RZ 3841 (# ZUR1302-026) 02/18/2013
Computer Science 16 pages

Research Report

On Verifying the Consistency of Remote Untrusted Services

C. Cachin*, Olga Ohrimenko‡

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

‡Brown University
Providence
RI 02912, USA

Research
Africa • Almaden • Austin • Australia • Brazil • China • Haifa • India • Ireland • Tokyo • Watson • Zurich

On Verifying the Consistency of Remote Untrusted Services

Christian Cachin[1]   Olga Ohrimenko[2]

February 18, 2013

Abstract

A group of mutually trusting clients outsources a computation service to a remote server, which they do not fully trust and that may be subject to attacks. The clients do not communicate with each other and would like to verify the correctness of the remote computation and the consistency of the server's responses. This paper presents the *Commutative-Operation verification Protocol (COP)* that ensures linearizability when the server is correct and preserves fork-linearizability in any other case. Fork-linearizability ensures that all clients that observe each other's operations are consistent in the sense that their own operations and those operations of other clients that they see are linearizable. COP goes beyond previous protocols in supporting wait-free client operations for sequences of commutative operations.

1 Introduction

With the advent of *cloud computing*, most computations run in remote data centers and no longer on local devices. As a result, users are bound to trust the service provider for the confidentiality and the correctness of their computations. This work addresses the *integrity* of outsourced data and computations and the *consistency* of the provider's responses. Consider a group of mutually trusting clients who want to collaborate on a resource that is provided by a remote minimally trusted server. This could be a wiki containing data of a common project, an archival document repository, or a groupware tool running in the cloud. A subtle change in the remote computation, whether caused inadvertently by a bug or deliberately by a malicious adversary, may result in wrong responses to the clients. Although the clients generally trust the provider, they would like to assess the integrity of the computation, to verify that responses are correct, and to check that they all get consistent responses.

In an asynchronous network model without communication among clients such as considered here, the server may perform a *forking attack* and omit the effects of operations by some clients in her responses to other clients. Not knowing which operations other clients execute, the latter group cannot detect such violations. The best achievable consistency guarantee in this setting is captured by *fork-linearizability*, introduced by Mazières and Shasha [15] for storage systems. Fork-linearizability ensures that whenever the server in her responses to a client C_1 has ignored a write operation executed by a client C_2, then C_1 can never again read a value written by C_2 afterwards and vice versa. From this property, clients can detect server misbehavior from a single inconsistent operation, which is much easier than comparing the effects of *all* past operations one-by-one.

Several conceptual [4, 14, 2, 3] and practical advances [19, 6, 13, 17] have recently been made that improve consistency checking and verification with fork-linearizability and related notions for remote storage and computation. The resulting protocols ensure that when the server is correct, the service is

[1] IBM Research - Zurich, CH-8803 Rüschlikon, Switzerland. cca@zurich.ibm.com.

[2] Brown University, Providence, RI 02912, USA. olya@cs.brown.edu.

linearizable and (ideally) the algorithm is *wait-free*, that is, every client's operations complete independently of other clients. It has been recognized, however, that read/write conflicts often cause such protocols to block; this applies to consistency verification for storage with fork-linearizable semantics [15, 4] and for other forking consistency notions [2, 3].

In this paper, we go beyond storage services and propose a new protocol for consistency verification of remote computations on a Byzantine server, called the *Commutative-Operation verification Protocol* or *COP*. It supports arbitrary functionalities, exploits commuting operations, and allows clients to operate concurrently and without blocking or aborting whenever feasible, while imposing fork-linearizable semantics. Through this guarantee Byzantine behavior of the server can be exposed easily. Clients may therefore verify the correctness of a service in an end-to-end way.

Support for wait-free operations is a key feature for collaboration with remote coordination, as geographically separated clients may operate with totally different timing characteristics. Consequently, previous work has devoted a lot of attention to identifying and avoiding blocking situations [15, 4, 11]. For example, read operations in a storage service commute and do not lead to a conflict. On the other hand, when a client writes to a data item concurrently with another client reading from the item, the reader has to wait until the write operation completes; otherwise, fork-linearizability is not guaranteed. If all operations are to proceed without blocking, though, it is necessary to weaken the consistency guarantees to fork-* linearizability [11] or weak fork-linearizability [3], for instance. COP is wait-free and never blocks because it aborts non-commuting operations that cannot proceed. Abortable operations have been introduced in this context by Majuntke et al. [14].

The *Blind Stone Tablet (BST)* protocol [19] supports an encrypted remote database hosted by an untrusted server that is accessed by multiple clients. Its consistency checking algorithm allows some commuting client operations to proceed concurrently, but only to a limited extent, as we explain below. Furthermore, the protocol guarantees fork-linearizability for database state updates, but does not ensure it for certain responses output by a client.

SPORC considers a groupware collaboration service whose operations may not commute, but can be made to commute by applying operational transformations. Through this mechanism, different execution orders still converge to the same state. All SPORC operations are wait-free and respect fork-* linearizability.

1.1 Contributions

This paper considers a generic service executed by an untrusted server and investigates protocols for consistency verification through fork-linearizable semantics. It explores the relation between commuting operations in the service specification and client operations that may proceed concurrently.

More concretely, this paper introduces the Commutative-Operation verification Protocol (COP) and makes three contributions:

1. COP is the first wait-free protocol that emulates an arbitrary functionality on a Byzantine server with fork-linearizability and supports commuting operation sequences.
2. COP allows clients to proceed at their own speed, regardless of the behavior of other clients, when they execute non-conflicting sequences of operations.
3. We formally prove COP correct and demonstrate that all completed operations and their responses respect fork-linearizability.

COP follows the general pattern of most previous fork-linearizable emulation protocols. For determining when to proceed with concurrent operations, it considers *sequences* of operations that jointly commute, in contrast to earlier protocols, which considered only isolated operations.

In COP, the server merely coordinates client-side operations but does not compute the results. This conceptually simple approach can be found in many related protocols [19, 8, 6] and practical collaboration systems (git[1], Mercurial[2]); it also represents the common trend of cloud computing to shift computation to the client and coordination to the cloud.

1.2 Related work

Storage protocols. Fork-linearizability has been introduced (under the name of *fork consistency*) together with the SUNDR storage system [15, 10]. Conceptually SUNDR operates on storage objects with simple read/write semantics. Subsequent work of Cachin et al. [4] improves the efficiency of untrusted storage protocols. A lock-free storage protocol with abortable operations, which lets all operations complete in the absence of step contention, has been proposed by Majuntke et al. [14].

FAUST [3] and Venus [17] go beyond the fork-linearizable consistency guarantee and model occasional message exchanges among the clients. This allows FAUST and Venus to obtain stronger semantics, in the sense that they eventually reach consistency (in the sense of linearizability) or detect server misbehavior. In the model considered here, fork-linearizability is the best possible guarantee [15].

Blind Stone Tablet (BST). The BST protocol [19] considers transactions on a common database, coordinated by the remote server. Clients first *simulate* a transaction on their own copy, potentially generating local output, then coordinate with the server for ordering the transaction. From the server's response the client determines if his transaction commutes with other, pending transactions invoked by different clients that were reported by the server. If they conflict, the client undoes the transaction and basically aborts; otherwise, he commits the transaction and relays it via the server to other clients. When a client receives such a relayed transaction, the client *applies* the transaction to its database copy.

BST has two limitations: First, because a client applies his own transactions only when all pending transactions by other clients have been applied to his own state, state changes induced by his transactions are delayed in dependence on other clients. Thus, he cannot always execute his next transaction from the modified state and obtain a correct output. Second, the notion of "trace consistency" in the analysis of the BST protocol considers only transactions that have been applied to the local state, not local output generated by the client. Hence fork-linearizability is not shown for the service responses but only for those transactions that clients have applied to their state (the former may occur long before the latter).

COP is strictly more general than BST, as it allows one client to execute multiple operations independently of the other clients, as long as his *sequence* of operations jointly commutes with the *sequence* of pending operations by other clients. Note that two operations o_1 and o_2 may independently commute with an operation o_3 from a particular starting state, but their concatenation, $o_1 \circ o_2$, may not commute with o_3.

Non-blocking protocols. SPORC [6] is a group collaboration system where operations do not need to be executed in the same order at every client by virtue of employing *operational transforms*. The latter concept allows to shift operations to a different position in an execution by transforming them according to properties of the skipped operations. Differently ordered and transformed variants of a common sequence converge to the same end state.

SPORC achieves fork-* linearizability [11], which is closely related to weak fork-linearizability [3]; both notions are relaxations of fork-linearizability that permit concurrent operations to proceed without

[1] `http://git-scm.com`

[2] `http://mercurial.selenic.com`

blocking, such that protocols become wait-free. The increased concurrency is traded for weaker consistency, as up to one diverging operation may exist between the views of different clients and cannot be detected.

FAUST [3], mentioned before, never blocks clients and enjoys eventual consistency, but guarantees only weak fork-linearizability.

In contrast to the SPORC and FAUST protocols, COP ensures the stronger fork-linearizability condition, where every operation is consistent as soon as it completes. SPORC is not weaker nor stronger than COP: On one hand, SPORC seems more general as it never blocks clients even for operations that do not appear to commute; on the other hand, though, SPORC only supports functions with suitably transformable operations and it has no provisions for handling conflicting operations, whereas COP works for arbitrary functions.

In all above protocols for generic services (BST, SPORC, and COP), all clients execute all operations. This is not necessary for storage protocols (SUNDR and FAUST) because their operations are simpler.

Last but not least, the protocol of Cachin [1] provides also fork-linearizable execution for generic services like COP. However, the approach is inherently blocking and requires the service to satisfy a cryptographic notion of “separated authenticated execution.”

1.3 Organization of the paper

The paper continues by introducing the notation and basic concepts in Section 2. The subsequent section presents COP and Section 4 proves that COP emulates an arbitrary functionality on a Byzantine server with fork-linearizability.

2 Definitions

System model. We consider an asynchronous distributed system with n clients, $C_1, \ldots, C_n$ and a server S, modeled as processes. Each client is connected to the server through an asynchronous, reliable communication channel that respects FIFO order. A protocol specifies the operations of the processes. All clients are *correct* and follow the protocol, whereas S operates in one of two modes: either she is *correct* and follows the protocol or she is *Byzantine* and may deviate arbitrarily from the specification.

Functionality. We consider a deterministic *functionality* F (also called a type) defined over a set of *states* $\mathcal{S}$ and a set of *operations* $\mathcal{O}$. F takes as arguments a state $s \in \mathcal{S}$ and an operation $o \in \mathcal{O}$ and returns a tuple (s', r), where $s' \in \mathcal{S}$ is a state that reflects any changes that o caused to s and $r \in \mathcal{R}$ is a response to o:

$$(s', r) \leftarrow F(s, o).$$

This is also called the *sequential specification* of F.

We extend this notation for executing a sequence of operations $\langle o_1, \ldots, o_k \rangle$, starting from an initial state s_0, and write

$$(s', r) \;=\; F(s_0, \langle o_1, \ldots, o_k \rangle)$$

for $(s_i, r_i) = F(s_{i-1}, o_i)$ with $i = 1, \ldots, k$ and $(s', r) = F(s_k, r_k)$. Note that an operation in $\mathcal{O}$ may represent a batch of multiple application-level operations.

Commutative Operations. Commutative operations of F play a role in protocols that may execute multiple operations concurrently. Two operations $o_1, o_2 \in \mathcal{O}$ are said to *commute in a state s* if and only if these operations, when applied in different orders starting from s, yield the same respective states and responses. Formally, if

$$\begin{array}{lllll} (s', r_1) \leftarrow F(s, o_1), & (s'', r_2) \leftarrow F(s', o_2); & \text{and} \\ (t', q_2) \leftarrow F(s, o_2), & (t'', q_1) \leftarrow F(t', o_1) \end{array}$$

then

$$r_1 = q_1,\ r_2 = q_2,\ s'' = t''.$$

Furthermore, we say two operations $o_1, o_2 \in \mathcal{O}$ *commute* when they commute in any state of $\mathcal{S}$.

Also sequences of operations can commute. Suppose two sequences ρ_1 and ρ_2 consisting of operations in $\mathcal{O}$ are mixed together into one sequence π such that the partial order among the operations from ρ_1 and from ρ_2 is retained in π, respectively. If executing π starting from a state s gives the same respective responses and the same final state as for every other such mixed sequence, in particular for $\rho_1 \circ \rho_2$ and for $\rho_2 \circ \rho_1$, where $\circ$ denotes concatenation, we say that ρ_1 and ρ_2 *commute in state s*. Analogously, we say that ρ_1 and ρ_2 *commute* if they commute in any state.

Operations that do not commute are said to *conflict*. Commuting operations have been investigated by Weihl [18] in the context of concurrency control. We define a Boolean predicate *commute*$_F(s, \rho_1, \rho_2)$ that is true if and only if ρ_1 and ρ_2 commute in s according to F.

Abortable services. When operations of F conflict, a protocol may either decide to block or to abort. Aborting and giving the client a chance to retry the operation at his own rate has often advantages compared to blocking, which might delay an application in unexpected ways.

As in previous work [14], we permit operations to abort and augment F to a functionality F' accordingly. F' is defined over the same set of states $\mathcal{S}$ and operations $\mathcal{O}$ as F, but returns a tuple defined over $\mathcal{S}$ and $\mathcal{R} \cup \{\bot\}$. F' may return the same output as F, but F' may also return $\bot$ and leave the state unchanged, denoting that a client is not able to execute F. Hence, F' is a non-deterministic relation and satisfies

$$F'(s, o) = \big\{(s, \bot),\ F(s, o)\big\}.$$

Since F' is not deterministic, a sequence of operations no longer uniquely determines the resulting state and response value.

Operations and histories. The clients interact with F through *operations* provided by F. As operations take time, they are represented by two events occurring at the client, an *invocation* and a *response*. A *history* of an execution σ consists of the sequence of invocations and responses of F occurring in σ. An operation is *complete* in a history if it has a matching response.

An operation o *precedes* another operation o' in a sequence of events σ, denoted $o <_\sigma o'$, whenever o completes before o' is invoked in σ. A sequence of events π *preserves the real-time order* of a history σ if for every two operations o and o' in π, if $o <_\sigma o'$ then $o <_\pi o'$. Two operations are *concurrent* if neither one of them precedes the other. A sequence of events is *sequential* if it does not contain concurrent operations. For a sequence of events σ, the subsequence of σ consisting only of events occurring at client C_i is denoted by $\sigma|_{C_i}$ (we use the symbol $|$ as a projection operator). For some operation o, the prefix of σ that ends with the last event of o is denoted by $\sigma|^o$.

An operation o is said to be *contained in* a sequence of events σ, denoted $o \in \sigma$, whenever at least one event of o is in σ. We often simplify the terminology by exploiting that every *sequential* sequence of events corresponds naturally to a sequence of operations, and that analogously every sequence of operations corresponds to a sequential sequence of events.

An execution is *well-formed* if the events at each client are alternating invocations and matching responses, starting with an invocation. An execution is *fair*, informally, if it does not halt prematurely when there are still steps to be taken or messages to be delivered (see the standard literature for a formal definition [12]). We are interested in a protocol where the clients never block each other. Assuming the server is correct, then every operation of a client should complete independently of the other clients, and only through steps of the client and the server. We call such a protocol *wait-free*.

Consistency properties. Clients interact with F via operations. Recall that every operation at a client C_i is associated with an invocation and a response event that occurs at C_i. We say that C_i *executes* an operation between the corresponding invocation and completion events.

Definition 1 (View). A sequence of events π is called a *view* of a history σ at a client C_i w.r.t. a functionality F if:

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

Definition 2 (Linearizability [9]). A history σ is linearizable w.r.t. a functionality F if there exists a sequence of events π such that:

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

Definition 3 (Fork-linearizability [15]). A history σ is fork-linearizable w.r.t. a functionality F if for each client C_i there exists a sequence of events π_i such that:

1. π_i is a view of σ at C_i w.r.t. F;
2. π_i preserves real-time order of σ; and
3. for every client C_j and every operation $o \in \pi_i \cap \pi_j$ it holds that $\pi_i|^o = \pi_j|^o$.

Definition 4 (Fork-linearizable Byzantine emulation [4]). We say that a protocol P for a set of clients *emulates* a functionality F *on a Byzantine server* S *with fork-linearizability* if and only if in every fair and well-formed execution of P, the sequence of events observed by the clients is fork-linearizable with respect to F, and moreover, if S is correct, then the execution is linearizable w.r.t. F.

Cryptography. In this paper, we make use of several cryptographic primitives, namely hash functions and digital signatures. A hash function *hash* maps a bit string x of arbitrary length to a short, unique representation of fixed length. We use a collision-free hash function; this property ensures that it is computationally infeasible to produce two different inputs x and x' such that $\mathit{hash}(x) = \mathit{hash}(x')$. A digital signature scheme provides two operations, *sign* and *verify*. We parametrize these operations for each client C_i as sign_i and verify_i. The invocation of sign_i takes a bit string m as a parameter and returns a signature ϕ with the response. The verify_i operation takes a string m, and a putative signature ϕ as parameters and returns a Boolean value. It satisfies that $\mathit{verify}_i(m, \phi)$ is true for all i and m if and only if C_i has executed $\mathit{sign}_i(m) = \phi$ before. Only C_i may invoke $\mathit{sign}_i(\cdot)$, but every client and S may invoke verify_i.

3 The commutative-operation verification protocol

The pseudocode of COP for the clients and the server is presented in Figures 1–3. We assume that the execution of each client is well-formed and fair.

Notation. The function $\mathit{length}(a)$ for a list a denotes the number of elements in a. Several variables are *dynamic arrays* or *maps*, which associate keys to values. A value is stored in a map H by assigning it to a key, denoted $H[k] \leftarrow v$; if no value has been assigned to a key, the map returns $\perp$. Recall that F' is the abortable extension of functionality F.

Overview. COP adopts the structure of previous protocols that guarantee fork-linearizable semantics [15, 19, 1]. It aims at obtaining a globally consistent order for the operations of all clients, as determined by the server.

When a client C_i invokes an operation o, he sends a INVOKE message to the server S. He expects to receive a REPLY message from S telling him about the position of o in the global sequence of operations. The message contains the operations that are *pending* for o, that is, operations that C_i may not yet know and that are ordered before o by S. We distinguish between *pending-other* operations invoked by other clients and *pending-self* operations, which are operations executed by C_i up to o.

Client C_i then verifies that the data from the server is consistent. In order to ensure fork-linearizability for his response values, the client first simulates the pending-self operations and tests if *o commutes* with the pending-other operations. If the test succeeds, he declares o to be *successful*, executes o, and computes the response r according to F; otherwise, O is *aborted* and the response is $r = \perp$. According to this, the *status* of o is either SUCCESS or ABORT. Through these steps the client *commits o*. Then he sends a corresponding COMMIT message to S and outputs r.

The server records the committed operation and relays it to all clients via a BROADCAST message. When the client receives such a broadcasted operation, he verifies that it is consistent with everything the server told him so far. If this verification succeeds, we say that the client *confirms* the operation. If the operation's status was SUCCESS, then the client executes it and *applies* it to his local state.

Data structures. Every client locally maintains a set of variables during the protocol. The state $s \in \mathcal{S}$ is the result of applying all successful operations, received in BROADCAST messages, to the initial state s_0. Variable c stores the sequence number of the last operation that the client has confirmed. H is a map containing a *hash chain* computed over the global operation sequence as announced by S. The contents of H are indexed by the sequence number of the operations, such that entry $H[l]$ is computed as $\mathit{hash}(H[l-1]\|o\|l\|i)$ and represents operation o with sequence number l executed by C_i. A variable u is set to o whenever the client has invoked an operation o but not yet completed it; otherwise u is $\perp$. Variable Z maps the sequence number of every operation that the client has executed himself to the status of the operation.

The server also keeps several variables locally. She stores the invoked operations in a map I and the completed operations in a map O, both indexed by the operations' sequence numbers. Variable t determines the global sequence number for the invoked operations. Finally, variable b is the sequence number of the last broadcasted operation and ensures that S disseminates operations to clients in the global order.

Protocol. When client C_i invokes an operation o, he stores it in u and sends an INVOKE message to S containing o, c, and τ, a digital signature computed over o and i. In turn, a correct S sends a REPLY message with the list ω of pending operations; they have a sequence number greater than c. Upon

Algorithm 1 Commutative-operation verification protocol (client C_i)

```
State
    $u \in \mathcal{O} \cup \{\bot\}$: the operation being executed currently or $\bot$ if no operation runs, initially $\bot$
    $c \in \mathbb{N}_0$: sequence number of the last operation that has been confirmed, initially 0
    $H : \mathbb{N}_0 \to \{0,1\}^*$: hash chain (see text), initially containing only $H[0] = \text{NULL}$
    $Z : \mathbb{N}_0 \to \mathcal{Z}$: status map (see text), initially empty
    $s \in \mathcal{S}$: current state, after applying operations, initially $s_0$

upon invocation $o$ do
    $u \leftarrow o$
    $\tau \leftarrow \mathit{sign}_i(\text{INVOKE}\|o\|i)$
    send message $[\text{INVOKE}, o, c, \tau]$ to $S$

upon receiving message $[\text{REPLY}, \omega]$ from $S$ do
    $\gamma \leftarrow \langle\rangle$                                   // list of pending-other operations
    $\mu \leftarrow \langle\rangle$                                   // list of successful pending-self operations
    $k \leftarrow 1$
    while $k \leq \mathit{length}(\omega)$ do
        $(o, j, \tau) \leftarrow \omega[k]$
        $l \leftarrow c + k$                                   // promised sequence number of $o$
        if not $\mathit{verify}_j(\tau, \text{INVOKE}\|o\|j)$ then
            halt
        if $H[l] = \bot$ then
            if $H[l-1] = \bot$ then
                halt                                   // server replies are inconsistent
            $H[l] \leftarrow \mathit{hash}(H[l-1]\|o\|l\|j)$
        else if $H[l] \neq \mathit{hash}(H[l-1]\|o\|l\|j)$ then
            halt                                   // server replies are inconsistent
        if $j = i \wedge Z[l] = \text{SUCCESS}$ then
            $\mu \leftarrow \mu \circ \langle o \rangle$
        else if $j \neq i$ then
            $\gamma \leftarrow \gamma \circ \langle o \rangle$
        $k \leftarrow k + 1$
    if $k = 1 \vee o \neq u \vee j \neq i$ then
        halt                   // last pending operation must equal the current operation
    $(a, r) \leftarrow F(s, \mu)$          // compute temporary state with successful pending-self operations
    if $\mathit{commute}_F(a, \langle o \rangle, \gamma)$ then
        $(a, r) \leftarrow F(a, o)$
        $Z[l] \leftarrow \text{SUCCESS}$
    else
        $r \leftarrow \bot$
        $Z[l] \leftarrow \text{ABORT}$
    $\phi \leftarrow \mathit{sign}_i\big(\text{COMMIT}\|u\|l\|H[l]\|Z[l]\big)$
    send message $[\text{COMMIT}, u, l, H[l], Z[l], \phi]$ to $S$
    $u \leftarrow \bot$
    return $r$
```

Algorithm 2 Commutative-operation verification protocol (client C_i, continued)

```
upon receiving message $[\textsc{broadcast}, o, q, h, z, \phi, j]$ from $S$ do
    if not $\big(q = c + 1$ and $\mathit{verify}_j(\phi, \textsc{commit}\|o\|q\|h\|z)\big)$ then
        halt                                                      // server replies are not consistent
    if $H[q] = \bot$ then                                         // operation has not been pending at client
        $H[q] \leftarrow \mathit{hash}(H[q-1]\|o\|q\|j)$
    if $h \neq H[q]$ then
        halt                          // server replies are not consistent, operation is not confirmed
    if $z = \textsc{success}$ then
        $(s, r) \leftarrow F(s, o)$                               // apply the operation and ignore response
    $c \leftarrow c + 1$
```

Algorithm 3 Commutative-operation verification protocol (server S)

```
State
    $t \in \mathbb{N}_0$: sequence number of the last invoked operation, initially 0
    $b \in \mathbb{N}_0$: sequence number of the last broadcasted operation, initially 0
    $I : \mathbb{N} \rightarrow \mathcal{O} \times \mathbb{N}_0 \times \{0,1\}^*$: invoked operations (see text), initially empty
    $O : \mathbb{N} \rightarrow \mathcal{O} \times \{0,1\}^* \times \mathcal{Z} \times \{0,1\}^* \times \mathbb{N}$: committed operations (see text), initially empty

upon receiving message $[\textsc{invoke}, o, c, \tau]$ from $C_i$ do
    $t \leftarrow t + 1$
    $I[t] \leftarrow (o, i, \tau)$
    $\omega \leftarrow \langle I[c+1], I[c+2], \ldots, I[t] \rangle$              // include non-committed operations and $o$
    send message $[\textsc{reply}, \omega]$ to $C_i$

upon receiving message $[\textsc{commit}, o, q, h, z, \phi]$ from $C_i$ do
    $O[q] \leftarrow (o, h, z, \phi, i)$
    while $O[b+1] \neq \bot$ do                  // broadcast operations ordered by their sequence number
        $b \leftarrow b + 1$
        $(o, h', z', \phi', j) \leftarrow O[b]$
        send message $[\textsc{broadcast}, o, b, h', z', \phi', j]$ to all clients
```

receiving a REPLY message, the client checks that ω is consistent with any previously sent operations and uses ω to assemble the successful pending-self operations μ and the pending-other operations γ. He then determines whether o can be executed or has to be aborted.

In particular, during the loop in Algorithm 1, for every operation o in ω, C_i determines its sequence number l and verifies that o was indeed invoked by C_j from the digital signature. He computes the entry of o in the hash chain from o itself, l, j, and $H[l-1]$. If $H[l] = \bot$, then C_i stores the hash value there. Otherwise, if $H[l]$ has already been set, C_i verifies that the hash values are equal; this means that o is consistent with the pending operation(s) that S has sent previously with indices up to l.

If operation o is his own and its saved status in $Z[l]$ was SUCCESS, then he appends it to μ. The client remembers the status of his own operations in Z, since $\mathit{commute}_F$ depends on the state and that could have changed if he applied operations after committing o.

Finally, when C_i reaches the end of ω (i.e., when C_i considers $o = u$), he checks that ω is not empty and that it contains o at the last position. He then creates a temporary state a by applying μ to the current state s, and tests whether o commutes with the pending-other operations γ in a. If they do, he records the status of o as SUCCESS in $Z[l]$ and computes the response r by executing o on state a. If o does not commute with γ, he sets status of o to ABORT and $r \leftarrow \bot$. Then C_i signs o together with its sequence number, status, and hash chain entry $H[l]$ and includes all values in the COMMIT message sent to S.

Upon receiving a COMMIT message for an operation o with a sequence number q, the server records its content as $O[q]$ in the map of committed operations. Then she is supposed to send a BROADCAST message containing $O[q]$ to the clients. She waits with this until she has received COMMIT messages for all operations with sequence number less than q and broadcasted them. This ensures that completed operations are disseminated in the global order to all clients. Note that this does not forbid clients from progressing with their own operations as we explain below.

In a BROADCAST message received by client C_i, the committed operation is represented by a tuple (o, q, h, z, ϕ, j). He conducts several verification steps before confirming the operation o and applying it to his state s. First, he verifies that the sequence number q is the next operation according to his variable c, hence, o follows the global order and the server did not omit any operations. Second, he uses the digital signature ϕ on the information in the message to verify that the client C_j indeed committed o. Lastly, the client computes his own hash-chain entry $H[q]$ for o and confirms that it is equal to the hash-chain value h from the message. This ensures that C_i and C_j have received consistent operations from S up to o. Once the verification succeeds, the client applies o to his state s only if its status z was SUCCESS, that is, when C_j has not aborted o.

Memory requirements. For saving storage space, the client may garbage-collect entries of H and Z with sequence numbers smaller than c. The server can also save space by removing the entries in I and O for the operations that she has broadcast. However, if new clients are allowed to enter the protocol, the server should keep all operations in O and broadcast them to new clients upon their arrival.

With the above optimizations the client has to keep only pending operations in H and pending-self operations in Z. The same holds for the server: the maximum number of entries stored in I and O is proportional to the number of pending operations at any client.

Communication. Every operation executed by a client requires him to perform one roundtrip to the server: send an INVOKE message and receive a REPLY. For every executed operation the server simply sends a BROADCAST message. Clients do not communicate with each other in the protocol. However, as soon as they do, they benefit from fork-linearizability and can easily discover a forking attack by comparing their hash chains.

Messages INVOKE, COMMIT and BROADCAST are all of constant size, while the REPLY message contains the list of pending operations ω. If even one client is slow, then the length of ω for all other clients grows proportionally to the number of further operations they are executing. To reduce the size of REPLY messages, the client can remember all pending operations received from S, and S can send every pending operation only once.

Wait-freedom. Every client executing COP can proceed with an operation o for F as long as it does not conflict with pending operations of other clients. He outputs the response immediately after receiving the REPLY message from S. A conflict arises when o does not commute with the pending operations of other clients. In this case, the client aborts o and outputs $\perp$, according to F'.

It is important that the state used by the client for executing o reflects all of his own operations executed so far, even if he has not yet confirmed or applied them to his state because operations of other clients have not yet completed. Otherwise, the protocol might violate fork-linearizability. COP is wait-free because regardless of whether operation o aborts, the client may proceed executing further operations.

4 Analysis

Theorem 1. *The commutative-operation verification protocol in Figures 1–3 emulates functionality F' on a Byzantine server with fork-linearizability.*

We prove this theorem through a sequence of lemmas in the remainder of this section. We start by introducing additional notation.

When a client issues a COMMIT signature for some operation o, we say that he *commits o*. The client's sequence number included in the signature thus becomes the *sequence number of o*; note that with a faulty S, two different operations may be committed with the same sequence number by separate clients.

Lemma 2. *If the server is correct, then every history σ is linearizable w.r.t. F'. Moreover, if the clients execute all operations sequentially, then σ is linearizable w.r.t. F.*

Proof. Recall that σ consists of invocation and response events. We construct a sequential permutation π of σ in terms of the operations associated to the events in σ. Note that a client sends an INVOKE message with his operation to the server, the server assigns a sequence number to the operation and sends it back. The client then computes the response and sends a signed COMMIT message to S, containing the operation and its sequence number. Since each executed operation appears in σ in terms of its invocation and response events, π contains all operations of all clients.

We order π by the sequence number of the operations. If the server is correct she processes INVOKE messages in the order they are received and assigns sequence numbers accordingly. This implies that if an operation o' is invoked after an operation o completes, then the sequence number of o' is higher than o's. Hence, π preserves the real-time order of σ.

We now use induction on the operations in π to show that π satisfies the sequential specification of F'. Note that F' requires a bit of care, as it is not deterministic. For a sequence ω of operations of F' in an actual execution, we write *successful*(ω) for the subsequence whose status was SUCCESS; restricted to such operations, F' is deterministic. In particular, consider some operation $o \in \pi$, executed by client C_i. We want to show that C_i computes (s', r) such that $(s', r) \in F'(s_0, \textit{successful}(\pi|^o))$, whereby it outputs r after committing o and stores s' in its variable s after applying o.

Consider the base case where o is the first operation in π. Note that S has not reported any pending operations to C_i because o is the first operation. Thus, C_i determines that the status of o is SUCCESS, computes $(s', r) \leftarrow F(s_0, o)$ and outputs r. Hence, F' is satisfied. When C_i later receives o in the BROADCAST message from S with sequence number 1, the state is also updated correctly.

Now consider the case when o is not the first operation in π and assume that the induction assumption holds for an operation that appears in π before o. If the status of o is ABORT, then the client does not invoke F, returns $\perp$, and leaves the state unchanged upon applying o. The claim follows.

Otherwise, we need to show that the response $r \neq \perp$ and the state s' after applying o satisfy $(s', r) = F(s_0, \mathit{successful}(\pi|^o))$. Since S is correct, she assigns unique sequence numbers to the operations. We split the operations with a sequence number smaller than that of o in three groups: a sequence ρ of operations that C_i has confirmed before he committed o, this sequence is in the order in which C_i confirmed these operations; a sequence δ of operations of *other* clients that were reported by S as pending to C_i when executing o, ordered as in the REPLY message; and a sequence ν of operations that C_i has committed *itself* before o but not yet confirmed or applied, ordered by their sequence number.

Observe that C_i computes r starting from its own copy of the state $\bar{s}$ that results after applying all operations in $\mathit{successful}(\rho)$. From the induction assumption, it follows that $(\bar{s}, \cdot) = F(s_0, \mathit{successful}(\rho))$ because ρ is a prefix of π. From variable ω in the REPLY message, C_i computes the pending-other operations γ and the successful pending-self operations μ. Note that $\gamma = \delta$ and $\mu = \mathit{successful}(\nu)$ as the server is correct. The client computes a temporary state $(a, \cdot) = F(\bar{s}, \mu)$. Because o does not abort, C_i has determined that o commutes with γ in a and computed $(\cdot, r) = F(a, o)$. By the definition of commuting operation sequences, we have that $(s', r) = F(a, \mathit{successful}(\gamma) \circ o)$ and $(s', r) = F(\bar{s}, \mathit{successful}(\omega))$ since the order of operations in μ and γ is preserved in ω. Hence, $(s', r) = F(s_0, \mathit{successful}(\pi|^o))$.

The sequence π preserves the real-time order of σ and satisfies the three conditions of a view of σ at every client C_i w.r.t. F', hence, σ is linearizable w.r.t. F'.

The second part of the lemma claims that if clients execute operations sequentially, then no client outputs $\perp$. Since the sequence of events at every client is well-formed, a client does not invoke an operation before he has completed the previous one. Moreover, if clients execute operations sequentially then no client invokes an operation while there is a client who has not completed his operation. Hence, the server never includes any pending operations in ω of the REPLY message. The check for conflicts is never positive, and all operations have status SUCCESS. Hence, no client returns $\perp$ and σ satisfies the sequential specification of F. □

The promised view of an operation. Suppose a client C_i executes and thereby commits an operation o. We define the *promised view to C_i of o* as the sequence of all operations that C_i has confirmed before committing o, concatenated with the sequence ω of pending operations received in the REPLY message during the execution of o, including o itself (according to the protocol C_i verifies that the last operation in ω is o).

Lemma 3. *If C_j has confirmed some operation o that was committed by a client C_i, then the sequence of operations that C_j has confirmed up to (and including) o is equal to the promised view to C_i of o In particular,*

1. *if C_i and C_j have confirmed an operation o, then they have both confirmed the same sequence of operations up to o; and*
2. *the promised view to C_i of o contains all operations executed by C_i up to o.*

Proof. Note that every client computes a hash chain H in which every defined entry contains a hash value that represents a sequence of operations. More precisely, if C_i commits o with sequence number l, then he has set $H[l] \leftarrow \mathit{hash}(H[l-1] \| o \| l \| i)$; this step recursively defines the sequence represented by $H[l]$ as the sequence represented by $H[l-1]$ followed by o. According to the collision-resistance of the hash function, no two different operation sequences are represented by the same hash value. Note that no client ever overwrites an entry of H; moreover, if a client arrives at a point in the protocol where he might assign some value h to entry $H[l]$ but $H[l] \neq \bot$, then he verifies that $H[l] = h$ and aborts if this fails.

Consider the moment when C_i receives the REPLY message during the execution of o. The view of o promised to C_i contains the sequence of operations that C_i has confirmed, followed by the list ω in the REPLY message, including o.

For every pending operation $p \in \omega$, client C_i checks if he has already an entry in H at index l, which is the promised sequence number of p to C_i according to ω. If there is no such entry, he computes the hash value $H[l]$ as above. Otherwise, C_i must have received an operation for sequence number l earlier, and so he verifies that o is the same pending operation as received before. Moreover, C_i verifies that his last invoked operation is also returned to him as pending and adds it to H. Hence, the new hash value h stored in H at the sequence number of o represents the promised view to C_i of o.

Subsequently, C_i signs o and h together and sends it to the server. Client C_j receives it in a BROADCAST message from S, to be confirmed and applied with sequence number q. Because C_j verifies the signature of C_i on o, q, and h, the hash value h received by C_j represents the promised view to C_i of o. Before C_j applies o as his q-th operation, according to the protocol he must have already confirmed $q - 1$ operations one by one. Client C_j also verifies that he has either already computed the same $H[q] = h$ or he computes $H[q]$ from his value $H[q - 1]$ and checks $H[q] = h$. As $H[q]$ represents the sequence of operations that C_j has confirmed up to o, from the collision resistance of the hash function, this establishes the main statement of the lemma.

The first additional claim follows simply by noticing that the statement of the lemma holds for $i = j$. For showing the second additional claim, we note that if C_i confirms an operation of himself, then he has previously executed it (successful or not). There may be additional operations that C_i has executed but not yet confirmed, but C_i has verified according to the above argument that these were all contained in ω from the REPLY message. Thus they are also in the promised view of o. □

The view of a client. We construct a sequence π_i from σ as follows. Let o be the operation committed by C_i which has the highest sequence number among those operations of C_i that have been confirmed by some client C_k (including C_i). Define α_i to be the sequence of operations confirmed by C_k up to and including o. Furthermore, let β_i be the sequence of operations committed by C_i with a sequence number higher than that of o. Then π_i is the concatenation of α_i and β_i. Observe that by definition, no client has confirmed operations from β_i.

Lemma 4. *The sequence π_i is a view of σ at C_i w.r.t. F'.*

Proof. Note that π_i is defined through a sequence of operations that are contained in σ. Hence π_i is sequential by construction.

We now argue that all operations executed by C_i are included in π_i. Recall that $\pi_i = \alpha_i \circ \beta_i$ and consider o, the last operation in α_i. As o has been confirmed by C_k, Lemma 3 shows that α_i is equal to the promised view to C_i of o and, furthermore, that it contains all operations that C_i has executed up to o. By construction of π_i all other operations executed by C_i are contained in β_i, and the property follows.

The last property of a view requires that π_i satisfies the sequential specification of F'. Note that F' is not deterministic and some responses might be $\bot$. But when we ensure that two operation sequences of F' have responses equal to $\bot$ in exactly the same positions, then we can conclude that two equal operation sequences give the same resulting state and responses from the fact that F is deterministic.

We first address the operations in α_i. Consider again o, the last operation in α_i, which has been confirmed by C_k. For the point in time when C_i executes o, define ρ to be the sequence of operations that

C_i has confirmed prior to this and define $\bar{s}$ as the resulting state from applying the successful operations in ρ, as stored in variable s; furthermore, let ω be the pending operations contained in the REPLY message from S. Observe that ω can be partitioned in the pending-other operations γ, the successful pending-self operations of C_i as stored in μ, the aborted pending-self operations of C_i, and o. Client C_i computes the response r for o in state a that results from $F(s, \mu)$. Before executing o, C_i verifies that o commutes with γ in a. Note that when C_i committed some operation $p \in \mu$ he has also verified that p commuted with the pending-other operations in $\omega|^p$. Hence, the response resulting from executing the operations in ρ followed by $\mu \circ o$ is the same as that resulting from executing $\mu \circ \mathit{successful}(\gamma) \circ o$ after ρ (recall the notation $\mathit{successful}(\cdot)$ from Lemma 2). Since ω preserves the order of operations in μ and γ, the response is also the same after the execution of $\rho \circ \mathit{successful}(\omega)$. Moreover, the state resulting from executing the operations in ρ followed by $\mu \circ \mathit{successful}(\gamma) \circ o$ is the same as that resulting from executing $\rho \circ \mathit{successful}(\omega)$. Since $\rho \circ \omega$ is the promised view to C_i of o, and since C_k has confirmed o, Lemma 3 now implies that $\rho \circ \omega$ is equal to α_i.

To conclude the argument, we only have to show that the abort status for all operations in the sequences is the same. Then they will produce the same responses and the same final state. Note that when C_i executes some operation o he either computes a response according to F or aborts the operation, declaring its status to be SUCCESS or ABORT, respectively. For operations in ρ this is clear from the protocol as the status is included in the BROADCAST message. And whenever C_i later obtains o again as a pending-self operation in ω at some index l, he verifies that it is the same operation as previously at index l and applies or skips it as before according to the status remembered in $Z[l]$. Hence, the responses of C_i from executing the operations in α_i respect the specification of F'.

The remainder of π_i consists of β_i, whose operations C_i executes himself using F'. Hence, π_i satisfies the sequential specification of F'. □

Lemma 5. *If some client C_k confirms an operation o_1 before an operation o_2, then o_2 does not precede o_1 in the execution history σ.*

Proof. Let δ_k denote the sequence of operations that C_k has confirmed up to o_2. According to the protocol logic, δ_k contains o_1, and o_1 has a smaller sequence number than o_2. Lemma 3 shows that δ_k is equal to the promised view to C_k of o_2, hence, o_1 is in the promised view to C_k of o_2. Recall that the promised view contains operations that have been committed or are pending for other clients. Hence, o_1 has been invoked before o_2 completed. □

Lemma 6. *The sequence π_i preserves the real-time order of σ.*

Proof. Recall that $\pi_i = \alpha_i \circ \beta_i$ and consider first those operations of π_i that appear in α_i, that is, they have been confirmed by some client C_k. Lemma 5 shows that these operations preserve the real-time order of σ. Second, the operations in β_i are ordered according to their sequence number and they were committed by C_i. According to the protocol, C_i executes only one operation at a time and always assigns a sequence number that is higher than the previous one. Hence, β_i also preserves the real-time order of σ.

We are left to show that no operation in β_i precedes an operation from α_i in σ. Recall that α_i is the promised view to C_i of o (the last operation in α_i) and includes the operations that C_i has confirmed or received as pending from S after C_i invoked o. Since o precedes all operations from β_i, it follows that no operation in α_i precedes an operation from β_i. □

Lemma 7. *If $o \in \pi_i \cap \pi_j$ then $\pi_i|^o = \pi_j|^o$.*

Proof. As $\pi_i = \alpha_i \circ \beta_i$ and $\pi_j = \alpha_j \circ \beta_j$, we need to consider four cases to analyze all operations that can appear in $\pi_i \cap \pi_j$ and the rest are symmetrical.

1. $o \in \alpha_i$ and $o \in \alpha_j$: This case happens when (a) C_i and C_j both confirmed o, or when (b) C_i has confirmed an operation of C_j or vice versa, or when (c) a client C_k has confirmed operations of C_i and C_j. For (a) and (b) Lemma 3 shows that $\alpha_i|^o = \alpha_j|^o$. In case (c) neither C_i nor C_j has confirmed o, but o is in their views because C_k has confirmed pending operations of C_i and C_j. Hence, $\pi_k|^o = \alpha_i|^o$ and $\pi_k|^o = \alpha_j|^o$ again from Lemma 3.
2. $o \in \beta_i$ and $o \in \alpha_j$: This case cannot happen, since no client has confirmed operations from β_i by definition.
3. $o \in \alpha_i$ and $o \in \beta_j$: Analogous to the case above.
4. $o \in \beta_i$ and $o \in \beta_j$: This case cannot happen since β_i and β_j contain only pending-self operations of C_i and C_j, correspondingly.

□

5 Conclusion

This paper has introduced the Commutative-Operation verification Protocol (COP), which allows a group of clients to execute a generic service coordinated by a remote untrusted server. COP ensures fork-linearizability and allows clients to easily verify the consistency and integrity of the service responses. In contrast to previous work, COP is wait-free and supports commuting operation sequences, but may sometimes abort conflicting operations.

Given the popularity of outsourced computation and the cloud-computing model, the problem of checking the integrity of remote computations has received a lot of attention recently [7, 5, 16]. But such cryptographic protocols typically address only a two-party model with a single client. Combining them with COP or other protocols that guarantee fork-linearizability will represent an important step toward a comprehensive consistency-verification solutions for realistic distributed systems.

Acknowledgments

This work has been supported in part by the European Union's Seventh Framework Programme (FP7/2007–2013) under grant agreement number ICT-257243 TCLOUDS, and by the United States National Science Foundation under grant CNS-1012060.

References

[1] C. Cachin, "Integrity and consistency for untrusted services," in *Proc. 37th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011)* (I. Cerná *et al.*, eds.), vol. 6543 of *Lecture Notes in Computer Science*, pp. 1–14, Springer, 2011.

[2] C. Cachin, I. Keidar, and A. Shraer, "Fork sequential consistency is blocking," *Information Processing Letters*, vol. 109, pp. 360–364, Mar. 2009.

[3] C. Cachin, I. Keidar, and A. Shraer, "Fail-aware untrusted storage," *SIAM Journal on Computing*, vol. 40, pp. 493–533, Apr. 2011. Preliminary version appears in *Proc. DSN 2009*.

[4] C. Cachin, A. Shelat, and A. Shraer, "Efficient fork-linearizable access to untrusted shared memory," in *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 129–138, 2007.

[5] G. Cormode, M. Mitzenmacher, and J. Thaler, "Practical verified computation with streaming interactive proofs," in *Proc. 3rd Conference on Innovations in Theoretical Computer Science (ITCS)*, pp. 90–112, 2012.

[6] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *Proc. 9th Symp. Operating Systems Design and Implementation (OSDI)*, 2010.

[7] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology: CRYPTO 2010* (T. Rabin, ed.), vol. 6223 of *Lecture Notes in Computer Science*, pp. 465–482, Springer, 2010.

[8] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter, "Zzyzx: Scalable fault tolerance through byzantine locking," in *Proc. 40th International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2010.

[9] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.

[10] J. Li, M. Krohn, D. Mazires, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, pp. 121–136, 2004.

[11] J. Li and D. Mazières, "Beyond one-third faulty replicas in Byzantine fault-tolerant systems," in *Proc. 4th Symp. Networked Systems Design and Implementation (NSDI)*, 2007.

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

[13] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *Proc. 9th Symp. Operating Systems Design and Implementation (OSDI)*, 2010.

[14] M. Majuntke, D. Dobre, M. Serafini, and N. Suri, "Abortable fork-linearizable storage," in *Proc. 13th Conference on Principles of Distributed Systems (OPODIS)* (T. F. Abdelzaher, M. Raynal, and N. Santoro, eds.), vol. 5923 of *Lecture Notes in Computer Science*, pp. 255–269, Springer, 2009.

[15] D. Mazières and D. Shasha, "Building secure file systems out of Byzantine storage," in *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[16] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, , and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *Proc. 21st USENIX Security Symposium*, 2012.

[17] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *Proc. Cloud Computing Security Workshop (CCSW)*, ACM, 2010.

[18] W. E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Transactions on Computers*, vol. 37, pp. 1488–1505, Dec. 1988.

[19] P. Williams, R. Sion, and D. Shasha, "The blind stone tablet: Outsourcing durability to untrusted parties," in *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2009.