE operation *op* is *complete* in a (partial) execution if the execution contains a response step for *op*. In any run, we say that a complete operation op_1 *precedes* operation op_2 (or op_2 *follows* op_1) if the response step of op_1 precedes the invocation step of op_2 in that run. If neither op_1 nor op_2 precedes the other, the operations are said to be *concurrent*.

Timestamped storage. We use a special storage variant called *timestamped storage* with a slightly different sequential specification. Besides x (initialized to $\perp$), timestamped storage maintains a timestamp TS (an integer, initially 0). Timestamped storage exports the following operations:

- TSWRITE((ts, v)) takes a pair of an integer timestamp ts and a value $v \in V$; $\underline{\text{if } ts \geq TS}$, then it stores ts to TS and v to x atomically[2]. Regardless of timestamp ts, TSWRITE returns ok.

- TSREAD() returns the pair (TS, x).

[2] Here, in the sequential specification of *timestamped storage*, it is critical to notice that the guard for a TSWRITE to "take effect" requires ts to be *greater or equal* to TS. With such a condition, *timestamped storage* has consensus number [18] *one*, and can be implemented with SWMR atomic registers as we discuss in Section 3.3. In contrast, [6] defines a "replica" object that is exactly the same as *timestamped storage* except that the guard for the conditional write requires ts to be *strictly greater* than TS; this object, however, has consensus number ∞.

Safety and liveness. An algorithm *implements safe* (or *atomic*) storage if every (partial) execution of the algorithm satisfies *safety* (or *atomicity*, respectively) properties [23]. We define safe storage for a single writer only and say that a partial execution satisfies *safety* if every READ operation rd that is not concurrent with any WRITE operation returns value v written by the last WRITE that precedes rd, or $\perp$ in case there is no such WRITE. An execution ex satisfies atomicity (or linearizability [20]) if ex can be extended (by appending zero or more response events) to an execution ex' and there is a sequential permutation π of ex' (without incomplete invocations) such that π preserves the real-time precedence order of operations in ex and satisfies the sequential specification. Moreover, a storage algorithm is *obstruction-free* or *wait-free* if every execution satisfies *obstruction-freedom* [19] or *wait-freedom* [18], respectively. Obstruction-freedom states that if a correct client invokes operation op and no other client takes steps, op eventually completes. Wait-freedom states that if a correct client invokes operation op, then op eventually completes. Atomicity and wait-freedom also apply to timestamped storage.

3 Protocol *MDStore*

In this section, we first give an overview of *MDStore* and then explain its modular pseudocode. We then discuss possible implementations of the *MDStore* metadata service module using existing BFT storage protocols. For lack of space, a full correctness proof is postponed to Appendix A.

3.1 Overview

MDStore emulates multi-writer multi-reader (MWMR) atomic wait-free BFT storage, using $2t+1$ data replicas and $3t_M+1$ metadata replicas. Our implementation of *MDStore* is modular. Namely, metadata replicas are hidden within a *metadata service* module MDS which consists of: (a) a MWMR atomic wait-free timestamped storage object (denoted by MDS_{dir}), which stores the metadata about the latest authoritative storage timestamp ts and acts as a *directory* by pointing to a set of $t+1$ data replicas that store the value associated with the latest timestamp (in the vein of [2, 14]); and (b) an array of SWMR safe wait-free storage objects (denoted by MDS_{hash}), which each stores a hash of a value associated with a given timestamp ts, i.e., timestamps are used as indices for the MDS_{hash} array. Every client may write to and read from MDS_{dir}, but the entries of MDS_{hash} are written only once by a single client. Timestamps in *MDStore* are classical multi-writer timestamps [4,5], comprised of an integer num and a process identifier cid that serves to break ties. Their comparison 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 $ts_1.cid > ts_2.cid$.

The *MDStore* client pseudocode is given in Algorithm 1 with data replica pseudocode given in Algorithm 2. On a high level, WRITE(v) proceeds as follows: (i) a writer w reads from MDS_{dir} to determine the latest timestamp ts (Alg. 1, lines 14–18); (ii) w increments ts and writes the hash of value v to $MDS_{hash}[ts]$ (Alg. 1, lines 19–20); (iii) w sends a write message to all data replicas containing (ts, v) and waits for a set Q of $t+1$ data replicas to reply (Alg. 1, lines 21–24); (iv) w writes (ts, Q) to MDS_{dir} where Q is a set of $t+1$ data replicas that have responded previously (Alg. 1, line 25–26); and (v) w sends a commit message to allow data replicas to garbage collect the data with timestamp less than ts (Alg. 1, lines 27–28).

On the other hand, a reader r upon invoking a READ: (i) reads from MDS_{dir} the latest authoritative metadata md with latest timestamp $md.ts$ and a set $md.replicas$ containing the identifiers of $t+1$ data replicas that store the latest value (Alg. 1, lines 33–34); and (ii) sends a read message to $md.replicas$ to read timestamp/value pairs not older than $md.ts$. Since clients do not trust replicas, reader r needs to *validate* every timestamp/value received in a readVal message sent by a data replica in response to a read

Algorithm 1 Algorithm of client c.

```
1: Types:
2:    TS: $(\mathbb{N}_0 \times \{\mathbb{N}_0 \cup \bot\})$ with fields num and cid        // timestamps
3:    TSVals: $(TS \times \{V \cup \bot\})$ with fields ts and val
4:    TSMeta: $(TS \times 2^{\mathbb{N}}) \cup \{\bot\}$ with fields ts and replicas
5: Shared objects:
6:    $MDS_{dir}$, is a MWMR atomic wait-free timestamped storage object storing $x \in$ TSMeta
7:    $MDS_{hash}[ts \in TS]$ is an array of SWMR safe wait-free storage objects storing $x \in H(V)$
8: Client state variables:
9:    $md$: TSMeta, initially $\bot$
10:   $ts$: TS, initially $(0, \bot)$
11:   $Q$ : $2^{\mathbb{N}}$, initially $\emptyset$
12:   readval: TSVals $\cup \{\bot\}$, initially $\bot$

13: operation WRITE($v$)
14:    $md \leftarrow MDS_{dir}$.TSREAD()
15:    if $md = \bot$ then
16:       $ts \leftarrow (0, c)$
17:    else
18:       $ts \leftarrow md.ts$
19:    $ts \leftarrow (ts.num + 1, c)$
20:    $MDS_{hash}[ts]$.WRITE($H(v)$)
21:    $Q \leftarrow \emptyset$
22:    for $1 \leq i \leq D$ do
23:       send write$\langle ts, v\rangle$ to $d_i$
24:    wait until $|Q| \geq t + 1$
25:    $md \leftarrow (ts, Q)$
26:    $MDS_{dir}$.TSWRITE($md$)
27:    for $1 \leq i \leq D$ do
28:       send commit$\langle ts\rangle$ to $d_i$
29:    return OK

30: upon receiving writeAck$\langle ts\rangle$ from replica $d_i$
31:    $Q \leftarrow Q \cup \{i\}$

32: operation READ()
33:    readval $\leftarrow \bot$
34:    $md \leftarrow MDS_{dir}$.TSREAD()
35:    if $md = \bot$ then
36:       return $\bot$
37:    for $i \in md.replicas$ do
38:       send read$\langle md.ts\rangle$ to $d_i$
39:    wait until readval $\neq \bot$
40:    return readval.val

41: upon receiving readVal$\langle ts', v'\rangle$ from replica $d_i$
42:    if readval $= \bot$ then
43:       if $ts' = md.ts$ then
44:          CHECK($ts', v'$)
45:       if $ts' > md.ts$ then
46:          $md' \leftarrow MDS_{dir}$.TSREAD()
47:          if $md'.ts \geq ts'$ then
48:             CHECK($ts', v'$)

49: procedure CHECK($ts, v$)
50:    if $H(v) = MDS_{hash}[ts]$.READ() then
51:       readval $\leftarrow (ts, v)$
```

message (Alg. 2, lines 55–58). To this end, readers consult the metadata service (Alg. 1, lines 41–51): (i) in case the timestamp received from a data replica ts' equals the timestamp in $md.ts$ (Alg. 1, line 43) then the reader only checks whether the value has indeed been written by reading $MDS_{hash}[md.ts]$ and comparing this to the hash of the received value; otherwise (ii), i.e., when $ts' > md.ts$ (Alg. 1, line 45), the reader first validates ts' itself by checking if MDS_{dir} points to ts' or even a later timestamp, and, if yes, proceeds to check the integrity of the value by comparing its hash to the value in $MDS_{hash}[ts']$.

3.2 *MDStore* details

We further illustrate *MDStore* using an execution ex, depicted in Figure 1. In ex, we assume $t = 1$ and hence $D = 3$ data replicas. In ex, data replica d_1 due to asynchrony does not receive messages in a timely manner, whereas data replica d_3 is Byzantine.

Execution ex starts with a complete $wr_1 =$ WRITE(v_1) which stores (ts_1, v_1) into data replicas $\{d_2, d_3\}$,

Algorithm 2 Algorithm of data replica d_i.

```
52: Server state variables:
53:     $data$: $2^{TSVals}$, initially $\emptyset$
54:     $ts$: $TS$, initially $(0, \perp)$

55: upon receiving read$\langle ts' \rangle$ from client $c$
56:     if $ts' < ts$ then $ts' \leftarrow ts$
57:     $v \leftarrow v' \in V : (ts', v') \in data$
58:     send readVal$\langle ts', v \rangle$ to $c$

59: upon receiving write$\langle ts', v' \rangle$ from client $c$
60:     if $ts' > ts$ then
61:         $data \leftarrow data \cup \{(ts', v')\}$
62:     send writeAck$\langle ts' \rangle$ to client $c$

63: upon receiving commit$\langle ts' \rangle$ from client $c$
64:     if $ts' > ts \land \exists (ts', \cdot) \in data$ then
65:         $ts \leftarrow ts'$
66:         $data \leftarrow data \setminus \{(ts', \cdot) \in data : ts' < ts\}$
```

where ts_1 is a pair $(1, w_1)$ that writer w_1 generated in line 19 of wr_1. Notice that WRITE wr_1 is not explicitly shown in Figure 1; however, the states of MDS_{dir} and $MDS_{hash}[ts_1]$ upon completion of wr_1 are shown.

In ex, the initial wr_1 is followed by two concurrent operations depicted in Figure 1: (i) a $wr_2 = \text{WRITE}(v_2)$ by writer w_2, and (ii) READ rd by reader r_1. Upon invoking wr_2, writer w_2 in Step ① (we refer to numbers in Fig. 1) first reads from MDS_{dir} the latest timestamp by invoking $MDS_{dir}.\text{TSREAD}()$ (line 14). MDS_{dir} eventually responds and w_2 reads timestamp $ts_1 = (1, w_1)$. Then, writer w_2 increments the timestamp and adds its own identifier (line 19) to obtain timestamp $ts_2 = (2, w_2)$. Then, writer w_2 invokes $MDS_{hash}[ts_2].\text{WRITE}(H(v_2))$ where $H(v_2)$ is a hash of written value v_2 (line 20, Step ②). Values written to MDS_{hash} serve to ensure integrity in the presence of potentially Byzantine data replicas; a writer writes to MDS_{hash} before exposing the current WRITE to other clients by writing to MDS_{dir} in order to prevent Byzantine replicas forging values with a given timestamp. Eventually, MDS_{hash} responds and writer w_2 then sends a write$\langle ts_2, v_2 \rangle$ message to all data replicas containing an entire value v_2 (lines 22–23, Step ③). In ex, write messages are received only by data replicas d_2 and d_3 (which is Byzantine). A correct replica d_2 simply adds the pair (ts_2, v_2) to its $data$ set (line 61) but d_2 does not update its local authoritative timestamp ts which still reflects ts_1. At this point in time of execution ex, we make writer w_2 wait for asynchronous replies from data replicas.

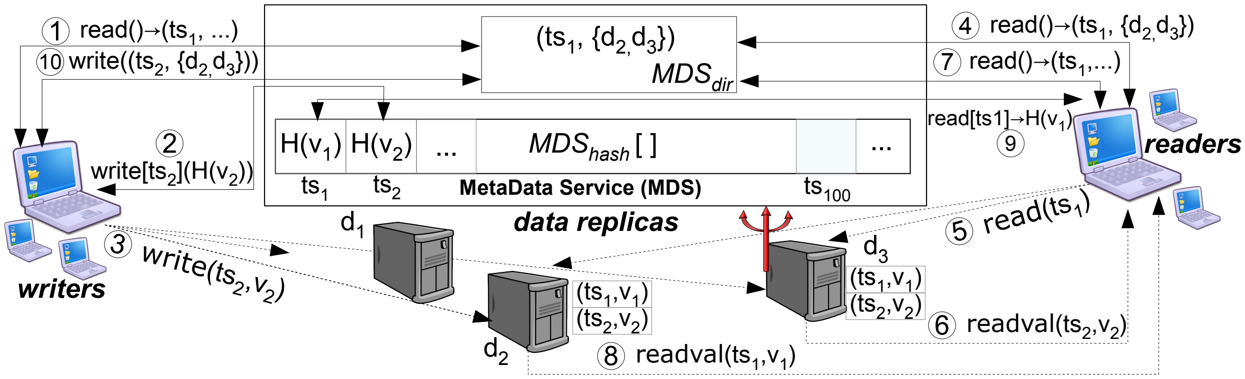

Figure 1: Illustration of a *MDStore* execution with a concurrent WRITE and READ.

At the same time, concurrently with wr_2, reader r_1 invokes READ rd. Reader r_1 first queries MDS_{dir} for metadata md by invoking $MDS_{dir}.\text{TSREAD}()$, to determine the latest timestamp $md.ts$ and the set of data replicas $md.replicas$ that store the latest value (line 34, Step ④). MDS_{dir} eventually responds and r_1 sees $md.ts = ts_1$ and $md.replicas = \{d_2, d_3\}$. Then, r_1 sends read message to data replicas d_2 and d_3 (lines 37–38, Step ⑤). By the algorithm, a data replica d replies to a read message with a readVal message containing the value associated with its local authoritative timestamp ts, which does not necessarily reflect the highest timestamp that replica d stores in $data$; e.g., in case of d_2 (and d_3) in ex, ts equals ts_1 and not ts_2. However, a Byzantine d_3 could mount a sophisticated attack and respond with the pair (ts_2, v_2) (Step ⑥); although this pair is in fact written concurrently, it is dangerous for r_1 to return v_2 since, in *MDStore* readers do not write back data and the value v_2 has not been completely written — this may violate atomicity. To prevent this attack, a reader invokes $MDS_{dir}.\text{TSREAD}()$ to determine whether ts_2 (or a higher timestamp) became authoritative in the mean time (lines 45–43, Step ⑦). Since this is not the case, r_1 discards the reply from d_3 and waits for an additional reply (from d_2).

An alternative attack by Byzantine d_3 could be to make up a timestamp/value pair with a large timestamp, say ts_{100}. In this case, r_1 would also first check with MDS_{dir} whether ts_{100} or a higher timestamp has been written (just like in Step ⑦). However, if so, r_1 would then proceed to check the integrity of a hash of the value reported by d_3 by invoking $MDS_{hash}[ts_{100}].\text{READ}()$ (lines 49–51); this check would assuming a bounded adversary as the hash function is collision-free.

In ex, eventually d_2 responds to r_1 with pair (ts_1, v_1) (lines 55–58, Step ⑧). By the protocol (optimizations omitted for clarity) reader r_1 verifies the integrity of v_1 by reading a hash from $MDS_{hash}[ts_1]$ (lines 49–51, Step ⑨). This time, the check succeeds and rd completes returning value v_1.

Eventually, writer w_2 receives writeAck replies in wr_2 from replicas d_2 and d_3. Then, writer w_2 invokes $MDS_{dir}.\text{TSWRITE}(ts_2, \{d_2, d_3\})$ (lines 25–26, Step ⑩) only now, when the write wr_2 finally "takes effect", i.e., at the linearization point of WRITE which coincides with the linearization point of the TSWRITE to MDS_{dir}. Finally, the writer sends a commit message to all replicas to allow them to garbage collect stale data (lines 27–28); notice that data replicas update their local variable ts, which reflects a value they will serve to a reader, only upon receiving a commit message (lines 63–66).

Finally, we point out that *MDStore* uses timestamped storage (MDS_{dir}) as a way to avoid storing entire history of a shared variable at data replicas. We could not achieve this with MDS_{dir} being a classical storage object, since such a classical storage object would allow overwrites of MDS_{dir} with a lower timestamp. With our protocol at data replicas (notably lines 59–62) and our goal of not storing entire histories, such an overwrite could put MDS_{dir} in inconsistent state with data replicas.

3.3 Metadata service implementations

We show how to implement the *MDStore* metadata service from existing asynchronous BFT storage protocols that rely on $3t + 1$ replicas — in our case these are exactly $3t_M + 1$ metadata replicas. To qualify for reuse, existing BFT protocols should also tolerate an arbitrary number of Byzantine readers, any number of crash-faulty writers, and, ideally, make no cryptographic assumptions.

First, it is critical to see that MDS_{dir}, our MWMR atomic wait-free timestamped storage, can be implemented as a straightforward extension of the classical SWMR to MWMR atomic storage object transformation (e.g., [5, page 163]). In this transformation, there is one SWMR storage object per writer and writers store timestamp/value pairs in "their" storage object, after first reading and incrementing the highest timestamp found in any other storage object. In this 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 (with $3t+1$ replicas); examples include [3, 11] (with an unbounded adversary) and [7, 29] (with a bounded adversary).

Second, MDS_{hash} is an array of SWMR safe storage objects that can directly be implemented from the protocols with atomic semantics mentioned above, or even from protocols with weaker implementations, such as (i) SWMR safe wait-free storage [1] or (ii) its regular variant, both without cryptographic assumptions [16], or (iii) regular storage with digital signatures [28].

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

4 Lower bounds

In this section we prove two lower bounds: (i) we show that using $2t + 1$ data replicas to tolerate t data replica *crash* faults is necessary implementing distributed single-writer single-reader obstruction-free safe storage, even with the help of a *metadata service oracle*; and (ii) we also show that the same result extends to $3t + 1$ replicas in the model with Byzantine data replicas. However, this second lower bound applies in the model with an *unbounded adversary* and does not hold when the adversary is *bounded*, i.e., when it cannot break cryptographic hash functions (see Sec. 2).

Technically, we unify the two results into one single argument in a *hybrid failure* model, where we consider $D = 2t + b + 1$ data replicas, out of which up to b can be Byzantine and up to $t - b$ can only crash. For the purpose of this proof, we focus on the model with a single writer.

Preliminaries. Our lower bound model assumes a *metadata service* (Def. 4.1): in short, a metadata service is an oracle, modeled as a *correct* process.[3] A metadata service is parameterized by the domain of values V of the implemented storage. Roughly speaking, a metadata service can implement an arbitrary functionality, except that it might not be able to help a reader distinguish whether the writer wrote value $v \in V$ or value $v' \in V$, where $v \neq v'$.

Definition 4.1 (Metadata service) *A metadata service for a value domain V (denoted by MDS$_V$) is a* correct *process that can implement an arbitrary automaton with the following limitation. There exist two values $v, v' \in V$ (we say they are* indistinguishable to *MDS$_V$), such that there is no distributed storage algorithm for MDS$_V$, the writer and a set of processes P, such that some process $p \in P$ can distinguish execution ex_v from $ex_{v'}$, where:*

- *In ex_v, the writer invokes a complete* WRITE(v) *and crashes, such that no process in P receives any message from the writer in ex_v; and*
- *In $ex_{v'}$, the writer invokes a complete* WRITE(v') *and crashes, such that no process in P receives any message from the writer in $ex_{v'}$.*

Intuitively, Definition 4.1 models metadata service as a general oracle with arbitrary functionality, with the restriction that it cannot store or relay data values in V. Observe that if we extend both executions ex_v and $ex_{v'}$ in Definition 4.1 by appending a READ by a correct reader (from P), to obtain partial executions ex'_v and $ex'_{v'}$, respectively, obstruction-freedom or safety is violated in ex'_v or in $ex'_{v'}$.

To state our lower bound precisely, we change the model of Section 2 and assume that, out of D data replicas, up to b can be Byzantine and additionally $t - b$ of them are benign (that is, they may crash), for $0 \leq b \leq t$. We assume an *unbounded adversary* that can coordinate Byzantine processes and that either knows values v and v' that are indistinguishable to *MDS$_V$*, or can compute such a v given v', or vice-versa.

We now state the main result of this section:

[3] In our proof we do not use metadata replicas (defined in Section 2) which are "replaced" by a metadata service oracle.

Theorem 4.2 *Assuming an umbounded adversary, there is no asynchronous distributed algorithm that implements single-writer single-reader (SWSR) obstruction-free safe storage (with domain V), with $D \leq 2t+b$ data replicas and a metadata service for V.*

Proof: Assume by contradiction that such implementation I exists. We develop a series of executions of I to show that at most $2t + b$ data replicas do not help the reader distinguish the values indistinguishable to *MDS*$_V$, v and v'. To this end, we divide the set of data replicas in three disjoint *groups* T_1 and T_2, each containing at most t data replicas, and group B with at most b data replicas.

Consider first partial execution ex_1 in which the reader and the replicas from group T_1 crash at the beginning of ex_1 and the writer invokes WRITE(v). By obstruction-freedom wr_1 eventually completes. Then, the writer crashes and ex_1 ends at time t_1. In ex_1, the reader and data replicas from T_1 do not deliver any message, whereas the writer, *MDS*$_V$ and data replicas from $T_2 \cup B$ deliver all the messages per implementation I. We denote the state of data replicas from group B at t_1 by σ_v.

Let ex_2 be a partial execution in which the writer invokes WRITE(v') that ends at time t_2, such that ex_2 is otherwise similar to ex_1, with the reader and replicas from T_1 crashing at the beginning of ex_2 and the other processes delivering all messages. We denote the state of data replicas from group B at t_2 by $\sigma_{v'}$.

Let ex'_1 be a partial execution similar to ex_1, except that the reader and the data replicas from T_1 do not crash, yet they still do not receive any message by time t_1 (due to asynchrony). At time t_1, data replicas from T_2 crash. This is followed by READ rd_1 by the reader at $t_3 > max\{t_1, t_2\}$. The reader and data replicas in T_1, never receive any message from faulty data replicas from T_2 or the faulty writer. By obstruction-freedom rd_1 eventually completes (at t_4) and, by safety, returns the value written by wr_1, i.e., v.

Let ex'_2 be a partial execution similar to ex_2, except that the reader and the replicas from T_1 are not faulty, yet they still do not receive any message by time t_2 (due to asynchrony). At time t_2, data replicas from B (if any) exhibit a Byzantine fault, by *changing their state from $\sigma_{v'}$ to σ_v* (see ex_1). After this, data replicas from B follow the protocol. This is followed by a READ rd_2 by the reader at t_3. Moreover, assume that due to asynchrony, *MDS*$_V$, the reader and data replicas in T_1, do not receive any message from data replicas from T_2 until after t_4. Notice that, by Definition 4.1 and since they do not receive any message from the writer or data replicas in T_2, the reader and the data replicas in T_1 cannot distinguish ex'_2 from ex'_1. Hence, in ex'_2, rd_2 returns v (at t_4) like in ex'_1. However, this violates safety by which rd_2 must return v'. A contradiction. □

Discussion. We make two observations about Theorem 4.2. First, in the crash model, where $b = 0$, Theorem 4.2 implies that $2t+1$ data replicas are necessary for implementing SWSR obstruction-free safe storage, even with a metadata service oracle. Second, notice that the Byzantine part of the proof critically relies on the ability of the adversary to successfully switch from the state where Byzantine replicas "observed" v' to the state where Byzantine replicas seemingly have "observed" v (see ex'_2). In practice, when assuming a bounded adversary, cryptographic hash functions easily prevent this attack — the proof of Theorem 4.2 breaks down for $b > 0$. Protocols in this realistic model, including *MDStore*, are only subject to the lower bound of $2t + 1$ data replicas from the crash model.

5 Related work

The read/write storage abstraction (also known as a *register*) was formalized by Lamport [23]. Martin et al. [30] demonstrated a tight lower bound of $3t + 1$ replicas needed for any register implementation that tolerates t Byzantine replicas in an asynchronous system. Their bound applies even to single-writer single-reader safe register, where the reader and the writer are benign. In this paper, we refine this bound by logically separating storage replicas into data replicas and metadata replicas. With such a separation, we show that the $3t + 1$ lower bound of [30] applies to register metadata replicas only, but it does not hold for the number of data replicas. Only $2t + 1$ data replicas are needed to tolerate t Byzantine data replicas in an asynchronous system with a bounded adversary.

Protocol *MDStore* that matches this lower bound is similar in style to Farsite [2], a BFT file service, and Hybris [12], a recent hybrid cloud storage system. Namely, like *MDStore*, Farsite and Hybris separate metadata from data and keep cryptographic hashes and the directory information as metadata and require at least $2t + 1$ data replicas. However, unlike *MDStore*, Farsite and Hybris metadata services are based on replicated state machines; hence both Farsite and Hybris are subject to the FLP impossibility result [14] and require stronger timing assumptions, such as partial synchrony [11]. In addition, Farsite supports single-writer and multiple readers and uses read/write locks for concurrency control, whereas our *MDStore* supports multiple writers and offers wait-free [18] atomic semantics, without resorting to locks. On the other hand, Hybris only supports FW-terminating reads and is not wait-free.

Data and metadata have also been separated in asynchronous *crash-tolerant* storage [8, 14] and in variants of state-machine replication [33]. Interestingly, separating data from metadata does not reap benefits in terms of reduced resource costs with crash-faults: indeed all of the mentioned crash-tolerant protocols that exercise data/metadata separation [8, 14, 33] still need $2t + 1$ data replicas. We prove this inherent: even with a fault-free metadata service, $2t + 1$ data replicas are necessary to tolerate t data replica faults.

Separation of data from the control plane is well-known in consensus and state machine replication. Lamport's Paxos algorithm [24, 25] separated consensus roles into proposers, acceptors, and learners. In this context, the lower bound of $3t + 1$ replicas for partially synchronous BFT consensus was shown to apply only to acceptors [26], but not to proposers or learners. For example, [17] demonstrated a partially synchronous BFT consensus protocol in which any number of proposers and learners can be Byzantine. Yin et al. [34] apply the ideas from Lamport's consensus role separation and separate *agreement* from *execution* to obtain state machine replication protocols with $3t + 1$ agreement replicas and $2t + 1$ execution replicas. However, just like [2, 12], the results of [17, 34] that apply to state-machine replication and consensus are fundamentally different from ours; they are subject to the FLP impossibility result [15] and the protocols therefore rely on stronger timing assumptions [13].

6 Conclusion and future work

This paper presents *MDStore*, the first asynchronous BFT storage protocol that uses $2t + 1$ data replicas to tolerate t Byzantine faults in a general model without trusted components. To achieve this, *MDStore* separates data from metadata and stores metadata leveraging a novel abstraction we call *timestamped storage* which can can be implemented using existing asynchronous BFT storage protocols that need $3t + 1$ replicas to tolerate t Byzantine faults. In addition, *MDStore* implements strong guarantees such as wait-freedom and atomicity (linearizability). Finally, *MDStore* relies on collision-resistant cryptographic hash functions which we show inherent. In this paper we show also that, perhaps surprisingly, no asynchronous crash-tolerant storage implementation can achieve better resilience with respect to data replicas than our BFT *MDStore*.

Our work opens many avenues for future work in BFT storage systems, especially for those of practical relevance. It requires to revisit other important aspects of asynchronous BFT storage, such as their complexity or erasure-coded implementations, which have been extensively studied in the traditional model with unified data and metadata.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[3] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded Wait-free Implementation of Optimally Resilient Byzantine Storage Without (Unproven) Cryptographic Assumptions. In *Proceedings of DISC*, 2007.

[4] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.

[5] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

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

[7] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 115–124, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Brian Cho and Marcos K. Aguilera. Surviving congestion in geo-distributed storage systems. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association.

[9] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proc. 21st Symposium on Operating Systems Principles*, SOSP '07, pages 189–204, 2007.

[10] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proc. 23rd Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183, 2004.

[11] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. Proofs of writing for efficient and robust storage. *CoRR*, abs/1212.3555, 2012.

[12] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Consistency hardening in robust hybrid cloud storage. Eurecom Research Report RR-13-291, 2013.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[14] Rui Fan and Nancy Lynch. Efficient Replication of Large Data Objects. In *Proc. DISC*, pages 75–91, 2003.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):372–382, Apr 1985.

[16] Rachid Guerraoui and Marko Vukolić. How Fast Can a Very Robust Read Be? In *Proceedings of PODC*, pages 248–257, 2006.

[17] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

[18] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[19] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, 2003.

[20] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[21] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proc. EuroSys*, pages 295–308, 2012.

[22] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.

[23] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(2):77–101, 1986.

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

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

[26] Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, LNCS, pages 22–23. Springer Verlag, May 2003.

[27] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[28] Dahlia Malkhi and Michael K. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.

[29] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. In *Proc. SRDS*, pages 51–58, 1998.

[30] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine Storage. In *Proceedings of DISC*, pages 311–325, 2002.

[31] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.

[32] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[33] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX ATC*, 2012.

[34] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Michael Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, pages 253–267, 2003.

A Correctness of *MDStore*

In this section we prove that the pseudocode in Algorithm 1 and Algorithm 2 is correct by showing that it satisfies atomicity and wait-freedom.

Definition A.1 (Timestamp of an Operation) *If o is an operation, then we define the timestamp of o, denoted $ts(o)$ as follows. If o is a* WRITE *operation, then $ts(o)$ is ts when its assignment completes in line 19. Else, if o is* READ *operation, then $ts(o)$ is the timestamp associated to $readval$ when its assignment completes in line 51.*

Lemma A.2 (Timestamped Storage Safety) *Let x be a timestamped storage object and let $tsrd$ be an operation x.*TSREAD *returning (ts', v'). If $tsrd$ follows after an operation x.*TSWRITE*$((ts, v))$ or after an operation x.*TSREAD *returning (ts, v), then $ts' \geq ts$.*

Proof: Follows from the sequential specification of timestamped storage. □

Lemma A.3 (Sandwich) *Let rd be a complete* READ *operation and let md and md' be the value returned by MDS_{dir} in lines 34 and 46 respectively, Then $md.ts \leq ts(rd) \leq md'.ts$.*

Proof: By Definition A.1, $ts(rd)$ is *readval.ts* when the assignment in line 51 completes. For this to happen, either the condition in line 43 or line 45 must be satisfied. This is implies that either $ts(rd) = md.ts$ or $md.ts < ts(rd) \leq md'.ts$. □

Lemma A.4 (Partial Order) *Let o and o' be two operations with timestamps $ts(o)$ and $ts(o')$, respectively, such that o precedes o'. Then $ts(o) \leq ts(o')$ and if o' is a* WRITE *operation then $ts(o) < ts(o')$.*

Proof: Let o' be a READ (resp. a WRITE) operation. By Definition A.1 and Lemma A.3, $ts(o') \geq o'.md.ts$ ($o.$ denotes the context of operation o). In the following we distinguish whether o is a WRITE or a READ.
Case 1 (o is a WRITE): if o is a WRITE, then $o.MDS_{dir}$.TSWRITE$(o.md)$ in line 26 precedes $o'.md \leftarrow o'.MDS_{dir}$.TSREAD$()$ in line 34 (resp. 14). By Lemma A.2, it follows that $o'.md.ts \geq o.md.ts$. By Definition A.1 $o.md.ts = ts(o)$, and therefore $o'.md.ts \geq ts(o)$. There are two possible subcases; either o' is a READ or a WRITE. If o' is a READ then $ts(o') \geq o'.md.ts$, and therefore $ts(o') \geq ts(o)$. Otherwise, if o' is a WRITE, then $ts(o') > o'.md.ts$ because $ts(o')$ is obtained from incrementing the first component of $o'.md.ts$. Therefore, $ts(o') > o'.md.ts \geq ts(o)$.
Case 2 (o is a READ): if o is a READ, then by Definition A.1 and Lemma A.3, $o.md.ts \leq ts(o) \leq o.md'.ts$. In what follows, we treat the only two possible cases $ts(o) = o.md.ts$ and $o.md.ts < ts(o) \leq o.md'.ts$ separately.

(2a): if $ts(o) = o.md.ts$, then since $o'.md \leftarrow o'.MDS_{dir}$.TSREAD$()$ in line 34 (resp. 14) follows after $o.md \leftarrow o.MDS_{dir}$.TSREAD$()$, by Lemma A.2 $o'.md.ts \geq o.md.ts$. Since $o.md.ts = ts(o)$, it follows that $o'.md.ts \geq ts(o)$. If o' is a READ, then $ts(o') \geq o'.md.ts$, and we conclude that $ts(o') \geq ts(o)$. Otherwise, if o' is a WRITE, then $ts(o') > o'.md.ts$ and therefore, $ts(o') > ts(o)$.

(2b): if $o.md.ts < ts(o) \leq o.md'.ts$, then $o.md' \leftarrow o.MDS_{dir}$.TSREAD$()$ in line 46 precedes $o'.md \leftarrow o'.MDS_{dir}$.TSREAD$()$ in line 34 (resp. 14). By Lemma A.2 $o'.md.ts \geq o.md'.ts$ and since $o.md'.ts \geq ts(o)$, it follows that $o'.md.ts \geq ts(o)$. If o' is a READ, then $ts(o') \geq o'.md.ts$, and we conclude that

$ts(o') \geq ts(o)$. Otherwise, if o' is a WRITE, then $ts(o') > o'.md.ts$ and therefore, $ts(o') > ts(o)$, which completes the proof. □

Lemma A.5 (Unique Writes) *If o and o' are two* WRITE *operations with timestamps $ts(o)$ and $ts(o')$, then $ts(o) \neq ts(o')$.*

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 A.4, $ts(o') \neq ts(o)$. □

Lemma A.6 (Integrity) *Let rd be a* READ *operation with timestamp $ts(rd)$ returning value $v \neq \perp$. Then there is a single* WRITE *operation wr of the form* WRITE*(v) such that $ts(rd) = ts(wr)$.*

Proof: Since rd returns v and has an associated timestamp $ts(rd)$, rd receives $(ts(rd), v)$ from one of the data replicas. Suppose for the purpose of contradiction that v is never written. Then, then by the collision resistance of H, the check in line 50 does not pass and rd does not return v. Therefore, we conclude that some operation wr sends a message write$\langle ts(rd), v\rangle$ in line 23. Since $ts(wr)$ is set only once during the execution of a WRITE and that occurs in line 19, it follows that $ts(wr) = ts(rd)$. Finally, by Lemma A.5 no other write has the same timestamp, which completes the proof. □

Theorem A.7 (Atomicity (Linearizability)) *Every execution ex of Algorithm 1 and Algorithm 2 satisfies atomicity.*

Proof: Let ex be an execution of the algorithm. By Lemma A.6 the timestamp of a READ operation either has been written by some WRITE operation or the READ returns $\perp$.

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

To prove that π preserves the sequential specification of a MWMR register we must show that a READ always returns the value written by the latest preceding write which appears before it in π, or the initial value of the register $\perp$ if there is no preceding write in π. Let rd be a READ operation returning a value v. If $v = \perp$, then by construction rd is ordered before any WRITE in π.

Otherwise, $v \neq \perp$ and by Lemma A.6 there exists a WRITE(v) operation, with the same timestamp, $ts(rd)$. In this case, this write is placed in π before rd, by construction. By Lemma A.5, other write operations in π have a different associated timestamp and therefore appear in π either before WRITE(v) or after rd.

It remains to show that π preserves real-time order. Consider two complete operations o and o' in ex' such that o precedes o'. By Lemma A.4, $ts(o') \geq ts(o)$. If $ts(o') > ts(o)$ then o' appears after o in π by construction. Otherwise $ts(o') = ts(o)$ and by Lemma A.4 it follows that o' is a READ operation. If

o is a WRITE operation, then o' appears after o since we placed each read after the WRITE with the same timestamp. Otherwise, if o is a READ, then it appears before o' as in ex'. □

Theorem A.8 (Wait-Freedom) *The protocol comprising Algorithm 1 and Algorithm 2 satisfies wait-freedom.*

Proof: Since the shared storage objects used in Algorithm 1 are wait-free, every READ or WRITE operation invoked on MDS_{dir} and $MDS_{hash}[ts]$, where $ts \in TSVals$, eventually completes. It remains to show that no WRITE (resp. READ) operation blocks in line 24 (resp. 39). For a WRITE operation wr, the waiting condition in line 24 is eventually satisfied because there is a time after which all correct data replicas reply and there are at least $t + 1$ such replicas. On the other hand, let rd be a READ operation and suppose for the purpose of contradiction that the waiting condition in line 39 is never satisfied, and therefore *readval* is never set in line 51. Let d_i be a correct data replica such that $i \in md.replicas$. Since rd did previously sent a read$\langle md.ts \rangle$ message to d_i, eventually rd receives a reply from d_i consisting of a pair (ts', v) in line 41.

If ts' satisfies $md.ts \leq ts' \leq md'.ts$, then since d_i is a correct replica, the condition in line 50 is also satisfied, and therefore *readval* is set in line 51. Suppose for the purpose of contradiction that $ts' < md.ts$ or $ts' > md'.ts$. Notice that the requested timestamp is $md.ts$. If $ts' < md.ts$ then d_i replied with a smaller timestamp than $md.ts$. However, notice that according to the check in the replica code in line 56, d_i never replies with a timestamp smaller than the requested timestamp, contradicting our assumption. Otherwise, if $ts' > md'.ts$, then by Lemma A.3 $ts' > md.ts$, and therefore d_i replies with its local timestamp ts. According to the replica code, line 65 is the only place where ts is changed. Furthermore, if ts changes to $ts(wr')$ then wr' is a WRITE operation that committed. According to the WRITE code, wr' commits only after writing $ts(wr')$ to MDS_{dir}. Hence, if $ts' > md'.ts$, then rd invokes MDS_{dir} in line 46 and does so only after the corresponding WRITE wrote ts' to MDS_{dir}. By Lemma A.2, MDS_{dir} returns in line 46 a value whose timestamp is a least ts', which means that $md'.ts \geq ts'$, a contradiction. □