# Abortable Fork-Linearizable Storage*

Matthias Majuntke, Dan Dobre, Marco Serafini, and Neeraj Suri

{majuntke,dan,marco,suri}@cs.tu-darmstadt.de
TU Darmstadt, DEEDS Group,
Hochschulstraße 10, 64289 Darmstadt, Germany

**Abstract.** We address the problem of emulating a shared read/write memory in a message passing system using a storage server prone to Byzantine failures. Although cryptography can be used to ensure confidentiality and integrity of the data, nothing can prevent a malicious server from returning obsolete data. Fork-linearizability [1] guarantees that if a malicious server hides an update of some client from another client, then these two clients will never see each others' updates again. Fork-linearizability is arguably the strongest consistency property attainable in the presence of a malicious server. Recent work [2] has shown that there is no fork-linearizable shared memory emulation that supports *wait-free* operations. On the positive side, it has been shown that *lock-based* emulations exist [1,2]. Lock-based protocols are fragile because they are blocking if clients may crash. In this paper we present for the first time *lock-free* emulations of fork-linearizable shared memory. We have developed two protocols, Linear and Concur. With a correct server, both protocols guarantee linearizability and that every operation successfully completes in the absence of step contention, while interfering operations terminate by aborting. The Concur algorithm additionally ensures that concurrent operations invoked on different registers complete successfully.

**Keywords:** Fork-linearizability, abortable objects, lock-freedom, shared memory, online collaboration

## 1 Introduction

Fast broadband access to the Internet allows users to benefit from online services such as storing their data remotely and sharing it with other users. Examples for such services, also known as storage or computing "clouds" are Amazon S3, Nirvanix CloudNAS, and Microsoft SkyDrive [3]. These services offer full data administration such that a user does not need to care for backups or server maintenance and the data is available on demand. Such an infrastructure makes online collaboration (multiple users working on the same logical data) based on

* Research funded in part by IBM Faculty Award, Microsoft Research, and DFG GRK 1362 (TUD GKmM).

shared storage very attractive. Examples of existing solutions for online collaboration are the well-known revision control systems like CVS [4] and SVN [5], the storage management system WebDAV [6], upcoming Web 2.0 applications [7] like Google docs [8], and a large number of distributed file systems [9].

Online collaboration usually assumes that the participating clients trust each other — otherwise there exists no basis for reasonable communication. However, when the shared storage is provided by a third party, clients may not fully trust the service, e.g. it can corrupt or leak sensitive data. Cryptographic techniques such as hash functions, message authentication codes (MACs) and signatures can be used to prevent unauthorized access to data (confidentiality) and undetectable corruption of the data (integrity). Progress and consistency cannot always be guaranteed when the storage service[1] is untrusted. A malicious server may simply refuse to process client requests and it can violate linearizability by omitting a recent update of one client and presenting an outdated value to another client. This split brain attack is called *forking* and cannot be prevented. However, once a forking attack is mounted, it can be easily detected using a *fork-linearizable* storage protocol. *Fork-linearizability* [1] ensures that once two clients are forked, they never see each others' updates after that without revealing the server as faulty. Without fork-consistency, a malicious server is able to present data updates to clients in such a way that no client can say whether the set of updates of other clients it sees is complete or not, nor can such malicious behavior be easily detected, making reliable collaboration impossible. Once such a partitioning occurs, the clients stop hearing from each other. A client that has not seen updates from another client for a while can use out-of-band communication (as e.g. phone or e-mail) to find out if the server is misbehaving.

Recent work [2] has shown that even if the server behaves correctly, clients cannot complete their operations independently from each other because this introduces a vulnerability that can be exploited by a Byzantine server to violate fork-linearizability. This means that in an asynchronous system there is no *wait-free* [10] emulation of fork-linearizable storage on a Byzantine server. On the positive side, the SUNDR [1] protocol and the concurrent protocol by Cachin *et al.* [2] show the existence of fork-linearizable Byzantine emulations using locks. However, lock-based protocols are problematic as they can block in the presence of faulty clients that crash while holding the lock.

*Paper Contributions* In this paper we present two *lock-free* emulations of fork-linearizable shared memory on an untrusted server. In runs in which the server behaves correctly, our proposed protocols LINEAR and CONCUR ensure linearizability [11], and that each operation executed in the absence of concurrency successfully completes. Under concurrency, operations may complete by aborting. Both protocols emulate a shared memory consisting of $n$ single-writer multiple-reader (SWMR) registers, one for each of the $n$ clients, where register $i$ is updated only by client $C_i$ and may be read by all clients. While both protocols address lock-free fork-linearizability, they solve two distinct issues. The LIN-

[1] We will use the terms storage service, storage server, and server interchangeably.

EAR protocol, which is the first *lock-free* fork-linearizable implementation at all, offers a communication complexity of $\mathcal{O}(n)$. The CONCUR protocol improves on the handling of concurrent operations such that overlapping operations accessing *different* registers are not perceived as concurrent, and therefore they are not aborted. However, it has a communication complexity of $\mathcal{O}(n^2)$. Both protocols allow concurrent operations to abort in order to circumvent the impossibility result by Cachin *et al.* [2]. The necessary condition for aborting is step contention [12], and thus, pending operations of crashed clients never cause other operations to abort. As a final contribution, note that the existence of abortable fork-linearizable storage implies the existence of obstruction-free [13] fork-linearizable storage.

We now give a rough intuition of why aborting helps to circumvent the given impossibility of wait-free fork-linearizability. With both our protocols, if multiple operations compete for the same register, then there is only one winner and all other operations are aborted. On a correct server, this strategy ensures that all successful operations applied to the same register access the register sequentially. Operations have timestamps attached to them and the sequential execution establishes a total order on operations and the corresponding timestamps. The algorithm ensures that a forking attack breaks the total order on timestamps. If a malicious server does not present the most recent update to a read operation, then the timestamps of the omitted write operation and that of the read operation become incomparable and the two clients are forked. The algorithm guarantees that also future operations of those two clients cannot be ordered and thus they remain forked forever.

## 2 Related Work

Mazières and Shasha [1] have introduced the notion of fork-linearizability and they have implemented the first fork-linearizable multi-user network file system SUNDR. The SUNDR protocol may block in case a client crashes even when the storage server is correct. Cachin *et al.* [2] implements a more efficient fork-linearizable storage protocol based on SUNDR which reduces communication complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The presented protocols are blocking and thus they have the same fundamental drawback as SUNDR. The authors [2] also prove that there is no wait-free emulation of fork-linearizable storage. They do so by exhibiting a run with concurrent operations where some client has to wait for another client to complete. Oprea and Reiter [14] define the weaker notion of fork-sequential consistency. Intuitively the difference to fork-linearizability is that fork-sequential consistency does not necessarily preserve the real-time order of operations from different clients. In a recent work, Cachin *et al.* [15] show that there is no wait-free emulation of fork-sequential consistent storage on a Byzantine server. It is important to note that these impossibility results do not rule out the existence of emulations of fork-linearizable storage with abortable operations [16] or weaker liveness guarantees such as obstruction-freedom [13]. Cachin *et al.* [17] presents the storage service FAUST which wait-free emulates a shared memory with a new consistency semantics called *weak fork-linearizability*. The notion of weak fork-linearizability weakens fork-linearizability in two fundamental ways. After being forked, two clients may see each others' updates once (at-most-on-join property) and secondly, the real-time order among the operations which are the last of each client is not ensured.

Li and Mazières [18] study systems where storage is implemented from $3f+1$ server replicas and more than $f$ replicas are Byzantine faulty. They present a storage protocol which ensures *fork* consistency*. Similar to weak fork-linearizability, fork* consistency allows that two forked clients may be joined at most once (at-most-one-join property).

The notion of abortable objects has been introduced by Aguilera *et al.* [16]. The paper shows the existence of a universal abortable object construction from abortable registers. It is the first construction of an obstruction-free universal type from base objects weaker than registers. In a follow-up paper [19] it has been shown that in a partially synchronous system, abortable objects can be boosted to wait-free objects. This makes abortable objects, including our abortable fork-linearizable read/write emulation very attractive.

Summing up, to date there is no lock-free emulation of fork-linearizable storage even though lock-free solutions can be made practically wait-free using boosting techniques as described by Aguilera *et al.* [19].

## 3 System Model and Definitions

Similar to the models used in recent work on fork-linearizability [2],[1], we consider a distributed system consisting of a single server $S$ and $n$ clients $C_1, \ldots, C_n$. The clients may fail by crashing but they never deviate from the protocol. The server may be faulty and deviate arbitrarily from its protocol exhibiting non-responsive-arbitrary faults [20] (Byzantine [21]). The clients communicate with the server by sending messages over reliable channels directly to the server, forming an asynchronous network. The *shared functionality* provided by the server is a *read/write register*. A read/write register provides *operations* by which the clients can access the register. An operation is defined by two *events*, an *invocation* event and a *response* event. To represent an abort of execution, there are two types of response events: ABORT and OK events respectively. An additional event type constitute *crash* events representing the act of a client failing. We call operation *op complete*, if there exists a matching response event to the invocation event of *op*, else *op* is denoted as *incomplete*. An operation is *successful*, iff it is complete and the response event is an OK event. An operation is *aborted*, if it is complete and the response event is an ABORT event. Operation *op precedes* operation $op'$ iff *op* is complete before the invocation event of $op'$. If *op* precedes $op'$ we denote *op* and $op'$ as *sequential* operations. Else, if neither operation precedes the other, then *op* and $op'$ are said to be are *concurrent*. An *execution* of the system is defined as the sequence of events occurring at the clients.

A read/write register $X[i]$ provides a *Read* and a *Write* operation to the clients. The response event to a client's operation is either OK or ABORT. Client

$C_i$ may use the *Write* operation to store a value $v$ from domain *Value* in register $X[i]$, denoted as $Write(i, v)$. If the response to a *Read* of register $X[i]$ is OK, then a value $v$ is returned, denoted as $Read(i) \rightarrow v$. The server implements $n$ single-writer multiple-reader (SWMR) registers $X[1 \ldots n]$ where each client $C_i$ writes only to $X[i]$ and may read from all other registers. The *sequential specification* of a register requires that if a *Read* operation returns a value, it returns the value written by the last preceding *Write* operation.

We assume that each client interacts *sequentially* with the read/write register, i.e. a client invokes a new operation only after the previous operation has completed.

Further we assume that clients have access to a digital signature scheme used by each client to *sign* its messages such that any other client can determine the authenticity of a message by *verifying* the corresponding signature. Further, the Byzantine server is not able to forge the signatures.

The consistency condition for the read/write register is defined in terms of the sequence $\sigma$ of events the shared register exhibits in an execution as observed by the clients. Such a sequence, also called *history*, contains invocation, response, and crash events. To ease the definition of consistency conditions and the reasoning about correctness, we define two transformations to derive simpler histories from more complicated ones, while maintaining plausibility of execution. Intuitively, the transformations remove all operations from a history that do not take effect.

**Definition 1** An operation *op* of client *takes effect* if and only if

1. *op* is *successful* OR
2. *op* is a *Write* operation *and*
   there exists a *Read* operation that returns the value written by *op*.

We now define the two transformations CrashComplete and AbortComplete.

**Definition 2** The transformations CrashComplete and AbortComplete take a sequence of events $\sigma$ as input and return a sequence of events $\sigma'$ as output.

- CrashComplete: We define $\sigma'$ returned by CrashComplete by construction: At first we add all events from $\sigma$ to $\sigma'$. Then, we remove the invocation events of *incomplete* operations that did not take effect and the corresponding crash event if one exists[2] from $\sigma'$. Next, we add a matching OK event to each remaining *incomplete* operation and remove all remaining crash events in $\sigma'$.
- AbortComplete: We define $\sigma'$ returned by AbortComplete by construction: At first we add all events from $\sigma$ to $\sigma'$. Then, we remove all events of aborted operations in $\sigma'$ that did not take effect. Next, we replace all remaining ABORT events in $\sigma$ with matching OK events.

[2] Note, that the last operation of each client in $\sigma$ might be incomplete even if the client did not crash.

| | |
|---|---|
| **Variables used by Algorithm 2 and 3:** | |
| *sig* signature | `/* signature /*` |
| *abort* boolean | `/* flags if operation is aborted /*` |
| $value_{suc}$ value | `/* written value of last successful write /*` |
| *retval* value | `/* return value of the read operation /*` |
| **Variables used by Algorithm 2:** | |
| *op_cnt* integer | `/* operation counter /*` |
| $op, x\_op, lso$ operation with fields $id = (client\_id, op\_cnt, type, reg)$, *value*, *tsv*, *sig* | `/* operation structure /*` |
| $tsv_{comp}[1..n]$ vector of integers | `/* ts vector of last completed operation /*` |
| $ts_{suc}$ integer | `/* timestamp of last successful operation /*` |
| **Variables used by Algorithm 3:** | |
| $op\_cnt[1..n]$ array of integer | `/* operation counter /*` |
| $op, x\_op, lso$ operation with fields $id = (client\_id, op\_cnt, type, reg)$, *value*, *tsm*, *sig* | `/* operation structure /*` |
| $tsm^{1..n}_{comp}[1..n]$ timestamp matrix of integers | `/* ts matrix of last completed operation /*` |
| $ts_{suc}[1..n]$ vector of integers | `/* timestamps of last successful operations /*` |

**Fig. 1.** Variables for Algorithms 2 and 3

Transformation CrashComplete removes incomplete operations that did not take effect from $\sigma$. This is reasonable as such events do not influence the execution. Instead of removing them, such events could also be moved to the end of sequence $\sigma$. The same argument applies to aborted operations that do not take effect which are removed by transformation AbortComplete. By first applying transformation CrashComplete and then transformation AbortComplete to sequence $\sigma$, we have transformed $\sigma$ into a sequence of events containing only *successful* operations. On the transformed sequence we give two equivalent definitions of *fork-linearizability* taken from recent work of Cachin *et al.* [2].

**Definition 3 (Fork-Linearizability)** A sequence of events $\sigma$ observed by the clients is called *fork-linearizable* with respect to a functionality $F$ if and only if for each client $C_i$, there exists a subsequence $\sigma_i$ of $\sigma$ consisting only of completed operations and a sequential permutation $\pi_i$ of $\sigma_i$ such that:

1. All completed operations in $\sigma$ occurring[3] at client $C_i$ are contained in $\sigma_i$; and
2. $\pi_i$ preserves the real-time order of $\sigma_i$; and
3. the operations of $\pi_i$ satisfy the sequential specification of $F$; and
4. for every $op \in \pi_i \cap \pi_j$, the sequence of events that precede $op$ in $\pi_i$ is the same as the sequence of events that precede $op$ in $\pi_j$.

[3] All successful operations of client $C_i$ occur at client $C_i$; together with condition 3. this further includes all operations on which an operation of client $C_i$ causally depends, i.e. operations that have written a value client $C_i$ reads.

**Definition 4 (Global Fork-Linearizability)** A sequence of events $\sigma$ observed by the clients is called *fork linearizable* with respect to a functionality $F$ if and only if there exists a sequential permutation $\pi$ of $\sigma$ such that:

1. $\pi$ preserves the real-time order of $\sigma$; and
2. for each client $C_i$ there exists a subsequence $\pi_i$ of $\pi$ such that:
   (a) events in $\pi$ occurring at client $C_i$ are contained in $\pi_i$; and
   (b) the operations of $\pi_i$ satisfy the sequential specification of $F$; and
   (c) for every $op \in \pi_i \cap \pi_j$, the sequence of events that precede $op$ in $\pi_i$ is the same as the sequence of events that precede $op$ in $\pi_j$.

Using two distinct but equivalent definitions of fork-linearizability simplifies the correctness proof of protocol LINEAR (by using Definition 3) and of protocol CONCUR (by using Definition 4). The notion of fork-linearizability and global fork-linearizability has shown to be equivalent [2].

## 4 The Protocols

In this section we present two lock-free protocols LINEAR and CONCUR that emulate a fork-linearizable shared memory on a Byzantine server. The LINEAR protocol is based on *vectors* of timestamps (described later in section 4.2) resulting in a communication complexity of $\mathcal{O}(n)$. The LINEAR protocol serializes all operations, and therefore it aborts concurrent operations even if they are applied to distinct registers. The CONCUR protocol (introduced later in section 4.3) allows for concurrent operations if they are applied to distinct registers and only operations on the same register are serialized. To achieve this, timestamp *matrices* are used leading to a communication complexity of $\mathcal{O}(n^2)$.

### 4.1 Protocol Properties

As mentioned above, LINEAR and CONCUR introduced emulate the shared functionality of a read/write register among a collection of clients and a (possibly) Byzantine server $S$. The LINEAR (CONCUR) protocol consists of two algorithms, run by the clients and the server respectively. If the server is faulty, it may refuse to respond to client requests or return (detectably) corrupted data such that liveness of the emulated functionality is violated. A malicious server may also mount a forking attack and partition clients. However, if the server behaves correctly, we require that the emulation does not block and clients are not forked.

To formalize the desired properties of the LINEAR and CONCUR protocol, we redefine the notion of *sequential* and *concurrent* operations under step contention [22] when the server is correct. We say that two operations $op$ and $op'$ are *sequential under step contention* if $op'$ does not perform steps at the server $S$ after $op$ performed its first step and before $op$ performed its last step at server $S$. Otherwise, $op$ and $op'$ are *concurrent under step contention*. The LINEAR and CONCUR protocol satisfy *Fork-consistency* and two liveness properties *Nontriviality* and *Termination*:

**Algorithm 1:** Read / Write Operation of Client $i$

| | |
|---|---|
| *Read*($j$) **do** | *Write*($v$) **do** |
|     *rw_operation*(READ, $\perp$, $j$) |     *rw_operation*(WRITE, $v$, $i$) |
|     **if** *abort* **then** return ABORT |     **if** *abort* **then** return ABORT |
|     return *retval* |     return OK |

**Fork-consistency**: Every execution of the LINEAR and CONCUR protocols satisfies fork-linearizability with respect to a shared read/write register emulated on a Byzantine server $S$. If $S$ is correct, then every execution is complete and has a linearizable history.

**Nontriviality**: When the server is correct, in an execution of the LINEAR (resp. CONCUR) protocol every operation that returns *abort* is concurrent under step contention with another operation (resp. with another operation on the same register).

**Termination**: When the server is correct and $\sigma$ is the sequence of events exhibited by an execution of the LINEAR or CONCUR protocol, then after applying transformation CRASHCOMPLETE to $\sigma$, every operation in $\sigma$ is complete.

### 4.2 The LINEAR Protocol

The LINEAR protocol is based on two main ideas. The first idea is that when two or more operations access the registers concurrently, all but one are aborted. In the protocol, operations need two rounds of communication with the server, and an operation *op* is aborted if a first round message of another operation arrives at the server between the points in time when the first round message and the second round message of *op* is received by the server. Hence, among the concurrent operations, the LINEAR protocol does not abort the "newest" operation. This scheme ensures that a pending operation of a crashed client does not interfere with other operations. Observe that using this strategy of aborting, successful operations execute in isolation and therefore accesses to the shared memory are serialized.

As a second idea, the LINEAR protocol assigns vector timestamps to operations such that a partial order $\leq$ on operations can be defined based on these timestamp vectors. The basic principle is that a client reads the most recent timestamp vector from the server during the first round, increments its own entry and writes the updated timestamp vector back to the server. Since successful operations run in isolation, the corresponding timestamp vectors are totally ordered, as no two successful operations read the same timestamp vector during the first round. Clearly, a Byzantine server may fork two clients, but then there are operations of these two clients *op* and *op'* with incomparable timestamp vectors. By the requirement of fork-linearizability, these two clients must not see any later updates of each other. For this purpose, the protocol ensures that the

**Algorithm 2:** LINEAR Protocol, Algorithm of Client $i$

$rw\_operation$(TYPE, $value$, $r$) **do**
  $abort \leftarrow false$
  $op\_cnt \leftarrow op\_cnt + 1$
  $op.id \leftarrow (i, op\_cnt, \text{TYPE}, r)$
  send $\langle \text{SUBMIT}, op.id \rangle$ to server
  **wait for** message $\langle \text{SUBMIT\_R}, x\_op, lso \rangle$
  **if not** $\mathtt{verify}(lso.sig) \wedge \mathtt{verify}(x\_op.sig)$ **then halt**
  **if not** $\forall k \neq i : tsv_{comp}[k] \leq lso.tsv[k] \wedge ts_{suc} = lso.tsv[i]$ **then halt**
  **if not** $x\_op.id.client\_id = r$ **then halt**
  **if not** $x\_op \leq lso \wedge lso.tsv[r] = x\_op.tsv[r]$ **then halt**
  $op.tsv \leftarrow lso.tsv$
  $op.tsv[i] \leftarrow op\_cnt$
  **if** TYPE = WRITE **then** $op.value \leftarrow value$
  $sig \leftarrow \mathtt{sign}(op.id || op.value || op.tsv)$
  $op.sig \leftarrow sig$
  send $\langle \text{COMMIT}, op \rangle$ to server
  **wait for** message $\langle \text{COMMIT\_R}, ret\_type \rangle$
  $tsv_{comp} \leftarrow op.tsv$
  **if** $ret\_type$ = ABORT **then**
    $op.value \leftarrow value_{suc}$
    $abort \leftarrow true$
  **else**
    $ts_{suc} \leftarrow op\_cnt$
    $value_{suc} \leftarrow op.value$
    **if** TYPE = READ **then** $retval \leftarrow x\_op.value$

**Algorithm 3:** CONCUR Protocol, Algorithm of Client $i$

$rw\_operation$(TYPE, $value$, $r$) **do**
  $abort \leftarrow false$
  $op\_cnt[r] \leftarrow op\_cnt[r] + 1$
  $op.id \leftarrow (i, op\_cnt[r], \text{TYPE}, r)$
  send $\langle \text{SUBMIT}, op.id \rangle$ to server
  **wait for** message $\langle \text{SUBMIT\_R}, x\_op, lso \rangle$
  **if not** $\mathtt{verify}(lso.sig) \wedge \mathtt{verify}(x\_op.sig)$ **then halt**
  **if not** $\forall k \neq i : tsm^r_{comp}[k] \leq lso.tsm^r[k] \wedge ts_{suc}[r] = lso.tsm^r[i]$ **then halt**
  **if not** $x\_op.id.client\_id = r$ **then halt**
  **if not** $x\_op \leq lso \wedge lso.tsm^r[r] = x\_op.tsm^r[r]$ **then halt**
  **forall** $k = 1..n, k \neq r$ **do**
    **if not** $tsm^k_{comp}$, $lso.tsm^k$ are comparable **then halt**
    $op.tsm^k \leftarrow \mathtt{max}\{tsm^k_{comp}, lso.tsm^k\}$
  $op.tsm^r \leftarrow lso.tsm^r$
  $op.tsm^r[i] \leftarrow op\_cnt[r]$
  **if** TYPE = WRITE **then** $op.value \leftarrow value$
  $sig \leftarrow \mathtt{sign}(op.id || op.value || op.tsm)$
  $op.sig \leftarrow sig$
  send $\langle \text{COMMIT}, op \rangle$ to server
  **wait for** message $\langle \text{COMMIT\_R}, ret\_type \rangle$
  $tsm_{comp} \leftarrow op.tsm$
  **if** $ret\_type$ = ABORT **then**
    $op.value \leftarrow value_{suc}$
    $abort \leftarrow true$
  **else**
    $ts_{suc}[r] \leftarrow op\_cnt[r]$
    $value_{suc} \leftarrow op.value$
    **if** TYPE = READ **then** $retval \leftarrow x\_op.value$

two clients remain forked by preventing any client from committing an operation $op''$ which is both greater than $op$ and $op'$.

**Description of the LINEAR Protocol** The shared memory emulated by the LINEAR protocol consists of $n$ SWMR registers $X[1], \ldots, X[n]$ such that client $C_i$ may write a value from set *Value* only to register $X[i]$ and may read from any register. The detailed pseudo-code of the LINEAR protocol appears in Algorithm 1, 2 and 4 and the variables used are described in Figure 1.

A client performs two rounds of communication with the server $S$ for both *Read* and *Write* operations (see Algorithm 1). This is implemented by calling procedure *rw_operation* (Algorithm 2) with type READ or WRITE respectively. When executing *rw_operation*, the client sends a SUBMIT message to the server $S$ announcing a read or write operation and waits for a matching response. The server $S$ responds with a SUBMIT_R message containing information on the current state of the server and the value to be read. In the second communication

**Algorithm 4:** LINEAR Protocol, Algorithm of Server $S$

**Variables:**
$Pnd$ set of operation ids `/* pend. ops */`
$Abrt$ set of operation ids `/* pending ops to be aborted */`

**upon** receiving message $\langle$SUBMIT, $id\rangle$ from client $i$ **do**
  $Abrt \leftarrow Pnd$
  $Pnd \leftarrow Pnd \cup \{id\}$
  send $\langle$SUBMIT_R, $X[id.reg], lso\rangle$ to client $i$
**upon** receiving message $\langle$COMMIT, $op\rangle$ from client $i$ **do**
  $Pnd \leftarrow Pnd \setminus \{op.id\}$
  **if** $op.id \in Abrt$ **then**
    send $\langle$COMMIT_R, ABORT$\rangle$ to client $i$
  **else**
    $X[i] \leftarrow op$
    $lso \leftarrow op$
    send $\langle$COMMIT_R, OK$\rangle$ to client $i$

**Algorithm 5:** CONCUR Protocol, Algorithm of Server $S$

**Variables:**
$Pnd[1..n]$ array of set of operation ids
$Abrt[1..n]$ array of set of operation ids
`/* pending ops to be aborted */`

**upon** receiving message $\langle$SUBMIT, $id\rangle$ from client $i$ **do**
  $Abrt[id.reg] \leftarrow Pnd[id.reg]$
  $Pnd[id.reg] \leftarrow Pnd[id.reg] \cup \{id\}$
  send $\langle$SUBMIT_R, $X[id.reg], lso[id.reg]\rangle$ to client $i$
**upon** receiving message $\langle$COMMIT, $op\rangle$ from client $i$ **do**
  $Pnd[op.id.reg] \leftarrow Pnd[op.id.reg] \setminus \{op.id\}$
  **if** $op.id \in Abrt[op.id.reg]$ **then**
    send $\langle$COMMIT_R, ABORT$\rangle$ to client $i$
  **else**
    $X[i] \leftarrow op$
    $lso[op.id.reg] \leftarrow op$
    send $\langle$COMMIT_R, OK$\rangle$ to client $i$

round, the client sends a COMMIT message to the server and waits for a COMMIT_R message to complete the operation. The COMMIT_R message is either of type OK or ABORT indicating to the client the outcome of the operation.

Each operation $op$ has a timestamp vector of size $n$ assigned to it during the protocol. The timestamp vector is part of the operation data structure and is denoted as $op.tsv$. The timestamp vector is used to define a partial order $\leq$ on operations. For two operations $op$ and $op'$ we say that $op \leq op'$ iff $op.tsv[i] \leq op'.tsv[i]$ for all $i = 1 \ldots n$. Operations of the LINEAR protocol have the data structure of a 4-tuple with entries $id$, $value$, $tsv$ and $sig$, where $sig$ is a signature on the operation by the client, $tsv$ is the timestamp vector, $value$ is the value to be written by the operation. Note that for simplicity of presentation, a *Read* operation rewrites the value of the client's last successful *Write*. The entry $id$ is a 4-tuple $(client\_id, op\_cnt, type, reg)$ itself, where $client\_id$ equals $i$ for $C_i$, $op\_cnt$ is a local timestamp of the client which is incremented during every operation, $type$ indicates whether the operation is a READ or a WRITE, and $reg$ determines the index of the register the client intends to read from. For *Write* operations of client $C_i$, $reg$ is always $i$. The server $S$ maintains the $n$ registers in a vector $X[1..n]$, where each $X[i]$ stores the last successful operation of $C_i$. Further, the server maintains an copy of the latest successful operation in variable $lso$.

When client $C_i$ invokes a new operation $op$ on register $X[r]$, it increments its local timestamp $op\_cnt$, sets the entries of $op.id$ to the operation type and register $r$, and sends $op.id$ in a SUBMIT message to the server (lines 2.2–2.5). The server labels the received operation $op$ as *pending*. If the server receives the SUBMIT message of another operation before the COMMIT message of $op$, then $op$ is aborted. The server then responds with a SUBMIT_R message containing the last successful operation *lso*, and the last successful operation *x_op* applied to register $X[r]$ (lines 4.4–4.6).

After receiving operations *lso* and *x_op* from the server, client $C_i$ performs a number of consistency checks (lines 2.7–2.10). If any of the checks fails, which implies that the server is misbehaving, then the client halts. In the first check, $C_i$ verifies the signatures of *lso* and *x_op*. The next check is needed to determine a consistent timestamp vector for operation *op*. The goal is to obtain a timestamp vector for *op* which is greater than both *lso*'s timestamp vector and that of $C_i$'s last *completed* operation. The timestamp vector of the latter is stored in $tsv_{comp}$ at $C_i$. The client checks that all but the $i$th entry in *lso.tsv* are greater or equal than the corresponding entries in $tsv_{comp}$. $C_i$'s entry $lso.tsv[i]$ must equal the timestamp of the last *successful* operation stored in $ts_{suc}$. Checks three and four are needed only by *Read*: $C_i$ checks that *x_op* is indeed the content of register $X[r]$. The last check verifies that *lso* is at least as large as *x_op* and that $lso.tsv[r]$ equals $x\_op.tsv[r]$.

If all checks are passed, $C_i$ increments its own entry $lso.tsv[i]$ and *lso.tsv* becomes the timestamp vector of *op*. Then, $C_i$ signs *op.id*, the write value and the timestamp vector *op.tsv*, and sends *op* in a COMMIT message to the server (lines 2.11–2.16). The server, removes *op.id* from the set of pending operations and checks if it has to be aborted. As mentioned earlier, if this is case, a SUBMIT message of another operation was received before the COMMIT of *op* and the server replies with ABORT (lines 4.8–4.10). Else, *op* is stored in $X[i]$ and also stored in *lso* as the last successful operation and the server replies with OK (lines 4.12–4.14).

When client $C_i$ receives the COMMIT_R message for operation *op*, *op* is completed and thus $tsv_{comp}$ is updated with *op.tsv*. If *op* is successful, then additionally $ts_{suc}$ becomes the $i$th entry of *op.tsv*. If *op* is a READ, then the value of *x_op* is returned (lines 2.18–2.25).

**Correctness Arguments** Instead of returning the most recent value written to register $X[j]$ by a write operation $op_w$, a Byzantine server may return an old value written by $op'_w$. Let $C_i$ be the client whose read operation $op_r$ reads the stale value written by $op'_w$. Observe that the Byzantine server returns a stale version of *lso* to $C_i$. Let us assume that all checks in Algorithm 2 are passed, thus $C_i$ is unaware of the malicious behavior of the server. Note, that the $j$th entry in the timestamp vector of $op'_w$ is smaller than the corresponding entry of $op_w$, as both are operations of client $C_j$ whose $j$th entry increases with every operation. As the check in line 2.10 is passed, the $j$th entry in $op_r$'s timestamp vector is also smaller than the one of $op_w$. As $C_i$ increments the $i$th entry in the timestamp vector during $op_r$ but not the $j$th entry, $op_r$ and $op_w$ are incomparable. We argue that in this situation, no client commits an operation which is greater than both $op_w$ and $op_r$. As no client other than $C_i$ increments the $i$th entry in a timestamp vector, all operation of other clients that "see" $op_w$ have a timestamp vector whose $i$th entry is smaller than $op_r.tsv[i]$ and whose $j$th entry is larger than $op_r.tsv[j]$. Thus, such operations are also incomparable with $op_r$ and do not join

$op_w$ and $op_r$. When client $C_i$ "sees" such an operation incomparable to $op_r$ as the latest successful operation *lso*, the check in line 2.8 is not passed because the $i$th entry of *lso* is smaller than the timestamp of $C_i$'s last successful operation. Hence, $C_i$ stops the execution. Analogously, the same arguments can be applied for client $C_j$ and operation $op_w$.

As all checks are passed when the server behaves correctly, it is not difficult to see that with a correct server, all operations invoked by correct clients complete. Also with a correct server, operations are only aborted in the specified situations. For a detailed correctness proof we refer to appendix A.1.

### 4.3 The Concur Protocol

The Concur protocol differs from the Linear protocol in the way how concurrent access to the server is handled. In contrast to the Linear protocol, in the Concur protocol concurrent operations that access different registers at the server are not aborted. However, the same aborting scheme as in the Linear protocol is used in the Concur protocol on a register basis in order to serialize all accesses to the same register. This means, that a correct server aborts operation *op* accessing register $i$ if and only if a SUBMIT message of another operation accessing register $i$ is received while *op* is pending.

To deal with concurrent operations, in the Concur protocol, instead of one timestamp vector, each operation is assigned $n$ timestamp vectors, each corresponding to one register. Such $n$ timestamp vectors form the *timestamp matrix* of an operation. The basic idea is that when a client accesses register $j$ then the client updates its own entry in the $j$th timestamp vector of the timestamp matrix. It is important to note that even with a correct server, the Concur protocol allows that two clients with concurrent operations may read the same timestamp matrix from the server and update different timestamp vectors such that the corresponding operations become incomparable. However, the Concur protocol ensures that (1) operations of the same client are totally ordered by $\leq$ and (2) operations accessing the same register at the server are totally ordered by $\leq$. This is sufficient to show that for any operation *op*, all operations *op* causally depends on, are ordered before *op* by $\leq$. Further, the Concur protocol ensures that two forked operations — i.e. for some $i$, the $i$th timestamp vectors in the timestamp matrices of the two operations are incomparable — will never be rejoined by another operation.

**Description of the Concur Protocol** The Concur protocol has the same message pattern as the Linear protocol and provides the same interface to the clients (Algorithm 1). The Concur protocol uses a different implementation of procedure *rw_operation* as described in Algorithm 3, Figure 1, and Algorithm 5. As the Concur protocol follows the structure of the implementation of the Linear protocol, in the following we highlight only the differences between the two protocols. The *operation* data structure differs from the Linear protocol only to the fact that the timestamp vector *tsv* is replaced by a timestamp matrix *tsm* (Figure 1).

When client $C_i$ invokes a new operation *op* on register $r$, it generates a new operation id which it sends to the server in a SUBMIT message (lines 3.2–3.5). One difference is that $C_i$ maintains a *separate* operation counter for each register $op\_cnt[1..n]$. The server replies with operations *lso* and $x\_op$ contained in a SUBMIT_R message. Here, $x\_op$ is the last successful operation stored in register $r$, and *lso* is the last successful operation that accessed register $r$. Note, that *lso* may not be stored in register $r$. The server maintains information on pending operations for each register separately (lines 5.4–5.6).

The first and the third check are identical to the LINEAR protocol. The second check on operations *lso* and $x\_op$ performed by the client corresponds to the second check in the LINEAR protocol. As CONCUR operations hold a timestamp matrix, the check is performed on the $r$th timestamp vectors of the timestamp matrices of *lso* and $x\_op$. The goal is to obtain a timestamp matrix that makes *op* greater than the last completed operation of $C_i$ and the last successful operation accessing register $r$, stored in *lso*. Like in the LINEAR protocol, the last check ensures that *lso* is greater than $x\_op$ and, unlike LINEAR, that the $r$th entries in the $r$th timestamp vector of the timestamp matrices of *lso* and $x\_op$ are equal. This particular entry is the one which has been updated during $x\_op$ (lines 3.7–3.10).

To determine the timestamp matrix for *op*, client $C_i$ selects the $r$th timestamp vector from *lso* as $r$th timestamp vector of *op* and for all other indices it takes the maximum timestamp vector from *lso* and $C_i$'s last completed operation. Finally, client $C_i$ increments its own entry in the $r$th timestamp vector using $op\_cnt[r]$ (lines 3.12–3.15). The remainder of the protocol is analogous to the LINEAR protocol.

**Correctness Arguments** First, we show that all completed operations of client $C_i$ are totally ordered by $\leq$. This is reasonable as $C_i$ cannot know if an aborted operation was actually aborted by the malicious server. To achieve this, as the timestamp matrix of a new operation *op* of $C_i$ depends on operation *lso* received in the SUBMIT_R message, the check in line 3.8 is needed: It guarantees together with lines 3.14–3.15 that the $r$th timestamp vector of *lso* is greater than the one of $C_i$'s last completed operation stored in $tsm_{comp}$. For the remaining timestamp vectors it holds by line 3.12–3.13, as in each case the maximal timestamp vector among *lso* and $tsm_{comp}$ is picked, that they are greater than the respective one of $C_i$'s last completed operation. Hence, operation *op* is greater than the last completed operation of $C_i$.

Second, we show that when $C_i$ reads value $op_w.value$ from register $j$ during *op* then *op* is greater than the corresponding operation $op_w$ under $\leq$. Analogously, by the check in line 3.12 and lines 3.13–3.15, it also holds that *op* is greater than operation *lso*. As the check in line 3.10 ensures that $op_w$ is smaller or equal than *lso*, by transitivity, *op* is greater than $op_w$.

These two proof sketches give an intuition how the CONCUR protocol ensures that all operations, *op* causally depends on, are ordered by $\leq$ before *op*. For a detailed proof of the safety and liveness properties of the CONCUR protocol, we refer to appendix A.2.

### 4.4 Complexity

In the LINEAR and CONCUR protocol all operations need two communication rounds to complete. We argue why two rounds are necessary for *Write* operations: The reasoning is based on the fact that the information possibly written by some *one*-round *Write* is independent from some operations of other clients. Consider the following sequential run with a correct server and clients $C_1$ and $C_2$: $Write_1(1,x)$, $Read_2(1) \to x$, $Write_1(1,y)$, $Read_2(1) \to y$. Note, that by the one-round assumption, $Write_1(1,y)$ does not depend on the preceding $Read_2(1) \to x$. Thus, a Byzantine server may "swap" the order of these two operations unnoticeably. Hence, we can construct a run with a Byzantine server, which is indistinguishable for $C_2$: $Write_1(1,x)$, $Write_1(1,y)$, $Read_2(1) \to x$, $Read_2(1) \to y$. As $C_2$'s second *Read* returns $y$, the run violates the sequential specification and thereby also fork-linearizability. Thus, two rounds are needed for *Write* operations and the *Write* operations emulated by the LINEAR and CONCUR protocol are optimal in this sense. We conjecture, that *Read* operations can be optimized in the the LINEAR and CONCUR protocol to complete after a single round. This would also imply that *Read* operations can be made *wait-free*.

The messages exchanged during the LINEAR protocol have size $\mathcal{O}(2(n + \iota + |v| + \varsigma))$, where $\iota$ is the length of an operation id, $|v|$ denotes the maximal length of a value from *Value* and $\varsigma$ is the length of a signature. The message complexity of the CONCUR protocol is in $\mathcal{O}(2(n^2 + \iota + |v| + \varsigma))$.

## 5 Conclusion

We have presented lock-free emulations of fork-linearizable shared memory on a Byzantine server, LINEAR and CONCUR. The LINEAR protocol is based on timestamp vectors and it has a communication complexity of $\mathcal{O}(n)$. It is the first lock-free protocol that emulates fork-linearizable storage at all. The impossibility result by Cachin *et al.* [2] is circumvented by aborting concurrent operations. The CONCUR protocol improves on the LINEAR protocol in the way how concurrent operations are handled. In the CONCUR protocol only concurrent operations accessing the same register need to be aborted. To achieve this, the CONCUR protocol relies on timestamp matrices and has a communication complexity of $\mathcal{O}(n^2)$.

## References

1. Mazières, D., Shasha, D.: Building Secure File Systems out of Byzantine Storage. In: PODC, New York, NY, USA, ACM (2002) 108–117
2. Cachin, C., Shelat, A., Shraer, A.: Efficient Fork-Linearizable Access to Untrusted Shared Memory. In: PODC, New York, NY, USA, ACM (2007) 129–138

3. Cachin, C., Keidar, I., Shraer, A.: Trusting the Cloud. ACM SIGACT News, Distributed Computing in the Clouds **40**(2) (June 2009) 81–86
4. CVS: Concurrent Versions System. Website Available online at `http://www.nongnu.org/cvs/`; visited June 2009.
5. SVN: Subversion. Website Available online at `http://subversion.tigris.org/`; visited June 2009.
6. Whitehead, Jr., E.J.: World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction. StandardView **5**(1) (1997) 3–8
7. Yang, J., Wang, H., Gu, N., Liu, Y., Wang, C., Zhang, Q.: Lock-free Consistency Control for Web 2.0 Applications. In: WWW, New York, NY, USA, ACM (2008) 725–734
8. Google Inc.: Google docs. Website Available online at `http://docs.google.com`; visited June 2009.
9. Wikipedia: List of file systems, distributed file systems. Website Available online at `http://en.wikipedia.org/wiki/List_of_file_systems`; visited June 2009.
10. Herlihy, M.: Wait-Free Synchronization. ACM Trans. Program. Lang. Syst. **13**(1) (1991) 124–149
11. Herlihy, M.P., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
12. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P.: The Complexity of Obstruction-Free Implementations. J. ACM **56**(4) (2009) 1–33
13. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-Free Synchronization: Double-Ended Queues as an Example. In: ICDCS, Washington, DC, USA, IEEE Computer Society (2003) 522
14. Oprea, A., Reiter, M.K.: On Consistency of Encrypted Files. In: DISC. (2006) 254–268
15. Cachin, C., Keidar, I., Shraer, A.: Fork Sequential Consistency is Blocking. Inf. Process. Lett. **109**(7) (2009) 360–364
16. Aguilera, M.K., Frolund, S., Hadzilacos, V., Horn, S.L., Toueg, S.: Abortable and Query-Abortable Objects and Their Efficient Implementation. In: PODC: Principles of distributed computing, New York, NY, USA, ACM (2007) 23–32
17. Cachin, C., Keidar, I., Shraer, A.: Fail-Aware Untrusted Storage. In: DSN. (2009)
18. Li, J., Mazières, D.: Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In: NSDI. (2007)
19. Aguilera, M.K., Toueg, S.: Timeliness-Based Wait-Freedom: A Gracefully Degrading Progress Condition. In: PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (2008) 305–314
20. Jayanti, P., Chandra, T.D., Toueg, S.: Fault-tolerant Wait-free Shared Objects. J. ACM **45**(3) (1998) 451–500
21. Pease, M., Shostak, R., Lamport, L.: Reaching Agreement in the Presence of Faults. J. ACM **27**(2) (1980) 228–234
22. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with Reads and Writes in the Absence of Step Contention. In: DISC. (2005) 122–136

## A Correctness of the LINEAR and CONCUR Protocol

### A.1 LINEAR Algorithm Proof

We first define the type *operation*, as used in the LINEAR protocol in Algorithm 2 and 4.

**Definition 5 (LINEAR Operation)** An *operation identifier (operation id)* is a 4-tuple $(client\_id, op\_cnt, type, reg)$, where $client\_id$, $op\_cnt$, and $reg$ are integers and where $type$ is element of the set $\{\text{READ}, \text{WRITE}\}$.

An *operation* is a 4-tuple $(id, value, tsv, sig)$, where $id$ is an operation id, $value$ is from set $Value$, $tsv$ is a vector of size $n$ of integers, and $sig$ is a signature.

We define a partial order $\leq$ on timestamp vectors and on operations. Note, that we regard only such operations $op$ after the corresponding timestamp vector entry $op.tsv$ has been assigned in line 2.12.

**Definition 6 (Order Relation)** For two timestamp vectors $tsv$ and $tsv'$ holds $tsv \leq tsv'$ if and only if

$$\forall i : tsv[i] \leq tsv'[i].$$

It holds $tsv = tsv'$ if and only if $tsv$ and $tsv'$ are the same timestamp vectors.
For two operations $op$ and $op'$ holds $op \leq op'$ if and only if

$$op.tsv \leq op'.tsv.$$

It holds $op = op'$ if and only if $op$ and $op'$ are the same operations.

It is easy to see that $\leq$ relation on operations (timestamp vectors) is *transitive*. As relation $\leq$ is a partial order on operations (timestamp vectors), we define a notion of when two operation (timestamp vectors) cannot be ordered by $\leq$.

**Definition 7 (Comparable)** For two timestamp vectors $tsv$ and $tsv'$ holds $tsv$ and $tsv'$ are *comparable* if and only if

$$tsv \leq tsv' \vee tsv' \leq tsv.$$

Otherwise, they are *incomparable*.
For two operations $op$ and $op'$ holds $op$ and $op'$ are *comparable* if and only if

$$op.tsv \text{ and } op'.tsv \text{ are comparable.}$$

Otherwise, they are *incomparable*. We also call two incomparable operations *forked*.

The next Lemma shows that $\leq$ relation on LINEAR operations does not violate the real-time order of operations.

**Lemma 8** If $op \leq op'$ then $op'$ does not precede $op$.

*Proof.* Let $op$ and $op'$ be two operations of client $C_i$ and $C_j$ and let us assume by contradiction that $op'$ precedes $op$ and $op \leq op'$. During $op$, client $C_i$ updates the $i$th entry in the timestamp vector (line 2.12). As $op'$ precedes $op$ and as the server cannot forge signatures (line 2.14), at the point in time when $C_j$ received the SUBMIT_R message during $op'$, there exists no operation $op''$ such that $op''.tsv[i] \geq op.tsv[i]$. Thus, we have that $op.tsv[i] > op'.tsv[i]$. However, this contradicts the assumption that $op \leq op'$.

The following two Lemmas show that operations which causally influence each other are ordered by $\leq$ such that the causal order is respected. The operations of one client causally influence each other (Lemma 9) as well as a write operation and an operation which reads the written value (Lemma 10).

**Lemma 9** All operations of the same client are totally ordered by $\leq$ relation on operations.

*Proof.* We show that operation $op$ of client $C_i$ is greater than its previous completed operation $op_{comp}$. Note, that by line 2.18 $op_{comp}.tsv = tsv_{comp}$. Let $l$ be operation $lso$ as received in the SUBMIT_R message by $C_i$ during operation $op$ (line 2.6). To pass the check in line 2.8, $l$ must be greater or equal in all entries $\neq i$ of the timestamp vector than $op_{comp}$. By line 2.11 we have that $op.tsv[k] \geq op_{comp}.tsv[k]$ for all $k \neq i$. In line 2.12 the $i$th entry of the timestamp vector is updated by a larger entry, as $op\_cnt$ is incremented with every invoked operation of $C_i$ (line 2.3), and we get that $op.tsv > op_{comp}.tsv$, implying that $op > op_{comp}$. By induction on $C_i$'s operations, it follows that $op$ is greater than any operation of $C_i$ that precedes $op$.

**Lemma 10** If $op_r$ is a read operation of client $C_i$ that returns $op_w.value$ from register $j$, then $op_w < op_r$.

*Proof.* To pass the check in line 2.10, it must be that $op_w \leq lso$ and by lines 2.11 and 2.12 it holds that $lso < op_r$. Thus, if $op_r$ returns $op_w.value$ it must be that $op_w < op_r$.

Lemma 11 shows that the sequential specification of the emulated registers is not violated by the LINEAR protocol. As long as all operations are ordered by $\leq$, the LINEAR protocol ensures that a read operation returns the latest written value from a register.

**Lemma 11** Let $op_r$ be a read operation of client $C_i$ that returns $op_w.value$ from register $j$. If operations are totally ordered by $\leq$ than there is no successful write operation $op'_w$ between $op_w$ and $op_r$ that writes $v' \neq op_w.value$ to register $j$.

*Proof.* Note, that $op'_w$ and $op_w$ are both operations of client $C_j$ and they write different values. Thus, let us assume for contradiction that such $op'_w$ exists and we have $op_w < op'_w < op_r$. Let $l$ be operation $lso$ as seen by $op_r$ (line 2.6). If $i \neq j$ the $j$th entry of $l$ is not changed during $op_r$ and thus $op_r.tsv[j] = l.tsv[j]$. To pass the check in line 2.10, we also have $op_w.tsv[j] = l.tsv[j]$. Moreover, since $op_w$

and $op'_w$ are both operations of the same client and $op_w$ precedes $op'_w$ we have by line 2.3 $op'_w.tsv[j] > op_w.tsv[j]$ and thus $op_r.tsv[j] = op_w.tsv[j] < op'_w.tsv[j]$. This contradicts the assumption that $op'_w < op_r$.

If $i = j$ then client $C_i$ reads from its own register $i$. As $op'_w$ precedes $op_r$ we have that $ts_{suc} \geq op'_w.tsv[i]$ (line 2.23). To pass the check in line 2.8 we have that $ts_{suc} = op_w.tsv[i]$. This implies that $op_w.tsv[i] \geq op'_w.tsv[i]$ and contradicts the assumption that $op_w < op'_w$.

The next Lemma shows that two forked clients will never be rejoined. This ensures, that in the "local" history as observed by any client, operations are totally ordered by $\leq$.

**Lemma 12** Suppose $op_x$ and $op_y$ are two incomparable operations. Then there exists no operation which is greater than both $op_x$ and $op_y$.

*Proof.* We assume for contradiction that there exists an operation $op_{join}$ of client $C_z$ such that $op_{join} > op_x$ and $op_{join} > op_y$. Thus, there must be a minimal operation $op_z \leq op_{join}$ of client $C_z$ for which $op_x \leq op_z$ and $op_y \leq op_z$ holds. Note that $op_z \neq op_x$ and $op_z \neq op_y$ because otherwise $op_x$ and $op_y$ would be comparable. Let $l$ be *lso*, sent to client $C_z$ during operation $op_z$ (line 2.6). By lines 2.11 and 2.12 it holds that $l < op_z$. By the assumption that $op_z$ is minimal, we know that $l$ cannot be both greater than $op_x$ and $op_y$. Therefore, we assume w.l.o.g. that $l \not\geq op_x$. This means that there exists some index $k$ such that $op_x.tsv[k] > l.tsv[k]$. We distinguish the following two cases:

1. $k \neq z$. This implies, as by line 2.12 $op_z$ updates only its own entry in the timestamp vector, that $op_z.tsv[k] = l.tsv[k] < op_x.tsv[k]$ which contradicts the assumption that $op_x < op_z$.
2. $k = z$. This means that client of $op_z$ has updated the $k$th entry in its timestamp vector to some value $\geq op_x[k]$ during some operation $op'_z$. Thus, we have $op'_z.tsv[k] \geq op_x.tsv[k]$, implying that $op'_z.tsv[k] > l.tsv[k]$. However, to pass the check in line 2.8, it must be that $op'_z.tsv[k] = l.tsv[k]$ which is a contradiction.

Thus, the assumption $l \not\geq op_x$ is wrong and $l \geq op_x$ holds. The analogous arguments can be used to show that $l \geq op_y$. However, this contradicts the assumption that $op_z$ is minimal and we are done.

The next Lemma proves the main result that the LINEAR protocol satisfies fork-linearizability according to Definition 3.

**Lemma 13** The LINEAR protocol described in Algorithm 1, 2 and 4 emulates $n$ SWMR registers on a Byzantine server satisfying the Fork-Consistency property (section 4.1).

*Proof.* Let $\sigma$ be the sequence of events observed by the clients in the protocol. At first, apply transformation CRASHCOMPLETE and ABORTCOMPLETE in this order to $\sigma$ (Definition 2). We construct the sets $\sigma_i$ (for $i = 1, \ldots, n$) as required by the definition of fork-linearizability. We include in $\sigma_i$ the last operation of client $C_i$ in $\sigma$, $op_i$. Then, we include into $\sigma_i$ all completed operations $op'$ in $\sigma$ such that $op_i \geq op'$. We now create the sequences $\pi_i$ from $\sigma_i$ by sorting $\sigma_i$ according to $\leq$ relation on operations. Since every operation inserted into $\pi_i$ is less than or equal to $op_i$, according to Lemma 12 all operations in $\pi_i$ are totally ordered by $\leq$. We now show that all requirements of fork-linearizability are satisfied (Definition 3):

Requirement 1 of fork-linearizability is preserved as the last completed operation of $C_i$ was inserted into $\pi_i$, by Lemma 9 which shows that all operation of $C_i$ are totally ordered by $\leq$, by Lemma 10, and by transitivity of $\leq$ on operations.

As by Lemma 12 all operations in $\pi_i$ are totally ordered, Lemma 8 guarantees requirement 2 of fork-linearizability.

For requirement 3 of fork-linearizability, we need to show that $\pi_i$ satisfies the sequential specification for read/write registers. As all operations in $\pi_i$ are totally ordered, sequential specification holds by Lemma 11.

To show requirement 4, we suppose that $op$ was included in sequences $\pi_i$ and $\pi_j$ . Let $op_i$ be the last operation of $C_i$ in $\pi_i$. By construction of $\pi_i$, $op_i \geq op$. All operations $op'$ s.t. $op' \leq op$ were also included into $\pi_i$ due to the transitivity of $\leq$ on operations. For the same reason, the same group of operations were included into $\pi_j$ as well. Thus, requirement 4 of fork-linearizability holds.

**Lemma 14** If the server is correct then no operation in Algorithm 2 blocks.

*Proof.* We have to show that no operation blocks in lines 2.7 − 2.10.

- $\texttt{verify}(lso.sig) \wedge \texttt{verify}(x\_op.sig)$ is TRUE: As clients are non-malicious, all signatures are correct. Thus, the protocol does not block in line 2.7.
- Assume by contradiction that $\exists k : tsv_{comp}[k] > lso.tsv[k]$: Let $op_{comp}$ be the last completed operation that updated $tsv_{comp}$ in line 2.18. As the server is correct it returns only successful operations, so particularly operation *lso* is successful. By line 4.12 and 4.13 whenever some operation successfully completes, *lso* is also updated. Thus, as *lso* is monotonically increasing (Lemma 12 and Lemma 8), $\forall k : op_{comp}.tsv[k] \not> lso.tsv[k]$, which is a contradiction.
  Assume by contradiction that $ts_{suc} \neq lso.tsv[i]$: Let $op_{suc}$ be the last successful operation that updated $ts_{suc}$. By reasoning above, whenever some register at the server is updated, *lso* is also updated. This implies that $lso.tsv[i] \not< op_{suc}.tsv[i]$. Therefore, it must be that $lso.tsv[i] > op_{suc}.tsv[i]$. Entry $lso.tsv[i]$ is only updated when an operation of $C_i$ later than $op_{suc}$ successfully completes. However, in this case $ts_{suc}$ is also updated to the same value and we have $lso.tsv[i] = ts_{suc}$, a contradiction.
  Thus, the protocol does not block in line 2.8.
- $x\_op.id.client = r$ is TRUE: As the server is correct it returns $x\_op$ from the correct register. Thus the protocol does not block in line 2.9.
- Assume by contradiction that $x\_op \not\leq lso$: As the server is correct it returns only successful operations. Thus, both $x\_op$ and *lso* are successful. This implies that $x\_op$ and *lso* can be ordered by $\leq$. By line 4.12 and 4.13 whenever some register at the server is updated, *lso* is also updated. Thus, as *lso* is

monotonically increasing, $x\_op \not> lso$. Therefore, $x\_op \leq lso$ which is a contradiction.

Assume by contradiction that $lso.tsv[r] \neq x\_op.tsv[r]$: By the item above, whenever some register at the server is updated, $lso$ is also updated. This implies that $lso.tsv[r] \not< x\_op.tsv[r]$. Therefore, it must be that $lso.tsv[r] > x\_op.tsv[r]$. Entry $lso.tsv[r]$ is only updated when an operation of $C_r$ later than $x\_op$ successfully completes. However, in this case $x\_op.tsv[r]$ is also updated to the same value and we have $lso.tsv[r] = x\_op.tsv[r]$, a contradiction.

Thus, the protocol does not block in line 2.10

Hence, no operation blocks in lines 2.7 – 2.10.

**Lemma 15** *With a correct server, an operation op of a client aborts only if the server receives a* SUBMIT *message from another client after the* SUBMIT *and before the* COMMIT *message corresponding to op.*

*Proof.* If *op* aborts then the server has received the COMMIT message corresponding to *op*. As the server is correct and no SUBMIT message of another operation is received after the SUBMIT and before the COMMIT message of *op*, $op.id \notin Abrt$. By line 4.12 and 4.13, *op* is not aborted.

Finally, the following theorem proofs the correctness of the LINEAR protocol.

**Theorem 16** *The* LINEAR *protocol emulates $n$ SWMR registers on a Byzantine server satifying the properties Fork-Consistency, Termination and Nontriviality (section 4.1).*

*Proof.* By Lemma 13, safety of fork-linearizability is satisfied, by Lemma 14, the protocol does not block, and by Lemma 15, no operation running in isolation is aborted when the server is correct.

### A.2 CONCUR Algorithm Proof

We first define the type *operation*, as used in the CONCUR protocol in Algorithm 3 and 5.

**Definition 17 (CONCUR Operation)** An *operation* is a 4-tuple (*id*, *value*, *tsv*, *sig*), where *id* is an operation id according to Definition 5, *value* is from set *Value*, *tsm* is matrix consisting of $n$ timestamp vectors $tsm^1, \ldots, tsm^n$ where each timestamp vector is a vector of size $n$ of integers, and *sig* is a signature.

As in section A.1 we define a partial order $\leq$ on operations. Note, that we regard only such operations *op* after the corresponding timestamp matrix entry *op.tsm* has been assigned in line 3.15.

**Definition 18 (Order Relation)** For two operations $op$ and $op'$ holds $op \leq op'$ if and only if

$$\forall i : op.tsm^i \leq op'.tsm^i.$$

Relation $\leq$ on timestamp vectors is defined in Definition 6. It holds $op = op'$ if and only if $op$ and $op'$ are the same operations.

It is easy to see that $\leq$ relation on operations is transitive. As relation $\leq$ is a partial order on operations, we define a notion of when two operations cannot be ordered by $\leq$.

**Definition 19 (Comparable)** For two operations $op$ and $op'$ holds $op$ and $op'$ are *comparable* if and only if

$$op \leq op' \vee op' \leq op.$$

Otherwise, they are *incomparable*.

In contrast to the definitions for the LINEAR protocol in section A.1, in the CONCUR protocol clients with incomparable operations are not necessarily forked. Thus, the notion of *forking* is given in the next definition.

**Definition 20 (Forked)** For two operations $op$ and $op'$ holds $op$ and $op'$ are *forked* if and only if

$$\exists i : op.tsm^i \text{ and } op'.tsm^i \text{ are } \textit{incomparable}.$$

The next Lemma shows that $\leq$ relation on CONCUR operations does not violate the real-time order of operations.

**Lemma 21** If $op \leq op'$ then $op'$ does not precede $op$.

*Proof.* Let $op$ and $op'$ be two operations of client $C_i$ and $C_j$ and let us assume by contradiction that $op'$ precedes $op$ and $op \leq op'$. During $op$, client $C_i$ updates the $i$th entry in the $k$th timestamp vector of the timestamp matrix (line 3.15). As $op'$ precedes $op$ and as the server cannot forge signatures (line 3.17), at the point in time when $C_j$ received the SUBMIT_R message during $op'$ (line 3.6), there exists no operation $op''$ such that $op''.tsv[i] \geq op.tsv[i]$. Thus, we have that $op.tsm^k[i] > op'.tsm^k[i]$. As $op$ and $op'$ are comparable, this implies that $op.tsm^k > op'.tsm^k$. However, this contradicts the assumption that $op \leq op'$.

Analogously to the proof in section A.1, the following two Lemmas show that operations which causally influence each other are ordered by $\leq$ such that the causal order is respected. The operations of one client causally influence each other (Lemma 22) as well as a write operation and an operation which reads the written value (Lemma 23).

**Lemma 22** All operations of the same client are totally ordered by $\leq$ relation on operations.

*Proof.* We show that operation $op$ of client $C_i$ is greater than its previous completed operation $op_{comp}$. Note, that by line 3.21 $op_{comp}.tsm = tsm_{comp}$. By line 3.13, as check in line 3.12 is passed, we have that $op.tsm^k \geq op_{comp}.tsm^k$ for all $k \neq r$. To pass the check in line 3.8, $lso.tsm^r$ is greater or equal than $tsm^r_{comp}$ in all entries but the $i$th entry. However, in lines 3.14 and 3.15 the $i$th entry of the $r$th vector of the timestamp matrix is updated by a larger entry and we get that $op.tsm^r > op_{comp}.tsm^r$. Thus, we have that $op > op_{comp}$. By induction on $C_i$'s operations, it follows that $op$ is greater than any operation of $C_i$ that precedes $op$.

**Lemma 23** If $op_r$ is a read operation of client $C_i$ that returns $op_w.value$ from register $j$, then $op_w < op_r$.

*Proof.* To pass the check in line 3.10, it must be that $op_w \leq lso$ and by lines 3.13 and 3.15 it holds that $lso < op_r$. Thus, if $op_r$ returns $op_w.value$ it must be that $op_w < op_r$.

The next definition constructs a sequential permutation of the sequence of events produced by the run of the Concur protocol . The construction helps to simplify the proof of the main correctness proof of the Concur protocol.

**Definition 24 (Sequential Permutation)** Let $\sigma$ be the sequence of events observed by the clients in the protocol. We define a *sequential permutation* $\pi$ of $\sigma$ by construction: At first we add all events from $\sigma$ to $\pi$. Then, we apply transformations CrashComplete and AbortComplete (Definition 2) in this order to $\pi$. Finally, we totally order $\pi$ by the following rules:

1. The operations are sorted by relation $\leq$ on operations.
2. Yet unsorted operations are sorted according to the real-time order of their completion events in $\sigma$.

A subsequence $\pi_i$ of $\pi$ contains all operations $op$ of client $C_i$, and all operations $op' \leq op$.

In contrast to the Linear protocol, during the Concur protocol even non-forked clients may produce *incomparable* operations. The next Lemma shows how the Concur protocol ensures the sequential specification of a read/write register and that forked operations will never be rejoined.

**Lemma 25** Let $op_r$ be a read operation of client $C_i$ that returns $op_w.value$ from register $j$ and $op_r$ is contained in some $\pi_k$ as defined in Definition 24. Then

1. $op_w$ is in $\pi_k$, and
2. there is no write operation $op'_w$ of $C_j$ between $op_w$ and $op_r$ in $\pi_k$ that writes $v' \neq op_w.value$ to register $j$.

*Proof.* The first statement follows directly from Lemma 23, which states that $op_w < op_r$, and the construction of $\pi_k$.

For the second statement, note that $op'_w$ and $op_w$ are both operations of client $C_j$ and they write different values. Thus, let us assume for contradiction that such operation $op'_w$ exists and we have $op_w$ precedes $op'_w$ and $op'_w$ precedes $op_r$. We first show that (A) $op_r$ and $op'_w$ are forked. Then we show (B) that $op'_w$ is not in $\pi_k$.

*Proof of A:*

We first rule out the trivial case when $i = j$: If $i = j$ then client $C_i$ reads from its own register $i$. As $op_w$ precedes $op'_w$ we have that at client $C_i$, $ts_{suc}[i] = op'_w.tsm^i[i] > op_w.tsm^i[i]$. During $op_r$ the check in line 3.8 is not passed as $ts_{suc}[i] \neq op_w.tsm^i[i]$ or the check in line 3.10 is not passed as $op'_w.tsm^i[i] \neq op_w.tsm^i[i]$. Hence, $op_r$ blocks which contradicts the precondition that $op_r$ is in $\pi_k$. Therefore, $i \neq j$.

Let $l$ be operation *lso* as seen by $op_r$. To pass the check in line 3.10, we have $op_w.tsm^j[j] = l.tsm^j[j]$. As $i \neq j$ client $C_i$ updates only its own entry (line 3.15), the $j$th entry of $l.tsm^j$ is not changed during $op_r$ and thus $op_r.tsm^j[j] = l.tsm^j[j]$ (line 3.14). Moreover, since $op_w$ and $op'_w$ are both operations of the same client and $op_w$ precedes $op'_w$ we have $op'_w.tsm^j[j] > op_w.tsm^j[j]$ (lines 3.3 and 3.15) and thus

$$op_r.tsm^j[j] = op_w.tsm^j[j] < op'_w.tsm^j[j].$$

Further, during $op_r$ the $i$th entry of the $j$th timestamp vector is updated to $op_r.tsm^j[i]$. As $op'_w$ precedes $op_r$ and as the server cannot forge signatures, at the point in time when $C_j$ received the SUBMIT_R message during $op'_w$, there exists no operation $op''$ such that $op''.tsm^j[i] \geq op_r.tsm^j[i]$. Thus,

$$op'_w.tsm^j[i] < op_r.tsm^j[i]$$

implying that $op_r.tsm^j$ and $op'_w.tsm^j$ are incomparable, and thus, $op_r$ and $op'_w$ are forked.

*Proof of B:*

To show that $op'_w$ is not included in $\pi_k$, we assume by contradiction that $op'_w$ is element of $\pi_k$. By construction of $\pi_k$ there exist minimal operations $op$, $op'$ of client $C_k$ such that $op_r \leq op$ and $op'_w \leq op'$. As any two operations of $C_k$ are ordered we assume w.l.o.g. that $op' \geq op$ and thus $op' \geq op_r$ and $op' \geq op'_w$. Note that, by definition of $\leq$ relation, it must hold for the timestamp vectors that $op'.tsm^j \geq op_r.tsm^j$ and $op'.tsm^j \geq op'_w.tsm^j$. By line 3.15, the $i$th entry of each timestamp vector in a timestamp matrix is only incremented by client $C_i$. Thus, to satisfy $op'.tsm^j[i] \geq op_r.tsm^j[i]$, there must be a sequence of operations accessing register $j$, starting with $op_r$ and ending with $op'$ such that the $j$th timestamp vectors are monotonically increasing.

As the $j$th entry of each timestamp vector in a timestamp matrix is only incremented by client $C_j$ and $op_r.tsm^j[j] < op'_w.tsm^j[j]$, no operation of client $C_j$ is in this sequence. Otherwise, for client $C_j$ the check ($ts_{suc}[j] = lso.tsm^j[j]$) in line 3.8 would not be passed, as the $j$th entry of the $j$th timestamp vector is smaller than the corresponding entry of $ts_{suc}[j]$ after $op'_w$ was completed. Thus, as no operation of $C_j$ is in this sequence, all operations in this sequence have $op_r.tsm^j[j]$ as their $j$th entry in the $j$th timestamp vector. Therefore,

$op'.tsm^j[j] = op_r.tsm^j[j] < op'_w.tsm^j[j]$ and as we have shown that $op'.tsm^j[i] \geq op_r.tsm^j[i] > op'_w.tsm^j[i]$, we have that timestamp vectors $op'.tsm^j$ and $op'_w.tsm^j$ are incomparable. This contradicts the fact that $op' \geq op'_w$ and thus, $op'_w$ is not contained in $\pi_k$.

The next Lemma proves the main result that the CONCUR protocol satisfies fork-linearizability according to Definition 4.

**Lemma 26** The CONCUR protocol described in Algorithm 1, 3 and 5 emulates $n$ SWMR registers on a Byzantine server satisfying the Fork-Consistency property (section 4.1).

*Proof.* We show that the sequential permutation $\pi$ of $\sigma$ and all $\pi_i$ defined by Definition 24 satisfy the properties of fork-linearizability as given in Definition 4.

We first show that $\pi$ maintains real-time order of $\sigma$, i.e. if $op$ precedes $op'$ in $\sigma$, then $op$ precedes $op'$ in $\pi$. By Lemma 21, operations sorted by $\leq$ respect real-time order of $\sigma$. By the definition of $\pi$ all other operations are also ordered in real-time order.

Requirement 2.(a) of Fork-Consistency is satisfied by Lemma 22, which shows that all operations of one client are totally ordered by $\leq$, Lemma 23, and transitivity of $\leq$ on operations. Requirement 2.(b) follows from Lemma 25. Requirement 2.(c) follows directly from the construction of $\pi_i$.

**Lemma 27** If the server is correct, then the no operation in Algorithm 3 blocks.

*Proof.* We have to show that no operation blocks in lines 3.7 – 3.10.

- $\texttt{verify}(lso.sig) \wedge \texttt{verify}(x\_op.sig)$ is TRUE: As clients are non-malicious, all signatures are correct. Thus the protocol does not block in line 3.7.
- Assume by contradiction that $\exists k : tsm^r_{comp}[k] > lso.tsm^r[k]$: Let $op_{comp}$ be the last completed operation that updated $tsm_{comp}$ in line 3.21. As the server is correct it returns only successfully completed operations, so, particularly operations $op_{comp}$ and $lso$ are successful. By line 5.12 and 5.13 whenever some operation to $r$ takes effect, $lso[r]$ is also updated. Thus, as $lso[r]$ is monotonically increasing, $op_{comp}.tsm^r \not> lso.tsm^r$. Therefore, $tsm^r_{comp} \not> lso.tsm^r$ which is a contradiction.
  Assume by contradiction that $ts_{suc}[r] \neq lso.tsm^r[i]$: Let $op_{suc}$ be the successful operation accessing register $r$ that updated $ts_{suc}[r]$. By the reasoning above, whenever register $r$ at the server is updated, $lso[r]$ is also updated. This implies that $lso.tsm^r[i] \not< op_{suc}.tsm^r[i]$. Therefore, it must be that $lso.tsm^r[i] > op_{suc}.tsm^r[i]$. Entry $lso.tsm^r[i]$ is only updated when an operation of $C_i$ later than $op_{suc}$ successfully completes. However, in this case $ts_{suc}[r]$ is also updated to the same value and we have $lso.tsm^r[i] = ts_{suc}[r]$, a contradiction.
  Thus, the protocol does not block in line 3.8
- $x\_op.id.client = r$ is TRUE: As the server is correct it returns $x\_op$ from the correct register. Thus the protocol does not block in line 3.9.

– Assume by contradiction that $x\_op \not\leq lso$: As the server is correct it returns only operations that successfully completed. Thus, both $x\_op$ and $lso$ are successful. As $x\_op$ and $lso$ both access register $r$, this implies that $x\_op$ and $lso$ can be ordered by $\leq$. By line 5.12 and 5.13 whenever some register $r$ at the server is updated, $lso[r]$ is also updated. Thus, as $lso[r]$ is monotonically increasing, $x\_op \not> lso$. Therefore, $x\_op \leq lso$ which is a contradiction.
Assume by contradiction that $lso.tsm^r[r] \neq x\_op.tsm^r[r]$: Whenever some register at the server is updated, $lso[r]$ is also updated. This implies that $lso.tsm^r[r] \not< x\_op.tsm^r[r]$. Thus, it must be that $lso.tsm^r[r] > x\_op.tsm^r[r]$. Entry $lso.tsm^r[r]$ is only updated when an operation of $C_r$ accessing register $r$ later than $x\_op$ is successful. However, in this case $x\_op.tsm^r[r]$ is also updated to the same value and we have $lso.tsm^r[r] = x\_op.tsm^r[r]$, a contradiction.
Thus, the protocol does not block in line 3.10

Hence, no operation blocks in lines 3.7 – 3.10.

**Lemma 28** With a correct server, an operation *op* of a client accessing register $r$ aborts only if the server receives a SUBMIT message from another client accessing register $r$ after the SUBMIT and before the COMMIT message corresponding to *op*.

*Proof.* If *op* aborts then the server has received the COMMIT message corresponding to *op*. As the server is correct and no SUBMIT message of another operation accessing $r$ is received after the SUBMIT and before the COMMIT message of *op*, $op.id \notin Abrt[r]$. By line 5.12 and 5.13, *op* is not aborted.

Finally, the following theorem proofs the correctness of the CONCUR protocol.

**Theorem 29** The CONCUR protocol emulates $n$ SWMR registers on a Byzantine server satisfying the properties Fork-Consistency, Termination and Nontriviality (section 4.1).

*Proof.* By Lemma 26, Safety of fork-linearizability is satisfies, by Lemma 27, the protocol does not block, and by Lemma 28, no operation accessing a register in isolation is aborted when the server is correct.