Fork-Consistent Constructions From Registers*

Matthias Majuntke,[1] Dan Dobre,[2] Christian Cachin,[3] and Neeraj Suri[1]

[1] {majuntke,suri}@cs.tu-darmstadt.de
Technische Universität Darmstadt, Hochschulstraße 10, 64289 Darmstadt, Germany
[2] dan.dobre@neclab.eu
NEC Laboratories Europe, Kurfürsten-Anlage 36, 69115 Heidelberg, Germany
[3] cca@zurich.ibm.com
IBM Research – Zurich, Säumerstrasse 4, 8803 Rüschlikon, Switzerland

Abstract. Users increasingly execute services online at remote providers, but they may have security concerns and not always trust the providers. Fork-consistent emulations offer one way to protect the clients of a remote service, which is usually correct but may suffer from Byzantine faults. They feature linearizability as long as the service behaves correctly, and gracefully degrade to fork-consistent semantics in case the service becomes faulty. This guarantees data integrity and service consistency to the clients.
All currently known fork-consistent emulations require the execution of non-trivial computation steps by the service. From a theoretical viewpoint, such a service constitutes a *read-modify-write* object, representing the strongest object in Herlihy's wait-free hierarchy [1]. A read-modify-write object is much more powerful than a shared memory made of so-called *registers*, which lie in the weakest class of all shared objects in this hierarchy. In practical terms, it is important to reduce the complexity and cost of a remote service implementation as computation resources are typically more expensive than storage resources.
In this paper, we address the fundamental structure of a fork-consistent emulation and ask the question: Can one provide a fork-consistent emulation in which the service does not execute computation steps, but can be realized only by a shared memory? Surprisingly, the answer is yes. Specifically, we provide two such algorithms that can be built only from registers: A fork-linearizable construction of a universal type, in which operations are allowed to abort under concurrency, and a weakly fork-linearizable emulation of a shared memory that ensures wait-freedom when the registers are correct.

Keywords: distributed system, shared memory, fork-consistency, universal object, atomic register, Byzantine faults

1 Introduction

The increasing trend of executing services online "in the cloud" [2] offers many economic advantages, but also raises the challenge of guaranteeing security and strong consistency to its users. As the service is provided by a remote entity that wants to retain its customers, the service usually acts as specified. But online services may fail for

* Research funded in part by DFG GRK 1362 (TUD GKmM).

various reasons, ranging from simply closing down (corresponding to a crash fault) to deliberate and sometimes malicious behavior (corresponding to a Byzantine fault).

For some kinds of services, cryptographic techniques can prevent a malicious provider from forging responses or snooping on customer data. But other violations are still possible in the asynchronous model considered here: for instance, when multiple isolated clients interact only through a remote provider, the latter may send diverging and inconsistent replies to the clients. In this context, "forking" consistency conditions [3,4] offer a gracefully degrading solution because they make it much easier for the clients to detect such violations. More precisely, they ensure that if a Byzantine provider even *once* sent a wrong response to some client, then this client becomes *forever isolated* or *forked* from those other clients to which the provider responded differently. With this notion, clients may easily detect service misbehavior from a single inconsistent operation, e.g. by out-of-band communication.

Forking consistency conditions are often encapsulated in the notion of a *Byzantine emulation* [4], which ensures graceful degradation of the service's semantics: If the service is correct, then operations execute atomically. In any other case, the clients still observe operations according to the forking consistency notion. Fork-consistency represents a safety property — after all, a faulty service may simply stop. The liveness property in a Byzantine emulation refers to the good case when the service behaves correctly.

Fork-linearizability [3,4] ensures that clients always observe linearizable [5] service behavior and that two clients, once forked, will never again see each other's updates to the system (i.e. they share the same history prefix up to the forking point). However, it has been found that fork-linearizable Byzantine emulations of a shared memory *cannot* always provide *wait-free* operations [4], i.e., some clients may be blocked because of other clients that execute operations concurrently. An escape is offered by the weaker liveness property of abortable emulations, which allow client operations to *abort* under contention [6]. As another alternative, the notion of *weak fork-linearizability* relaxes fork-linearizability in order to allow wait-free client operations in Byzantine emulations [7]. *Weak fork-linearizability* [7] allows two clients, after being forked, to observe a single operation of the other one (at-most-one-join), and that the real-time order induced by linearizability may be violated by the last operation of each client (weak real-time order).

In this paper, we explore the fundamental assumptions required for building a Byzantine service emulation. Up to now, all fork-consistent emulation protocols have required the service to execute non-trivial computation steps, i.e., the service must be implemented by an object of *universal* type [1], capable of *read-modify-write* operations [8]. We show the surprising result that this requirement can be dropped, and implement fork-consistent emulation protocols only from memory objects, so-called *registers*. They provide simple read and write operations and represent one of the weakest forms of computational objects. A long tradition of research has already addressed how to realize powerful abstractions from weaker base objects (e.g., [1,9]).

Specifically, we propose the first *fork-linearizable Byzantine emulation* of a universal object only from *registers*. Our algorithm necessarily offers abortable operations because a wait-free construction of a universal object from registers is not possible in an asynchronous system using only registers [1]. Moreover, we give an algorithm for a *weakly fork-linearizable Byzantine emulation* of a shared memory only from registers. It allows wait-free client operations when the underlying registers are correct.

Our two algorithms may directly replace the computation-based constructions in the existing respective emulations of shared memory on Byzantine servers [6,7,10]. For instance, our second construction, which yields a weakly fork-linearizable Byzantine emulation, allows to eliminate the server code from Venus [10]. Currently, Venus runs server code implemented by a *cloud computing* service, but our construction may realize it from a *cloud storage* service. For practical systems this can make a big difference in cost because full-fledged servers or virtual machines (e.g., Amazon EC2) are typically more expensive than simple disks or cloud-based key-value stores (e.g., Amazon S3).

Note that although our approach uses a collection of registers, we refrain from making more specific failure assumptions on them. Our remote service is comprised of registers, and as soon as one register is faulty, we consider the service to be faulty. It is conceivable to use fault-prone registers in our algorithms. Standard methods implementing robust shared registers from fault-prone base registers show how to *tolerate* up to a fraction of Byzantine base registers [11]. This extension, which is orthogonal to our work, would further refine our notion of graceful service degradation with faulty base objects.

Related Work The notion of fork-linearizability was introduced by Mazières and Shasha [3]. They implemented a fork-linearizable multi-user storage system called SUNDR. An improved fork-linearizable storage protocol is described by Cachin *et al.* [4]; it reduces the communication complexity compared to SUNDR from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. More recently, fork-linearizable Byzantine emulations have been extended to *universal services* [12]. All fork-linearizable emulations are blocking and sometimes require one client to wait for another client to complete [4].

In order to circumvent blocking the clients, Majuntke *et al.* [6] propose the first *abortable* fork-linearizable storage implementations. Their work takes up the notion of an abortable object introduced by Aguilera *et al.* [13]. They demonstrated, for the first time, how an abortable (and, hence, obstruction-free [14]) universal object can be constructed from abortable registers, which are base objects weaker than registers. In more recent work, it has been shown that abortable objects can be boosted to wait-free objects in a partially synchronous system [15]. This makes our Byzantine emulations of abortable objects very attractive in practical systems.

Actually implemented systems offering data storage integrity through forking consistency semantics include SUNDR (LKMS) [16], which realizes the protocol of Mazières and Shasha [3]. Furthermore, Cachin *et al.* [17] add fork-linearizable semantics to the Subversion revision control system, such that integrity and consistency of the server can be verified. The "blind stone tablet" of Williams *et al.* [18] provides fork-linearizable semantics for an untrusted database server; it may abort conflicting operations. Using a relaxation of fork-linearizability, called *fork-* consistency*, Feldman *et al.* [19] introduce a lock-free implementation for online collaboration that protects consistency and integrity of the service against a malicious provider.

Cachin *et al.* [7] present the storage service FAUST, which emulates a shared memory in a wait-free manner by exploiting the notion of *weak fork-linearizability*. It relaxes fork-linearizability in two fundamental ways: (1) after being forked, two clients may observe each others' operations once more and (2) the real-time order of the last operation of each client is not preserved. FAUST incorporates client-to-client communication in a higher layer, which ensures that all operations become eventually consistent over time (or the server is detected to misbehave). The Venus system [10] implements the mechanisms behind FAUST and describes a practical solution for ensuring integrity and consistency to the users of cloud storage.

Li and Mazières [20] study storage systems, built from $3f+1$ server replicas, where more than f replicas are Byzantine faulty. Their storage protocol ensures *fork-* consistency*. Similar to weak fork-linearizability, fork-* consistency allows that two forked clients observe again at most one common operation.

Contributions We present, for the first time, Byzantine emulations with forking consistency conditions only from *registers*, instead of more powerful computation objects. Any number of registers may be affected by Byzantine failures. Our constructions are linearizable provided that the base registers are correct. The constructions are:

– A register-based abortable Byzantine emulation of a fork-linearizable universal type.
– A register-based wait-free Byzantine emulation of weak fork-linearizable shared memory.

In Section 1, we discuss related work; Section 2 introduces the underlying system model. The two main constructions are given in Sections 3 and 4. The paper concludes in Section 5. The correctness proofs of the protocols can be found in our Technical Report [21].

2 System Model

We consider a distributed system consisting of $n > 1$ *clients* $C_1, \ldots, C_n$ that communicate through shared *objects*. Each such base object has a *type* which is given by a set of *invocations*, a set of *responses*, and by its *sequential specification*. The sequential specification defines the allowed sequences of invocations and responses. An *invocation* and the corresponding *response* constitute an *operation* of an object. A collection of base objects is used to implement high-level objects, where clients execute algorithm A, consisting of n state machines $A_1, \ldots, A_n$ (where C_i implements A_i). When client C_i receives an *invocation* of an operation to the high-level object, it takes steps of A_i, where it (1) either invokes an operation on some base object, (2) or receives the response to its previous invocation to a base object, (3) or it performs some local computation. At the end of a step, C_i changes its local state and possibly returns a response to the pending high-level operation.

An *execution* of algorithm A is defined as the (interleaved) sequence of invocation and response events. Every execution induces a *history* which is the sequence of invocations and responses of the high-level operations. If σ is a history of an execution of algorithm A, then $\sigma|_{C_i}$ denotes the subsequence of σ containing all events of client C_i. For sequence σ and operation o, $\sigma|^{o}$ denotes the prefix of σ that ends with the last event of o. We say that a response *matches* an invocation, if both are events of the same operation. An operation is called *complete*, if there exists a matching response to its invocation, else *incomplete*. We assume that each client invokes a new operation only after the previous operation has completed. A history consisting only of matching invocation/response pairs is called *well-formed*. Operation o *precedes* operation o' in a sequence of events σ ($o <_\sigma o'$) iff o is complete and the response of o happens before the invocation of o'. If o precedes o' we denote o and o' as *sequential*, if neither one precedes the other, then o and o' are said to be *concurrent*.

For the proposed *abortable* construction (Sec. 3), we introduce the special response ABORT. A complete operation o is called *unsuccessful* ("o is aborted"), if it returns ABORT, else it is called *successful* ("o successfully completes"). The formal definition of an *abortable* object comprises a non-triviality property which allows aborts only under concurrency [13].

Clients may fail by *crashing*, i.e. they stop taking steps and hence, the last operation of each client might be *incomplete*. Base objects may deviate arbitrarily from their specification exhibiting *non-responsive-arbitrary faults* [22] (called *Byzantine*). Clients have access to a digital signature scheme used by each client to *sign* its data such that any other client can determine the authenticity of a datum by *verifying* the corresponding signature. We assume that signatures cannot be forged.

All constructions appearing in this paper are based on *atomic registers*. An atomic register provides two operations, *read* and *write*[4]. Operation *write*(v) stores value v from domain *Values* into the register. A call of *read*() returns the latest written value from the register or the special value $\perp$ if no value has been written. As the register is atomic, its history satisfies linearizability [1], i.e. operations seem to appear as sequential, atomic events[5]. Further, the atomic registers used allow single-writer-multiple-reader access (SWMR), i.e. to each register we assign a dedicated client that may call *write* and *read*, while all other clients may only call *read* to that register.

A sequence of operations π satisfies *weak real-time* order of σ if π, excluding the last operation of each client in π, satisfies real-time order of σ. *Causality* between two operations depends on the type of the implemented object[6]. For two operations of a shared memory o and o' in σ, o *causally precedes* o' ($o \to_\sigma o'$), if o, o' are called by the same client and o happens before o', or if o' is a READ operation that returns the value written by WRITE operation o. The next definition formalizes the notion of *fork-linearizability* [4] and *weak fork-linearizability* [7]; for a formal definition of the term *possible view* as well as the above-mentioned notions we refer to the Technical Report [21].

Definition 1 Let σ be a history of an object of type T and for each client C_i there exists a sequence of events π_i such that π_i is a possible view of σ at C_i with respect to T. History σ is *fork-linearizable* with respect to object type T if for each client C_i:

[4] We type operation calls to base registers in *italic* font and calls to constructed objects in CAPITALS.

[5] Hence, the "latest written value" is well-defined.

[6] As causality is needed to define *weak fork-linearizability*, here, we give causality for a *shared memory*, which is the type we implement with weak fork-linearizability.

1. π_i preserves the real-time order of σ, and
2. for every client C_j and for every $o \in \pi_i \cap \pi_j$, it holds $\pi_i|^o = \pi_j|^o$.

History σ is *weak fork-linearizable* with respect to object type T if for each client C_i:

1. π_i preserves the weak real-time order of σ, and
2. for every operation $o \in \pi_i$ and every operation $o' \in \sigma$ such that $o' \rightarrow_\sigma o$, it holds that $o' \in \pi_i$ and that $o' <_{\pi_i} o$, and
3. (At-most-one-join) for every client C_j and every two operations $o, o' \in \pi_i \cap \pi_j$ by the same client such that $o <_\sigma o'$, it holds $\pi_i|^o = \pi_j|^o$.

The notion of a *Byzantine emulation* [4] allows us to formally define the safety and liveness properties of our protocols. Note that the liveness condition of abortable operations is weaker than *wait-freedom* but still not weaker than *obstruction-freedom* [13].

Definition 2 An algorithm A *emulates* an object of type T on a set of Byzantine base objects B with $\{$fork$|$weak fork$\}$-linearizability whenever following conditions hold:

1. If all objects in set B are correct, the history of every fair[7] and well-formed execution of A is linearizable with respect to type T, and
2. the history of every fair and well-formed execution of A is $\{$fork$|$weak fork$\}$-linearizable with respect to type T.

Such an emulation is *wait-free* (*abortable* resp.), iff every fair and well-formed execution of the protocol with correct base objects is wait-free [1] (abortable [13] resp.).

3 A Fork-linearizable Universal Type

In this section we present as our first main contribution an abortable fork-linearizable Byzantine emulation of a universal type implemented from atomic registers. The shared object ensures fork-linearizability in the presence of any number of faulty base registers. High-level operations are *abortable* [13], i.e. under concurrency, the special response ABORT may be returned. The functionality of a universal type T is encoded in the procedure APPLY_T. For client C_i, state s and operation o, $\text{APPLY}_T(s, o, i)$ returns (s', res), where s' is the new state of the universal object, res the computation result, and where the sequence of invoking $\text{APPLY}_T(s, o, i)$ and returning (s', res) is defined by the sequential specification of type T.

Our algorithm uses timestamp vectors called *versions* whose order reflects the real-time order in which operations are applied to the shared object. Each operation carries a version and the linearization of operations is achieved through the use of an INC&READ counter object C with two atomic operations INC&READ and READ. An invocation to INC&READ(C) advances the counter object C and returns a value which is higher than any value returned before, and READ(C) returns the current value of the counter object. An implementation of the INC&READ counter is given in the Technical Report [21] together with its formal properties. Our implementation uses wait-free atomic registers as base objects which makes it a wait-free variant of the abortable INC&READ counter described by Aguilera *et al.* [13].

[7] For a formal definition we refer to standard literature [23]

3.1 Algorithm Ideas

Universal Type To implement universal type T, we use n SWMR registers $R_1, \ldots, R_n$ such that client C_i can read from all registers but may write only to R_i. The registers store states of the universal object. To implement high-level operations, client C_i reads from the register which holds the most current state, applies the relevant state transformation, and writes the new state to R_i. Note, that all information are digitally signed by the clients as base objects are untrusted. Thereby, operations "affect" each other which leads to the following relation on operations: Operation o of C_i *affects* operation o' of C_j, if during o', C_j is able to verify the signature of C_i on state s that has been written during o and if C_j executes APPLY_T on s during o'; further, an operation of C_i *affects* each later operation of C_i.

Concurrency detection We allow operations to abort under concurrency for two reasons: there is no wait-free construction of a universal type from registers, as shown by Herlihy [1], and no fork-linearizable protocol can be wait-free in all executions, as shown in a more recent work of Cachin *et al.* [4]. Cachin's impossibility is based on two runs, indistinguishable for the reader: In the first run a READ operation does not return value v as it is concurrently written, while in the second run v has been previously written and is hidden by malicious registers. To avoid such a situation, our protocol implements a concurrency detection mechanism [13] using INC&READ counter object C. If concurrency is detected, a pending operation is aborted. At the invocation of a high-level operation o, our protocol calls INC&READ(C) and remembers the timestamp returned. At the end of o, READ(C) is executed to check whether counter C still returns the same timestamp. If not, another operation o' was invoked during o — thus, o is aborted. Else, if at the end of o C has not been changed, all successful operations either terminated before o or will be invoked after o has terminated. This is because the timestamps, returned from INC&READ, are used to linearize operations: The current state is written together with the timestamp, and the timestamp is used to determine the most recent state. Hence, all other operations invoked so far write a state with a lower timestamp than o. Consequently, such operations are linearized before o and only the state written by o can be read by later operations.

Fork-Linearizability In addition to the timestamp from INC&READ counter C, each operation is assigned a vector of timestamps of length n, called *version*. The order relation $\leq$ defined on versions respects real-time order and the "affected by" relation on operations. The idea is that each operation reads the most recent version from the storage, increments its own entry and writes the new version back to the storage. Thereby, each operation checks, if the version it reads, has been affected by the version of its own last successful operation, i.e. one which was not aborted. If the last successful operation of client C_i is hidden from C_j, then C_i does not accept operations of C_j as they have *not* been affected by the last successful operation of C_i. This ensures that the views of the clients after a forking attack are not rejoined. This principle is based on ideas of Mazières and Shasha [3], and Cachin *et al.* [4]. To apply it to this work, we have to add a specific handling for aborted operations: If operation o of client C_i is aborted, C_i cannot expect that o will affect later operations. However, it is still possible that some operation of C_j is affected by aborted o. In this case we call o *relevant* for C_j (refer to the Technical Report [21] for a formal definition).

3.2 Description of Algorithm 1

We now describe the steps preformed by client C_i when executing high-level operation o. The algorithm is given as Algorithm 1, the variables used are collected in Figure 1.

The protocol is framed by INC&READ(C) and READ(C) calls to the counter object C implementing the concurrency detection mechanism (lines 1.2 and 1.14). If the returned timestamps are not equal, the operation is aborted in line 1.16. In lines 1.3–1.5, the client reads from all atomic registers $R_1, \ldots, R_n$ and determines by means of the assigned timestamps the index l of the register holding the latest written data $\langle ts_l, V_l, s_l, sig_l \rangle$, where ts_l is a timestamp, V_l is the version, s_l is the state and sig_l is a signature. If some data have been written to R_l, the signature of the content of R_l is verified (line 1.6). Then, client C_i checks whether the read version V_l is not smaller than V_{suc} the version of its own last successful operation (line 1.7). When the check is passed the new state of the universal object and the computation result is computed by calling APPLY$_T(s_l, o, i)$ (line 1.8). Finally the new version for operation o has to be computed. This is done by taking the per-entry maximum of version V, which is the local version of C_i, and V_l, and by incrementing the ith entry (lines 1.9–1.11). After signing the current timestamp, the new version V, and new state s in line 1.12, client C_i writes ts, V, s and the signature into register R_i (line 1.13). If operation o is successful, version V is stored as last successful version V_{suc} and the computation result is returned (lines 1.17–1.19).

Fig. 1. Variables used in Algorithm 1

C INC&READ counter object, initially 0
$R_1, \ldots R_n$ SWMR atomic register, initially $\langle 0, (0, ..., 0), \perp, \perp \rangle$ `/* ts+version+state+sig */`
ts, ts', ts_l, cn integer, initially 0 `/* timestamp & counter */`
$V[1..n], V_l[1..n], V_{suc}[1..n]$ array of integers, intially $(0, ..., 0)$ `/* version */`
s, s_l state, initially $\perp$ `/* state */`
res operation result, initially $\perp$ `/* return value */`
sig, sig_l signature, initially $\perp$ `/* signature */`

3.3 Correctness Arguments

In this section we argue why Algorithm 1 satisfies fork-linearizability. The goal is to construct for each client C_i a view π_i of σ that satisfies the properties of fork-linearizability. To construct π_i, we simplify our argumentation by ignoring operations that are not relevant for C_i. Recall, any operation is *relevant* for client C_i that affects C_i's last successful operation. Hence, operations that are not relevant for client C_i do not change the object's state from C_i's point of view. Thus, we can order them arbitrarily among the operations in π_i and the resulting sequences still satisfy fork-linearizability.

Algorithm 1: Universal Object Implementation, Code of Client i

```
1.1  EXECUTE($o$) do
1.2      $ts \leftarrow$ INC&READ($C$)        /* increment and read from counter */
1.3      for $j = 1, \ldots, n$ do
1.4          $\langle ts_j, V_j, s_j, sig_j \rangle \leftarrow read(R_j)$        /* low-level atomic read */
1.5      let $l$ be such that $ts_l = \max_{1 \leq j \leq n}(ts_j)$      /* find register with most
         recent data */
1.6      if $V_l \neq [0 \ldots 0] \wedge \neg \mathrm{verify}_l(sig_l, \langle ts_l, V_l, s_l \rangle)$ then halt          /* signature
         verified? */
1.7      if $\exists k : V_{suc}[k] > V_l[k]$ then halt          /* fork-linearizability check
         passed? */
1.8      $\langle s, res \rangle \leftarrow$ APPLY$_T(s_l, o, i)$        /* compute new state + result */
1.9      for $j = 1, \ldots, n$, $j \neq i$ do
1.10         $V[j] \leftarrow \max(V[j], V_l[j])$                          /* determine
1.11     $V[i] \leftarrow V[i] + 1$                                         new version */
1.12     $sig \leftarrow \mathrm{sign}_i(ts||V||s)$       /* signature on ts, version, state */
1.13     $write(R_i, \langle ts, V, s, sig \rangle)$                /* low-level atomic write */
1.14     $ts' \leftarrow$ READ($C$)                               /* read from counter */
1.15     if $ts \neq ts'$ then
1.16         return ABORT                              /* concurrency detected */
1.17     else
1.18         $V_{suc} \leftarrow V$                 /* reset last successful version */
1.19         return $res$                                     /* return result */
```

The idea behind the construction of the π_i in the proof is that operations are ordered according to their assigned versions. The proof shows that this order respects the "affected by" relation, the sequential specification of a universal type, and the real-time order. As during an operation the new version is computed using the client's last version and the read version, proving "affected by" and real-time order is straightforward. The core of the proof is to show that the order of version also respects the sequential specification. We sketch the intuition behind this with the following argument leading to a contradiction:

Assume that some operation o_c is not affected by the most recent state of the universal object, which has been written by relevant operation o_b, but is affected by an older state written by operation o_a. In this case, the clients of o_b and o_c are forked, and neither o_b nor o_c affect each other. We argue, that in such a situation, there is no relevant operation that has been affected by both o_b and o_c, as such an operation would join the two clients violating fork-consistency. We assume for contradiction, that a relevant operation o_{join} of client C_{join}, affected by o_b and o_c exists which is also the first among such operations (see Figure 2). Operation o_{join} is affected by o_{join_suc}, the last successful operation of C_{join} previous to o_{join}, *and* by o_r that wrote the state which is read during o_{join}. Hence, without loss of generality o_{join_suc} is affected by o_b while o_r is affected by o_c. During operation o_{join_suc}, client C_{join} raises its value in the version to $V[join]_{join\_suc}$. This implies that o_{join} only accepts versions where the $join$th entry is at least $V[join]_{join\_suc}$ (line 1.7). As o_{join_suc} is not on the path of "affected by" relations from o_c to o_r, o_{join} would block while reading the state of o_r which is a contradiction. Thus, o_{join} does not exist.

Finally, it follows directly from the described construction, that sequences π_i satisfy the no-join property. To complete the correctness proof of the Byzantine emulation, we show that when all base objects are correct, no operation blocks and that no operation trivially aborts.

$$\begin{tikzcd}
o_a \arrow[rr, bend right] & o_b \arrow[rr, bend left] & o_c \arrow[drr] & o_{join\_suc} \arrow[rr] & & o_{join} \arrow[r] & \ldots \\
 & & & & o_r \arrow[ur] & &
\end{tikzcd}$$

Fig. 2. Correctness Idea of Algorithm 1. Arrows denote the "affected by" relation.

4 A Weak Fork-Linearizable Shared Memory

In this section we describe as our second contribution a wait-free, weak fork-linearizable Byzantine emulation of a shared memory implemented from atomic registers. The presented construction satisfies weak fork-linearizability in the presence of any number of faulty base objects. The implemented shared memory provides n atomic registers, such that each client can write to one dedicated register exclusively and may read from all registers. Operation WRITE(v), called by client C_i, writes value v to C_i's register. Operation READ(i) returns the last written value from C_i's register, and may be called by any client. Our algorithm makes use of an atomic single-writer snapshot object S with n components [24,25]. Snapshot object S provides two atomic operations: UPDATE(d, S, i), that changes the state of component i of S to d, and SCAN(S) that returns vector $(d_1, \ldots, d_n)$ such that d_i is the state of component i of S, $i = 1 \ldots, n$. Formally, d_i is the state written by the last UPDATE to component i prior to SCAN. It has been shown, that such a shared snapshot object can be wait-free implemented only from registers [24,25].

4.1 Algorithm Ideas

Each client locally maintains a timestamp that respects causality and real-time order of its *own* operations. As the basic principle, during each operation this timestamp is written to the shared memory and timestamps left by other operations are read. For each client C_i our implementation uses two registers only C_i may write to, but which can be read by all clients. The first one is needed to store value and timestamp written by C_i's WRITE operations and is implemented by a SWMR atomic register W_i (i.e. registers $W_1, ..., W_n$ in total). The second "register" is required to store the latest timestamp of C_i's READ operations. It is implemented as the ith component within the single-writer snapshot object with n components, S.

During READ(j) operation of C_i, C_i's current timestamp is written to S using UPDATE, thereafter, C_i reads a timestamp-value pair from register W_j (using low-level *read*). High-level WRITE(v) of C_i proceeds analogously: C_i writes its current timestamp plus value v to register W_i using low-level *write*, thereafter, it reads *all* components from S using SCAN. By this, operations are able to observe each other, as expressed in the relation "seen": We say that a WRITE operation o_w of C_j *sees* a READ operation o_r of C_i with timestamp ts if C_i digitally signed ts and updated the ith component of S by signed ts during o_r and, if during o_w, C_j scanned S and was able to verify the signature of C_i on ts; READ operation o_r *sees* WRITE operation o_w if o_r returns the value written by o_w.

This construction guarantees the following property on interleaved high-level operations: Whenever high-level READ(j) o_r of C_i and WRITE(v) o_w of C_j appear in an execution such that o_r does not return v but a value written before v, then, by regularity of the atomic base registers, o_w*.write*[8] does not precede o_r*.read*, i.e., o_r*.read* has been invoked *before* o_w*.write* finishes. Consequently, o_r.UPDATE precedes o_w.SCAN (see Figure 3). Thus, if o_r does not "see" o_w, then o_w "sees" o_r. A similar property on interleaving operations has also been leveraged in our previous work [26] as well as by Aguilera *et al.* [9].

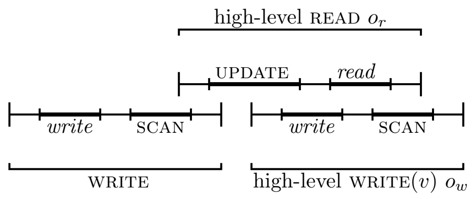

Fig. 3. Basic principle implemented by Algorithm 2.

We can expect that client C_j writes information during its next WRITE operation such that future operations of C_i may verify whether operation o_w actually has seen operation o_r. More concrete, if READ o_r has seen WRITE o_w then the client checks during o_r whether the next WRITE operation after o_w (of the same client as o_w), has seen READ operation o_r or a newer one. Else, the base objects are faulty, as shown in the following example: Let o_w and o'_w be two sequential WRITE operations of C_i, o'_w precedes READ operation o_r of C_j but it is hidden by the malicious base objects such that o_r sees only o_w. As o'_w precedes o_r, o'_w cannot see o_r. However, as o_r sees o_w, it expects that o'_w will see o_r. The next WRITE operation o''_w of C_i will write this information. If client C_j sees o''_w, which would violate weak fork-linearizablility, the check, explained above, is not passed.

Fig. 4. Variables used in Algorithm 2

S, atomic snapshot object with n componenets, initially $((0, \bot), ..., (0, \bot))$ `/* timestamp+sig */`
$W_1, \dots, W_n$, SWMR atomic registers, initially $(\bot, 0, \emptyset, \emptyset, \bot)$ `/* val+ts+rs+ws+sig */`
v, wv value, initially $\bot$ `/* value written to storage */`
$wts, ots, i, k, r, r', w, w', tmp_1, \dots, tmp_n$ integer, initially 0 `/* timestamps + temp. variables */`
$read\_seen[1..n][1..n], write\_seen[1..n][1..n]$, `/* matrices of seen`
$r\_write\_seen[1..n][1..n]$, matrix of sets of pairs (integer, integer), initially $\emptyset$
`operations */`
$sig, sig_1, \dots, sig_n$ signature, initially $\bot$ `/* signatures */`

4.2 Description of Algorithm 2

This section explains the steps taken by client C_i to implement high-level READ and WRITE operations. The algorithm is given as Algorithm 2, its variables in Figure 2.

At invocation of high-level READ(j), client C_i increments its local timestamp and generates a digital signature of it. The signed timestamp is stored to snapshot object S using operation UPDATE$((ots, sig), S, i)$ (lines 2.2–2.4). Then, client C_i reads register W_j and verifies the signature (line 2.5–2.6). The content of register W_j contains the written value wv, the corresponding timestamp wts, as well as two matrices r_read_seen and r_write_seen. Both matrices are of size $n \times n$ where each entry holds a set of integer pairs (r, w). Client C_i maintains a variable $read\_seen$ of the same type, where a pair $(r, w) \in read\_seen[i][j]$ denotes that READ of client C_i with timestamp r has seen WRITE of client C_j with timestamp w. Analogously, client C_i maintains a second matrix $write\_seen$, where $(r, w) \in write\_seen[i][j]$ denotes that WRITE of client C_i with timestamp w has seen READ of client C_j with timestamp r. In the next step (line 2.7), client C_i "merges" variables r_read_seen and $read\_seen$. The merge procedure returns for each entry of two $n \times n$ set matrices A, B set $A[i][j] \cup B[i][j]$, $i, j = 1, \dots, n$. Then, C_i adds a pair consisting of its current timestamp and timestamp wts from W_j to $read\_seen[i][j]$. To ensure weak fork-linearizability, client C_i calls procedure "check" (line 2.9). If all checks are passed, C_i merges r_write_seen and $write\_seen$ and returns value wv (lines 2.10–2.11).

At invocation of WRITE(v), client C_i increments its timestamp (line 2.13). It digitally signs value v, its timestamp, and variables $read\_seen$ and $write\_seen$ to write to register W_i (lines 2.14–2.15). Next, it reads all timestamps of READs by calling SCAN to snapshot object S (line 2.16). All entries in S are digitally signed and thus client C_i verifies the signatures (line 2.18). Then, it adds to all sets $write\_seen[i][k]$ $(k = 1, \dots, n)$ a pair consisting of the timestamp of the kth component of S and C_i's current timestamp (line 2.19). Finally, client C_i successfully returns (line 2.20).

Procedure "check" implements the principle sketched in section 4.1 for n clients. It ensures that *weak fork-linearizability* is never violated. The procedure, called by C_i

[8] The notation $x.y$ denotes the call of low-level operation y during high-level operation x.

Algorithm 2: Weak Fork-Linearizable Memory for n Clients, Code of Client C_i

```
2.1  READ($j$) do
2.2      $ots \leftarrow ots + 1$                              /* increment timestamp */
2.3      $sig \leftarrow \text{sign}_i(ots)$                   /* signature on timestamp */
2.4      UPDATE($(ots, sig), S, i$)       /* update call to snapshot object */
2.5      $(wv, wts, r\_read\_seen, r\_write\_seen, sig) \leftarrow read(W_j)$      /* low-level
         atomic read */
2.6      if not $\text{verify}_j(sig)$ then halt               /* signature verified? */
2.7      $read\_seen \leftarrow \text{merge}(read\_seen, r\_read\_seen)$     /* update read_seen */
2.8      $read\_seen[i][j] \leftarrow read\_seen[i][j].\text{add}((ots, wts))$  /* add seen write */
2.9      check()                                               /* check passed? */
2.10     $write\_seen \leftarrow \text{merge}(write\_seen, r\_write\_seen)$   /* update write_seen
         */
2.11     return $wv$                                           /* return read value */

2.12 WRITE($v$) do
2.13     $ots \leftarrow ots + 1$                              /* increment timestamp */
2.14     $sig \leftarrow \text{sign}_i(v, ots, read\_seen, write\_seen)$  /* signature on timestamp
         */
2.15     $write((v, ots, read\_seen, write\_seen, sig), W_i)$       /* low-level atomic
         write */
2.16     $\langle(tmp_1, sig_1), \ldots, (tmp_n, sig_n)\rangle \leftarrow$ SCAN($S$)  /* scan call to snapshot
         object */
2.17     for $k = 1, ..., n$ do
2.18         if not $\text{verify}_k(sig_k)$ then halt         /* signature verified? */
2.19         $write\_seen[i][k] \leftarrow write\_seen[i][k].\text{add}((tmp_k, ots))$  /* add all seen
             reads */
2.20     return OK                                             /* successfully return */

2.21 check() do
2.22     for $k = 1, ..., n$ do
2.23         forall $(r, w) \in read\_seen[k][i]$ do
                 /* check if own writes have seen read operations
                    reading my values                                   */
2.24             if $\exists(r', w') \in write\_seen[i][k]$ s.t. $w' > w$ and $w'$ minimal then
2.25                 if $r' < r$ then halt
2.26         forall $(r, w) \in read\_seen[i][k]$ do
                 /* check if own reads have been seen by other's
                    write operations                                    */
2.27             if $\exists(r', w') \in r\_write\_seen[k][i]$ s.t. $w' > w$ and $w'$ minimal then
2.28                 if $r' < r$ then halt
```

during READ(j) (line 2.21), moves through a loop performing two checks: The first check (line 2.24–2.25) considers the information left by clients during READ(i) operations (this information is stored in the ith column of $read\_seen$). If READ(i) with timestamp r of client C_k has seen WRITE of C_i with timestamp w, then it is tested whether the next WRITE of C_i has read (using SCAN) timestamp r or higher of client C_k. The check uses the local $write\_seen$ variable of C_i. The second check (line 2.27–

2.28) reviews the information left by client C_i during any $\textsc{read}(k)$ (which is kept in the ith row of $read\_seen$). If $\textsc{read}(k)$ with timestamp r of client C_i has seen WRITE of C_k with timestamp w, then we check whether the next WRITE of C_k has read (using SCAN) timestamp r or higher of client C_i. This check requires matrix r_write_seen, which has been fetched from W_j in line 2.5 before procedure "check" is called.

4.3 Correctness Arguments

In this section we give the intuition why Algorithm 2 satisfies the properties of a wait-free Byzantine emulation of a shared memory with weak fork-linearizability. Intuitively, the definition of weak fork-linearizability requires for each client C_i to construct a sequence π_i such that causality among operations, the sequential specification a shared memory, and weak real-time order is satisfied, and that two sequences π_i and π_j share the same prefix up to the second last common operation (at-most-one-join). The proof proceeds in steps, where in the first step all operations that have to be included in sequence π_i are causally ordered. Next, this order is extended such that it additionally respects the sequential specification. Intuitively, as all written values are digitally signed, the sequential specification never interferes with causality. The hardest step is to prove, that this order can be further refined such that it does not violate the weak real-time order. The intuition for this is given below as a proof by contradiction:

We assume that $\textsc{read}(j)$ operation o_r of client C_i does not return the latest value, written by WRITE operation o'_w, but an older value written by operation o_w (see Figure 5). Further, let o_r be not the last operation of C_i. During operation o_r, the pair (r, w)[9] is added to set $read\_seen[i][j]$. The data written by the next WRITE operation o''_w of C_i contains this information. Now, the algorithm prevents client C_j from reading the value written by o''_w which would violate weak real-time order (as o_r is ordered before o'_w according to the sequential specification). When during o''_r C_j sees operation o''_w, it finds the pair (r, w) in r_read_seen. As o'_w precedes o_r, it could not have seen o_r, thus $write\_seen[j][i]$ contains a pair (r', w') such that $r' < r$ and the check in line 2.25 is not passed. Hence, operation o''_r of client C_j would block — a contradiction. This implies that such a situation does not appear and the constructed order of operations also satisfies weak real-time order.

As the last step, showing that the sequences π_i satisfy the at-most-one-join property follows directly from a simple construction argument. To prove liveness, as required in the definition of a Byzantine emulation (Definition 2), we show that no operation blocks when all base objects are correct, which follows from the principle sketched in section 4.1 as in this case all checks are passed.

$$\begin{tikzcd}
\text{Client } C_j\text{:} & o_w \arrow[drr, bend right] & o'_w \arrow[rrr] & & & o''_r \\
\text{Client } C_i\text{:} & & & o_r \arrow[r] & o''_w \arrow[ur, dashed] &
\end{tikzcd}$$

Fig. 5. Correctness Ideas of Algorithm 2. Arrows denote the "seen" relation.

[9] We assume that operation o_x is assigned timestamp x.

5 Analysis & Conclusions

The abortable construction in Algorithm 1 requires n atomic registers plus n additional ones to implement the INC&READ counter. The presented construction has an overall communication complexity of $O(n^2)$, as the size of the version vectors used in Algorithm 1 is linear in the number of clients n and as a linear number of such version vectors are exchanged per operation. In contrast, the *lock-step* protocol of Cachin *et al.* [4], also based on linear size version vectors, has an overall communication complexity of $O(n)$. This difference results from the fact that the server objects used by Cachin *et al.* are computationally strong enough to select the latest written version vector while in Algorithm 1 the client is required to read from *all* register objects to find the latest one by itself. For the implementation of Algorithm 2, we need n atomic registers plus $2n$ additional ones for the atomic snapshot object. Algorithm 2, uses matrices of size $n \times n$ where the size of each entry depends on the total number of operations N, resulting in a communication complexity of $O(N \cdot n^2)$. We leave for future research whether this complexity can be reduced by implementing a "garbage collection". However, both of our algorithms require only a linear number of base registers.

We have shown by ways of two protocols as a first known result that fork-consistent semantics can be implemented only from registers. Our first protocol satisfies fork-linearizability and implements a shared object of universal type. Similar to non-fork-consistent universal constructions from registers, our protocol may abort operations under concurrency. Hence, fork-linearizability may be "added" to such protocols without making additional assumptions. Our second protocol implements a shared memory object that ensures *weak* fork-linearizability and where operations are wait-free as long as the base registers behave correctly. Weak fork-linearizability is the strongest known fork-consistency property that may be implemented in a wait-free manner. Although it weakens fork-linearizability, it has shown to be of practical relevance [7]. Moreover, our second algorithm shows for the first time that registers are sufficient to implement a fork-consistent shared memory. So far, all existing implementations are based on computationally stronger objects (featuring read-modify-write operations [8]). We leave as an open question whether there is a weak fork-linearizable construction of a universal type providing a stronger liveness condition than *abortable* in the fault-free case.

References

1. Herlihy, M.: Wait-Free Synchronization. ACM Trans. Program. Lang. Syst. **13**(1) (1991) 124–149
2. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. Report, National Institute of Standards and Technology (NIST) (January 2011) Available online at `http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf`.
3. Mazières, D., Shasha, D.: Building Secure File Systems out of Byzantine Storage. In: PODC, New York, NY, USA, ACM (2002) 108–117
4. Cachin, C., Shelat, A., Shraer, A.: Efficient Fork-Linearizable Access to Untrusted Shared Memory. In: PODC, New York, NY, USA, ACM (2007) 129–138
5. Herlihy, M.P., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492

6. Majuntke, M., Dobre, D., Serafini, M., Suri, N.: Abortable Fork-Linearizable Storage. In: Proceedings of the 13th International Conference on Principles of Distributed Systems. OPODIS '09, Berlin, Heidelberg, Springer-Verlag (2009) 255–269
7. Cachin, C., Keidar, I., Shraer, A.: Fail-Aware Untrusted Storage. SIAM Journal on Computing **40**(2) (April 2011) 493–533
8. Kruskal, C.P., Rudolph, L., Snir, M.: Efficient Synchronization of Multiprocessors with Shared Memory. ACM Trans. Program. Lang. Syst. **10** (October 1988) 579–601
9. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic Atomic Storage Without Consensus. J. ACM **58** (April 2011) 7:1–7:32
10. Shraer, A., Cachin, C., Cidon, A., Keidar, I., Michalevsky, Y., Shaket, D.: Venus: Verification for Untrusted Cloud Storage. In: Proceedings of the 2010 ACM Workshop on Cloud Computing Security. CCSW '10, New York, NY, USA, ACM (2010) 19–30
11. Malkhi, D., Reiter, M.K.: Byzantine Quorum Systems. Distributed Computing **11**(4) (1998) 203–213
12. Cachin, C.: Integrity and Consistency for Untrusted Services. In: Proceedings of the 37th international conference on Current trends in theory and practice of computer science. SOFSEM'11, Berlin, Heidelberg, Springer-Verlag (2011) 1–14
13. 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
14. 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
15. 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
16. Li, J., Krohn, M., Mazières, D., Shasha, D.: Secure Untrusted Data Repository (SUNDR). In: Proc. 6th Symp. Operating Systems Design and Implementation (OSDI 04). (2004) 121–136
17. Cachin, C., Geisler, M.: Integrity Protection for Revision Control. In: Proceedings of the 7th International Conference on Applied Cryptography and Network Security. ACNS '09, Berlin, Heidelberg, Springer-Verlag (2009) 382–399
18. Williams, P., Sion, R., Shasha, D.: The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties. In: Proc. NDSS. (2009)
19. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: SPORC: Group Collaboration on Untrusted Resources. In: Proc. 9th Symposium on Operating Systems Design and Implementation (OSDI 10), Vancouver, BC (October 2010)
20. Li, J., Mazières, D.: Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In: Proc. NSDI. (2007)
21. Majuntke, M., Dobre, D., Cachin, C., Suri, N.: Fork-consistent constructions from registers. In: Technical Report TR-TUD-DEEDS-09-01-2011. (September 2011) Available online at `http://www.deeds.informatik.tu-darmstadt.de/matze/fc_wo_sc_2011.pdf`.
22. Jayanti, P., Chandra, T.D., Toueg, S.: Fault-tolerant Wait-free Shared Objects. J. ACM **45**(3) (1998) 451–500
23. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1998)
24. Attiya, H., Guerraoui, R., Ruppert, E.: Partial Snapshot Objects. In: Proc. SPAA. (2008) 336–343
25. Fich, F.E.: How Hard Is It to Take a Snapshot? In: Proc. SOFSEM. (2005) 28–37
26. Dobre, D., Majuntke, M., Suri, N.: On the time-complexity of robust and amnesic storage. In: OPODIS. (2008) 197–216