SPORC: Group Collaboration using Untrusted Cloud Resources

Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten
Princeton University

Abstract

Cloud-based services are an attractive deployment model for user-facing applications like word processing and calendaring. Unlike desktop applications, cloud services allow multiple users to edit shared state concurrently and in real-time, while being scalable, highly available, and globally accessible. Unfortunately, these benefits come at the cost of fully trusting cloud providers with potentially sensitive and important data.

To overcome this strict tradeoff, we present SPORC, a generic framework for building a wide variety of collaborative applications with *untrusted* servers. In SPORC, a server observes only encrypted data and cannot deviate from correct execution without being detected. SPORC allows concurrent, low-latency editing of shared state, permits disconnected operation, and supports dynamic access control even in the presence of concurrency. We demonstrate SPORC's flexibility through two prototype applications: a causally-consistent key-value store and a browser-based collaborative text editor.

Conceptually, SPORC illustrates the complementary benefits of *operational transformation* (OT) and *fork* consistency*. The former allows SPORC clients to execute concurrent operations without locking and to resolve any resulting conflicts automatically. The latter prevents a misbehaving server from equivocating about the order of operations unless it is willing to *fork* clients into disjoint sets. Notably, unlike previous systems, SPORC can automatically recover from such malicious forks by leveraging OT's conflict resolution mechanism.

1 Introduction

An emerging class of cloud-based collaborative services, such as online document processing and calendaring, provides users with anywhere-available, real-time, and concurrent access to shared state. Their deployments on managed cloud platforms enjoy global accessibility, high availability, fault tolerance, and elastic resource allocation and scaling. Yet these benefits have come at the cost of having a fully trusted server, creating a risk of privacy problems due to server-side information leaks. The history of such services is one rife with unplanned data disclosures and malicious break-ins [24]. Indeed, the very centralization of information makes cloud providers high value targets for attack. Further, the behavior of service providers themselves is a source of users' privacy angst, as privacy policies may be weakened due to market expediencies. Finally, cloud providers face pressure from government agencies world-wide to release information on demand [15].

This paper challenges the belief that applications must sacrifice strong security and privacy to enjoy the benefits of cloud deployment. We present a system, SPORC, that offers managed cloud-based deployment for group collaboration services, yet does require users to trust the cloud provider to maintain data privacy or even to operate correctly. SPORC's cloud servers see only encrypted data, and clients will detect any deviation from correct operation (*e.g.*, adding, modifying, dropping, or reordering operations) and will recover from the error. Much like SUNDR [24], SPORC bases its security and privacy guarantees on the security of users' cryptographic keys, and not on the cloud provider's good intentions nor on some threshold-like protocol between servers [9] that is susceptible to administrative or software attacks.

SPORC provides a *generic* collaboration service in which users can create a document, modify its access control list, edit it concurrently, experience fully automated merging of updates, and even perform these operations while disconnected. The SPORC framework supports a broad range of collaborative applications. Data updates are encrypted before being sent to a cloud-hosted server. The server assigns a total order to all operations and redistributes the ordered updates to clients. If a malicious server drops or reorders updates, the SPORC clients can detect the server's misbehavior, switch to a new server, restore a consistent state, and continue. The same mechanism that allows SPORC to merge correct concurrent operations also enables it to transparently recover from attacks that fork clients' views.

From a conceptual distributed systems perspective, SPORC demonstrates the benefit of combining *operational transformation* [11] and *fork* consistency* protocols [23]. Operational transformation (OT) defines a framework for executing lock-free concurrent operations that both preserves causal consistency and converges to a common shared state. It does so by transforming operations so they can be applied commutatively by different clients, resulting in the same final state. While OT originated with decentralized applications using pairwise reconciliation [11, 18], recent systems like Google Wave [44] have used OT with a trusted central server that orders and transforms clients' operations. Fork* consistency, on the

other hand, was introduced as a consistency model for interacting with an untrusted server: If the server causes the views of two clients to diverge, the clients must either never see each others' subsequent updates or else identify the server as faulty.

Recovering from a malicious fork is similar to reconciling concurrent operations in the OT framework. Upon detecting a fork, SPORC clients use OT mechanisms to replay and transform forked operations, restoring a consistent state. Previous applications of fork* consistency [23] could only detect forks, but not resolve them.

This paper makes the following contributions:

§2 We identify and explore the conceptual connection between operational transformation protocols and the fork* consistency model, and use this connection to motivate SPORC's design.

§3 We describe SPORC's framework and protocols for real-time collaboration. SPORC provides security and privacy against both an untrusted server that mediates communication and other clients that lack access control permissions.

§4 We demonstrate how to support dynamic access control, which is challenging because SPORC supports concurrent operations and offline editing.

§5 We describe how clients can detect and recover from maliciously-instigated forks. We also present a checkpoint mechanism that reduces saved client state and minimizes the join overhead for new clients.

§6 We illustrate the extensibility of SPORC's pluggable data model by building both a key-value store and a browser-based collaborative text editor. We implement these services as both stand-alone applications and web services; the latter run in a browser, execute in JavaScript (compiled from Java via GWT [12]), and require no prior installation.

We evaluate SPORC's performance in Section 7 before discussing related work and concluding.

2 System Model

The purpose of SPORC is to allow a group of users who trust each other to collaboratively edit some shared state, which we call the *document*, with the help of an untrusted server. SPORC is comprised of a set of client devices that modify the document on behalf of particular users, and a potentially-malicious server whose main role is to impose a global order on those modifications. The server receives updates from individual clients, orders them, and then broadcasts them to the other clients. Access to the document is limited to a set of authorized users, but each user may be logged into arbitrarily many clients simultaneously (*e.g.*, her desktop, laptop, and mobile phone). Each client, even if it is controlled by the same user as another client, has its own local view of the document that must be synchronized with all other clients.

2.1 Goals

We designed SPORC with the following goals in mind:

Flexible framework for a broad class of collaborative services. Because SPORC uses an untrusted server which does not see application-level content, the server is generic and can handle a broad class of applications. On the client side, SPORC provides a library suitable for use by a range of desktop and web-based applications.

Propagate modifications quickly. When a client is connected to the network, its changes to the shared state should propagate quickly to all other clients so that clients' views are nearly identical. This property makes SPORC suitable for building collaborative applications requiring nearly real-time updates, such as collaborative text editing and instant messaging.

Tolerate slow or disconnected networks. To allow clients to edit the document while offline or while experiencing high network latency, clients in SPORC update the document *optimistically*. Every time a client generates a modification, the client applies it immediately to its local state, and only later sends it to the server for redistribution. As a result, clients' local views of the document will invariably diverge, and SPORC must be able to resolve these divergences automatically.

Keep data confidential from the server and unauthorized users. Since the server is untrusted, document updates must be encrypted before being sent to the server. For efficiency, the system should use symmetric-key encryption. SPORC must provide a way to distribute this symmetric key to every client of authorized users. When a document's access control list changes, SPORC must ensure that newly added users can decrypt the entire document, and that removed users cannot decrypt any updates subsequent to their expulsion.

Detect a misbehaving server. Even without access to document plaintext, a malicious server could still do significant damage by deviating from its assigned role. It could attempt to add, drop, alter, or delay clients' (encrypted) updates, or it could show different clients inconsistent views of the document. SPORC must give clients a means to quickly detect these kinds of misbehavior.

Recover from malicious server behavior. If clients detect that the server is misbehaving, clients should be able to failover to a new server and resume execution. Since a malicious server could cause clients to have inconsistent local state, SPORC must provide a mechanism for automatically resolving these inconsistencies.

To achieve these goals, SPORC builds on two conceptual frameworks: *operational transformation* and *fork* consistency*.

2.2 Operational Transformation

Operational Transformation (OT) [11] provides a general model for synchronizing shared state, while allowing each client to apply local updates optimistically. In OT, the application defines a set of *operations* from which all modifications to the document are constructed. When clients generate new operations, they apply them locally before sending them to others. To deal with the conflicts that these optimistic updates inevitably incur, each client *transforms* the operations it receives from others before applying them to its local state. If all clients transform incoming operations appropriately, OT guarantees that they will eventually converge to a consistent state.

Central to OT is an application-specific *transformation function* $T(\cdot)$ that allows two clients whose states have diverged by a single pair of conflicting operations to return to a consistent, reasonable state. $T(op_1, op_2)$ takes two conflicting operations as input and returns a pair of transformed operations (op_1', op_2'), such that if the party that initially did op_1 now applies op_2', and the party that did op_2 now applies op_1', the conflict will be resolved.

To use the example from Nichols *et al.* [30], suppose Alice and Bob both begin with the same local state "ABCDE", and then Alice applies op_1 = 'del 4' locally to get "ABCE", while Bob performs op_2 = 'del 2' to get "ACDE". If Alice and Bob exchanged operations and executed each others' naively, then they would end up in inconsistent states (Alice would get "ACE" and Bob "ACD"). To avoid this problem, the application supplies the following transformation function that adjusts the offsets of concurrent delete operations:

$$T(\text{del } x, \text{del } y) = \begin{cases} (\text{del } x-1, \text{del } y) & \text{if} \quad x > y \\ (\text{del } x, \text{del } y-1) & \text{if} \quad x < y \\ (\text{no-op}, \text{no-op}) & \text{if} \quad x = y \end{cases}$$

Thus, after computing $T(op_1, op_2)$, Alice will apply op_2' ='del 2' as before but Bob will apply op_1' = 'del 3', leaving both in the consistent state "ACE".

Given this pair-wise transformation function, clients that diverge in arbitrarily many operations can return to a consistent state by applying the transformation function repeatedly. For example, suppose that Alice has optimistically applied op_1 and op_2 to her local state, but has yet to send them to other clients. If she receives a new operation op_{new}, Alice must transform it with respect to both op_1 and op_2: She first computes $(op_{new}', op_1') \leftarrow T(op_{new}, op_1)$, and then $(op_{new}'', op_2') \leftarrow T(op_{new}', op_2)$. This process yields op_{new}'', an operation that Alice has "transformed past" her two local operations and can now apply to her local state.

Throughout this paper, we use the notation $op' \leftarrow T(op, \langle op_1, \ldots, op_n \rangle)$ to denote transforming op past a sequence of operations $\langle op_1, \ldots, op_n \rangle$ by iteratively applying the transformation function.[1] Similarly, we define $\langle op_1', \ldots, op_n' \rangle \leftarrow T(\langle op_1, \ldots, op_n \rangle, op)$ to represent transforming a sequence of operations past a single operation.

Operational transformation can be applied in a wide variety of settings, as operations, and the transforms on them, can be tailored to each application's requirements. For a collaborative text editor, operations may contain inserts and deletes of character ranges at specific cursor offsets, while for a causally-consistent key-value store, operations may contain lists of keys to update or remove. In fact, we have implemented both such systems on top of SPORC, which we describe further in Section 6.

For many applications, with a carefully-chosen transformation function, OT is able to automatically return divergent clients to a state that is not only consistent, but semantically reasonable as well. But for some applications, such as source-code version control, semantic conflicts must be resolved manually. OT can support such applications through the choice of a transformation function that does not try to resolve the conflict, but instead inserts an explicit conflict marker into the history of operations. A human can later examine the marker and resolve the conflict by issuing new writes. These write operations will supercede the conflicting operations, provided that the system preserves the global order of committed operations and the partial order of each client's operations. Section 3 describes how SPORC provides these properties.

While OT was originally proposed for decentralized n-way synchronization between clients, many prominent OT implementations are server-centric, including Jupiter [30] and Google Wave [44]. They rely on the server to resolve conflicts and to maintain consistency, and are architecturally better suited for web services. On the flip side, a misbehaving server can compromise the confidentiality, integrity, and consistency of the shared state.

Later, we describe how SPORC adapts these server-based OT architectures to provide security against a misbehaving server. At a high level, SPORC has each client *simulate* the transformations that would have been applied by a trusted OT server, using the server only for ordering. But we still need to protect against inconsistent orderings, for which we leverage fork* consistency techniques [23].

2.3 Fork* Consistency

To prevent a malicious server from forging or modifying clients' operations, clients in SPORC digitally sign all their operations with their user's private key. This is not sufficient for correctness, however: a misbehaving server could still equivocate and present different clients with divergent views of the history of operations.

[1] Strictly speaking, T always returns a pair of operations. For simplicity, however, we sometimes write T as returning a single operation, especially when the other is unchanged, as in our "delete char" example.

To defend against server equivocation, SPORC clients enforce *fork* consistency* [23].[2] In fork*-consistent systems, clients share information about their individual views of the history by embedding it in every operation they send. As a result, if clients to whom the server has equivocated ever communicate, they will discover the server's misbehavior. The server can still divide its clients into disjoint groups and only tell each client about operations by others in its group. But, once the server has *forked* two groups in this way, it cannot tell a member of one group about an operation submitted by another group's members without risking detection.

As in BFT2F [23], each SPORC client enforces fork* consistency by maintaining a *hash chain* over its view of the committed history. In this context, a hash chain is a method of incrementally computing the hash of a list of elements. More specifically, if $op_1, \ldots, op_n$ are the operations in the history, h_0 is a constant initial value, and h_i is the value of the hash chain over the history up to op_i, then $h_i = H(h_{i-1}||H(op_i))$, where $H(\cdot)$ is a cryptographic hash function and $||$ denotes concatenation. When a client with history up to op_n submits a new operation, it includes h_n in its message. On receiving the operation, another client can check whether the included h_n matches its own hash chain computation over its local history up to op_n. If they do not match, the client knows that the server has equivocated.

2.4 The Benefits of Having a Server

SPORC uses a central untrusted server, but the server's sole purpose is to order and store client-generated operations. This limited role may lead one to ask whether the server should be removed, leading to a completely peer-to-peer design. Indeed, many group collaboration systems, such as Bayou [43] and Network Text Editor [17], employ decentralized architectures. Decentralized designs are a poor fit, however, for applications in which a user needs a timely notification that her operation has been committed and will not be overridden by another's (not yet received) operation. For example, to schedule a meeting room, an online user should be able to quickly determine whether her reservation succeeded, without worrying if an offline client's request will override hers. Yet this is difficult to achieve without waiting to hear from all (or at least a quorum of) other clients, which poses a problem when clients are regularly offline. In reaction, Bayou delegates commits to a (statically) designated, trusted "primary" peer, which is little different from having a server.

SPORC, on the other hand, only requires an *untrusted* server for globally ordering operations. Thus, it can leverage the benefits of a cloud deployment—high availability and global accessibility—to achieve timely commits. We show in Section 4.2 how SPORC's centralized server also helps support dynamic access control and key rotation, even in the face of concurrent membership changes.

2.5 Deployment and Threat Model

Deployment Assumptions. While most of the paper discusses the SPORC protocol in terms of a single server and a single document, we assume that a cloud-based SPORC deployment would manage large numbers of users and documents by replicating functionality and partitioning state over many servers. Each document in SPORC can be managed independently, leading naturally to the shared-nothing architectures [36] already common to scalable cloud services.

For a client to recover from a misbehaving server, we assume there exists some alternative (untrusted) server to switch to after a client detects faulty behavior. These backup servers may belong to the same or different administrative domains as the original, depending upon the type of faults that a SPORC deployment expects to encounter.

Note that even if malicious (Byzantine) behavior among cloud servers is not a primary concern, this strong threat model also covers weaker non-crash failures related to server misconfiguration, Heisenbugs, or "split-brain" partitioned behavior. In all cases, failover and recovery is client driven. Crash failures, unlike Byzantine failures, would not result in forks and could be handled by traditional fault-tolerance techniques (*e.g.*, primary/backup replication) already employed in cloud services.

Threat Model. SPORC makes the following security assumptions:

Server: The server is potentially malicious, and a misbehaving server may be able to prevent progress, but it must not be able to corrupt the clients' shared state. A server may fork clients' states, but only within the confines of the fork* consistency model. If clients are able to communicate either in-band or out-of-band, server equivocation will be detected promptly by at least one client.

The server may be able to learn which users and clients are sharing a document, but it must not learn what is in the document or even the contents of the individual operations that the clients submit. Since the server has access to the size and timing of clients' operations, it may be able to glean some information about the document via traffic analysis. Traffic analysis is made more difficult by the fact that encrypted operations do not even reveal which portions of the shared state they modify. Nevertheless, the complete mitigation of traffic analysis is beyond the scope

[2] Fork* consistency is a weaker variant of an earlier model called *fork consistency* [27]. They differ in that under fork consistency, a pair of clients only needs to exchange one message to detect server equivocation, whereas under fork* consistency, they may need to exchange two. For OT systems like ours, this distinction makes little difference because clients constantly exchange small messages. On the other hand, fork* consistency permits a one-round protocol to submit operations, rather than two. Beyond efficiency, this also ensures that a crashed client cannot prevent the system from making progress.

of this work, but it would likely involve padding the length of operations and introducing cover traffic.

To attack availability, the server may arbitrarily erase or refuse to return any of the encrypted data that it stores. To mitigate this threat, the encrypted data could be replicated on servers in other administrative domains. Moreover, each client could replicate its own local state on cloud servers other than the main SPORC server. Notably, SPORC cannot guarantee recovery from every possible fork, unless every client stores every operation that it has seen either locally or remotely.

Clients: If a client is logged in as a particular user, that client is trusted to exercise the privileges granted to that user (*e.g.*, to see the state, modify it, or modify access privileges). Otherwise, clients are untrusted, and they should not be able to see the document, or to modify the document or its access control list, even if they collude with each other or with the server.

User authentication and keys: We assume that each user has a secure public/private key pair, and that clients have a secure way to verify the public key of other users.

Application code: We assume the presence of a code authentication infrastructure that can verify that the application code run by clients is genuine. This mechanism might rely on code signing or on HTTPS connections to a trusted server (different from the untrusted server used as part of SPORC's protocols).

3 System Design

This section describes SPORC's design in more detail, including its synchronization mechanisms and the measures that clients implement to detect a malicious server that may modify, reorder, duplicate, or drop operations. This section assumes that the set of users and clients editing a given document is fixed; we consider dynamic membership in Section 4.

3.1 System Overview

The parties and stages involved with SPORC operations are shown in Figure 1. At a high level, the local state of a SPORC application is synchronized between multiple clients, using a server to collect updates from clients, order them, then redistribute the client updates to others. There are four types of state in the system.

(1) The ***local state*** is a compact representation of the client's current view of the document (*e.g.*, the most recent version of a collaborative-edited text).

(2) The ***encrypted history*** is the set of operations stored at and ordered by the server. The payloads of operations that change the contents of the document are encrypted to preserve confidentiality. The server orders the operations oblivious to their payloads but aware of the previous operations on which they causally depend.

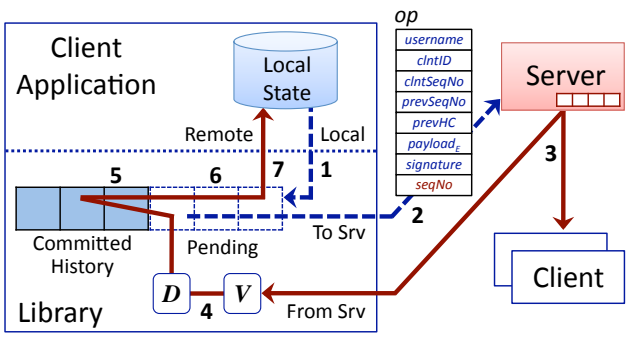


Figure 1: **SPORC architecture and synchronization steps**

(3) The ***committed history*** is the official set of (plaintext) operations shared among all clients, as ordered by the server. Clients derive this committed history from the server's encrypted history by transforming operations' payloads to reflect any changes that the server's ordering might have caused.

(4) A client's ***pending queue*** is an ordered list of the client's local operations that have already been applied to its local state, but that have yet to be committed (*i.e.*, assigned a sequence number by the server and added to the client's committed history).

SPORC synchronizes clients' local state for a particular document using the following steps, also shown in Figure 1. This section restricts its consideration to interactions with a static membership and well behaved server; we relax these restrictions in the next two sections, respectively. The flow of local operations to the server is illustrated by dashed blue arrows; the flow of operations received from the server is shown by solid red arrows.

1. A client application generates an operation, applies it to its local state immediately, and then places it at the end of the client's pending queue.
2. If the client does not currently have any operations under submission, it takes its oldest queued operation yet to be sent, *op*, assigns it a client sequence number (*clntSeqNo*), embeds in it the global sequence number of the last committed operation (*prevSeqNo*) along with the corresponding hash chain value (*prevHC*), encrypts its payload, digitally signs it, and transmits it to the server. (As an optimization, if the client has multiple operations in its pending queue, it can submit them as a single batched operation.)
3. The server adds the client-submitted *op* to its encrypted history, assigning it the next available global sequence number (*seqNo*). The server forwards *op* with this *seqNo* to all the clients participating in the document.
4. Upon receiving an encrypted operation *op*, the client verifies its signature (V) and checks that its *clntSe-*

qNo, *seqNo*, and *prevHC* fields have the expected values. If these checks succeed, the client decrypts the payload (D) for further processing. If they fail, the client concludes that the server is malicious.

5. Before adding *op* to its committed history, the client must transform it past any other operations that had been committed since *op* was generated (*i.e.*, all those with global sequence numbers greater than *op*'s *prevSeqNo*). Once *op* has been transformed, the client appends *op* to the end of the committed history.
6. If the incoming operation *op* was one that the client had initially sent, the client dequeues the oldest element in the pending queue (which will be the uncommitted version of *op*) and prepares to send its next operation. Otherwise, the client transforms *op* past all its pending operations and, conversely, transforms those operations with respect to *op*.
7. The client returns the transformed version of the incoming operation *op* to the application. The application then applies *op* to its local state.

SPORC maintains the following invariants with respect to the system's state:

Local Coherence: A client's local state is equivalent to the state it would be in if, starting with an initial empty document, it applied, in order, all of the operations in its committed history followed by all of the operations in its pending queue.

Fork* Consistency: If the server is well behaved, all clients' committed histories are linearizable (*i.e.*, for every pair of clients, one client's committed history is equal to or a prefix of the other client's committed history). If the server is faulty, however, clients' committed histories may be forked [23].

Client-Order Preservation: The order that a non-malicious server assigns to operations originating from a given client must be consistent with the order that the client assigned to those operations.

3.2 Operations

SPORC clients exchange two types of operations: *document operations*, which represent changes to the content of the document, and *meta-operations*, which represent changes to document metadata such as the document's access control list. Meta-operations are sent to the server in the clear, but the payloads of document operations are encrypted under a symmetric key that is shared among all of the clients but is unknown to the server. (See Section 4.1 for a description of how this key is chosen and distributed.) In addition, every operation is labeled with the name of the user that created it and is digitally signed by that user's private key. All operations also contain a unique client ID (*clntID*) that identifies from which of the user's client machines it came.

3.3 The Server's Limited Role

Because the SPORC server is untrusted, its role is limited to ordering and storing the operations that clients submit, most of which are encrypted. The server stores the operations in its encrypted history so that new clients joining the document or existing clients that have been disconnected can request from the server the operations that they are missing. This storage function is not essential, however, and in principle it could be handled by a different party.

Notably, since the server does not have access to the plaintext of document operations, the same generic server implementation can be used for any application that uses our protocol regardless of the kind of document being synchronized.

3.4 Sequence Numbers and Hash Chains

SPORC clients use sequence numbers and a hash chain to ensure that operations are properly serialized and that the server is well behaved. Every operation has two sequence numbers: a client sequence number (*clntSeqNo*) which is assigned by the client that submitted the operation, and a global sequence number (*seqNo*) which is assigned by the server. On receiving an operation, a client verifies that the operation's *clntSeqNo* is one greater than the last *clntSeqNo* seen from the submitting client, and that the operation's *seqNo* is one greater than the last *seqNo* that the receiving client saw. These sequence number checks enforce the "client order preservation" invariant and ensure that there are no gaps in the sequence of operations.

When a client uploads an operation op_{new} to the server, the client sets op_{new}'s *prevSeqNo* field to the global sequence number of the last committed operation, op_n, that the client knows about. The client also sets op_{new}'s *prevHC* field to the value of the client's hash chain over the committed history up to op_n. A client who receives op_{new} compares its *prevHC* with the client's own hash chain computation up to op_n. If they match, the recipient knows that its committed history is identical to the sender's committed history up to op_n, thereby guaranteeing fork* consistency.

A misbehaving server cannot modify the *prevSeqNo* or *prevHC* fields, because they are covered by the submitting client's signature on the operation. The server can try to tell two clients different global sequence numbers for the same operation, but this will cause the two clients' histories—and hence their future hash chain values—to diverge, and it will eventually be detected.

To simplify the design, each SPORC client has at most one operation "in flight" at any time: only the operation at the head of a client's pending queue can be sent to the server. Among other benefits, this rule ensures that operations' *prevSeqNo* and *prevHC* values will always refer

to operations that are in the committed history, and not to other operations that are "in flight." This restriction could be relaxed, but only at considerable cost in complexity. For similar reasons, other OT-based systems such as Google Wave adopt the same rule [44].

Prohibiting more than one in-flight operation per client is less restrictive than it might seem, as operations can be combined or batched. Like Wave, SPORC includes an application-specific composition function, which consolidates two operations into one. This can be used iteratively to combine a sequence of operations into a single one. Further, it is straightforward to batch multiple operations into a single logical operation, which is then submitted as a unit. Because operations can be composed or batched, a client can empty its pending queue every time it gets an opportunity to submit an operation to the server.

3.5 Resolving Conflicts with OT

Once a client has validated an operation received from the server, the client must use OT to resolve the conflicts that may exist between the new operation and other operations in the committed history and pending queue. These conflicts might have arisen for two reasons. First, the server may have committed additional operations since the new operation was generated. Second, the receiving client's local state might reflect uncommitted operations that reside on the client's pending queue but that other clients do not yet know about.

Before a client appends an incoming operation op_{new} to its committed history, it compares op_{new}'s *prevSeqNo* value with the global sequence number of the last committed operation. The *prevSeqNo* field indicates the last committed operation that the submitting client knew about when it uploaded op_{new}. Thus, if the values match, the client knows that no additional operations have been added to its committed history since op_{new} was generated, and the new operation can be appended directly to the committed history. But if they do not match, then other operations were committed since op_{new} was sent, and op_{new} needs to be transformed past each of them. For example, if op_{new} has a *prevSeqNo* of 10, but was assigned global sequence number 14 by the server, then the client must compute $op'_{new} \leftarrow T(op_{new}, \langle op_{11}, op_{12}, op_{13} \rangle)$ where $\langle op_{11}, op_{12}, op_{13} \rangle$ are the intervening committed operations. Only then can the resulting transformed operation op'_{new} be appended to the committed history. After appending the operation, the client updates the hash chain computed over the committed history so that future incoming operations can be validated.

At this point, if op'_{new} is one of the receiving client's own operations that it had previously uploaded to the server (or a transformed version of it), it will necessarily match the operation at the head of the pending queue. Since op'_{new} has now been committed, its uncommitted version can be retired from the pending queue, and the next pending operation can be submitted to the server. Furthermore, since the client has already optimistically applied the operation to its local state even before sending it to the server, the client does not need to apply op'_{new} again, and nothing more needs to be done.

If op'_{new} is not one of the client's own operations, however, the client must perform additional transformations in order to reestablish the "local coherence" invariant, which states that the client's local state is equal to the in-order application of its committed history followed by its pending queue. First, in order to obtain a version of op'_{new} that it can apply to its local state, the client must transform op'_{new} past all of the operations in its pending queue. This step is necessary because the pending queue contains operations that the client has already applied locally, but have not yet been committed and, therefore, were unknown to the sender of op'_{new}.

Second, the client must transform the entire pending queue with respect to op'_{new} to account for the fact that op'_{new} was appended to the committed history. More specifically, the client computes $\langle \overline{op'_1}, \ldots, \overline{op'_m} \rangle \leftarrow T(\langle \overline{op_1}, \ldots, \overline{op_m} \rangle, op'_{new})$ where $\langle \overline{op_1}, \ldots, \overline{op_m} \rangle$ is the pending queue. This transformation has the effect of pushing the pending queue forward by one operation to make room for the newly extended committed history. The operations on the pending queue need to stay ahead of the committed history because they will receive higher global sequence numbers than any of the currently committed operations. Furthermore, by transforming its unsent operations in response to updates to the document, the client reduces the amount of transformation that other clients will need to do when they eventually receive its operations.

4 Membership Management

Document membership in SPORC is controlled at the level of users, each of which is associated with a public-private key pair. When a document is first created, only the user that created it has access. Subsequently, privileged users can change the document's access control list (ACL) by submitting `ModifyUserOp` meta-operations, which get added to the document's history (covered by its hash chain), much like normal operations.

A user can be given one of three privilege levels: *reader*, which entitles the user to decrypt the document but not to submit new operations; *editor*, which entitles the user to read the document and to submit new operations (except those that change the ACL); and *administrator*, which grants the user full access, including the ability to invite new users and remove existing users. Because `ModifyUserOp`s are not encrypted, a non-malicious server will immediately reject operations from users with insufficient privileges. But because the server is untrusted,

every client maintains its own copy of the ACL, based on the history's `ModifyUserOp`s, and refuses to apply operations that came from unauthorized users.

4.1 Encrypting Document Operations

To prevent eavesdropping by the server or unapproved users, the payloads of document operations are encrypted under a symmetric key known only to the document's current members. More specifically, to create a new document, the creator generates a random AES key, encrypts it under her own public key, and then writes the encrypted key to the document's initial create meta-operation. To add new users, an administrator submits a `ModifyUserOp` that includes the document's AES key encrypted under each of the new users' public keys.

If users are removed, the AES key must be changed so that the removed users will not be able to decrypt subsequent operations. To do so, an administrator picks a new random AES key, encrypts it under the public keys of all the remaining participants, and then submits the encrypted keys as part of the `ModifyUserOp`.[3] This meta-operation also includes an encryption of the old AES key under the new AES key. This enables later users to learn earlier keys and thus decrypt old operations, without requiring the operations to be re-encrypted.

SPORC's model ensures proper access control over operations, based on how it tracks potential causality through *prevSeqNo* dependencies. Operations concurrent to a `ModifyUserOp` removal may be ordered before it and remain accessible to the user. However, once a client sees the removal meta-operation in its committed history any subsequent operation the client submits will be inaccessible to the removed user.

4.2 Barrier Operations

Concurrency also poses a challenge to membership management. Consider the situation when two clients concurrently issue `ModifyUserOp`s that both attempt to change the current symmetric key. If the server naively scheduled one after the other, then the continuous chain of old keys encrypted under new ones would be broken.

To address situations like this, we introduce a primitive called a *barrier operation*. When the server receives an operation that is marked "barrier" and assigns it global sequence number b, the server requires that every subsequent operation have a *prevSeqNo* $\geq b$. Subsequent operations that do not are rejected and must be revised and resubmitted with a later *prevSeqNo*. In this way, the server can force all future operations to depend on the barrier operation.[4]

Let us reconsider the example of two concurrent `ModifyUserOp`s, op_1 and op_2, that are marked as barriers. Suppose that the server received op_1 first and assigned it sequence number b. Since the operations were submitted concurrently, op_2's *prevSeqNo* will necessarily be less than b, and op_2 will be rejected. The client attempting to send op_2 must wait until it receives op_1, at which time it will adjust op_2 to depend on this operation before resubmitting (*i.e.*, encrypt op_1's key under its new key, and set op_2's *prevSeqNo* $\geq b$). As a result, the chain of old keys encrypted under new ones will be preserved.

Barrier operations have uses beyond membership management. For example, as described next, they are useful in implementing checkpoints on the history.

5 Extensions

This section describes extensions to the basic SPORC protocols: supporting checkpoints to reduce the size requirements for storing the committed history (Section 5.1), detecting forks through out-of-band communication (Section 5.2), and recovering from forks by replaying and possibly transforming forked operations (Section 5.3). Our current prototype does not yet implement these extensions, however.

5.1 Checkpoints

In order to reach a document's latest state, a new client in our current implementation must download and apply the entire history of committed operations. It would be more efficient for a new client to instead download a *checkpoint* of operations—a compact representation of the document's state, akin to each client's local state—and then only apply individual committed operations since the last checkpoint. Much as SPORC servers cannot transform operations, they similarly cannot perform checkpoints; SPORC once again has individual clients play this role.

To support checkpoints, each client maintains a compacted version of the committed history up to the most recent barrier operation. When a client is ready to upload a checkpoint to the server, it encrypts this compacted history under the current document key. It then creates a new `CheckpointOp` meta-operation containing the hash of the encrypted checkpoint data and submits it into the history. Requiring the checkpoint data to end in a barrier operation ensures that clients that later use the checkpoint will be able to ignore the history before the barrier without having to worry that they will need to perform OT transformations involving that old history. After all, no operation

[3] In our current implementation, the size of a `ModifyUserOp` may be linear in the number of users participating in the document, because the operation may contain the current AES key encrypted under each of the users' RSA public keys. An optimization to achieve constant-sized `ModifyUserOp`s could instead use a space-efficient broadcast encryption scheme [6].

[4] To prevent a malicious server from violating the rules governing barrier operations, an operation's "barrier" flag is covered by the operation's signature, and all clients verify that the server is handling barrier operations correctly.

after a barrier can depend on an operation before it. If the most recent barrier is too old, the client can submit a new null barrier operation before creating the checkpoint.[5]

Checkpoints raise new security challenges, however. A client that lacks the full history cannot verify the hash chain all the way back to the document's creation. It can verify that the operations it has chain together correctly, but the first operation in its history (i.e., the barrier operation) is "dangling," and its *prevHC* value cannot be verified. This is not a problem if the client knows in advance that the `CheckpointOp` is part of the valid history, but this is difficult to verify. The `CheckpointOp` will be signed by a user, and users who have access to the document are assumed to be trusted; but there must be a way to verify that the signing user had permission to access the document at the time the checkpoint was created. Unfortunately, without access to a verifiable history of individual `ModifyUserOp`s going back the beginning of the document, a client deciding whether to accept a checkpoint has no way to be certain of which users were actually members of the document at any given time.

To address these issues, we propose that the server and clients maintain a *meta-history*, alongside the committed history, that is comprised solely of meta-operations. Meta-operations are included in the committed history as before, but each one also has a *prevMetaSeqNo* pointer to a prior element of the meta-history along with a corresponding *prevMetaHC* field. Each client maintains a separate hash chain over the meta-history and performs the same consistency checks on the meta-history that it performs on the committed history.

When a client joins, before it downloads a checkpoint, it requests the entire meta-history from the server. The meta-history provides the client with a fork* consistent view of the sequence of `ModifyUserOp`s and `CheckpointOp`s that indicates whether the checkpoint's creator was an authorized user when the checkpoint was created. Moreover, the cost of downloading the entire meta-history is likely to be low because meta-operations are rare relative to document operations.

5.2 Checking for Forks Out-of-Band

Fork* consistency does not prevent a server from forking clients' state, as long as the server never tells any member of one fork about any operation done by a member of another fork. To detect such forks, clients can exchange state information out-of-band, for example, by direct socket connections, email, instant messaging, or posting on a shared server or DHT service.

Clients can exchange messages of the form $\langle c, d, s, h_s \rangle$, asserting that in client c's view of document d, the hash chain value as of sequence number s is equal to h_s. On receiving such a message, a client compares its own hash chain value at sequence number s with h_s, and if the values differ, it knows a fork has occurred. If the recipient does not yet have operations up to sequence number s, it requests them from the server; a well behaved server will always be able to supply the missing operations.

These out-of-band messages should be digitally signed to prevent forgery. To prevent out-of-band messages from leaking information about which clients are collaborating on a document, and to prevent a client from falsely claiming that it was invited into the document by a forked client, the out-of-band messages should be encrypted and MACed with a separate set of symmetric keys that are known only to nodes that have been part of the document.[6] These keys might be conveyed in the first operation of the document's history.

5.3 Recovering from a Fork

A benefit of combining OT and fork* consistency is that we can use OT to recover from forks. OT is well suited to this task because, in normal operation, OT clients are essentially creating small forks whenever they optimistically apply operations locally, and resolving these forks when they transform operations to restore consistency. In this section, we sketch an algorithm that a pair of forked clients can use to merge their divergent histories into a consistent whole. This pairwise algorithm can be repeated as necessary to resolve forks involving multiple clients, or multi-way forks.

The basic idea of the algorithm is that the two clients will abandon the malicious server and agree on a new one. Both clients will roll back their histories to their last common point before the fork, and one of them will upload the common history, up to the fork point, to the new server. Finally, each client will resubmit the operations that it saw after the fork. OT will ensure that these resubmitted operations are merged safely so that both nodes end up in the same state.

The situation becomes more complicated if the same operation appears in both histories. We cannot just remove the duplicate because later operations in the sequence may depend on it. Instead, we must cancel it out. To make this possible, we require that all operations be invertible:

[5] Having the checkpoint data end in an earlier barrier operation is better than making `CheckpointOp`s into barriers themselves. If `CheckpointOp`s were barriers, then either the client making the checkpoint would have to "lock" the history to prevent new operations from being admitted before the checkpoint was uploaded, or the system would have to reject checkpoints that did not reflect the latest state, which could potentially lead to livelock.

[6] A client falsely claiming to have been invited into the document in another fork will eventually be detected when the other clients try to recover from the (false) fork. However, this is expensive so we would prefer to avoid it. By protecting the out-of-band messages with symmetric keys known only to clients who have been in the document at some point, we reduce the set of potential liars substantially.

we must be able to construct an inverse operation op^{-1} such that applying op followed by op^{-1} results in a no-op. This is often easy to do in practice by having each operation store enough information about the prior state to determine what the inverse should be. For example, a delete operation can store the information that was deleted, enabling the creation of an insert operation as the inverse.

To cancel each duplicate, we cannot simply splice its inverse into the history right after it for the same reason that we cannot just remove the duplicate. Instead, we compute the inverse operation and then transform it past all of the operations following the duplicate. This process results in an operation that has the effect of canceling out the duplicate when appended to the end of the sequence.

6 Implementation

SPORC provides a framework for building collaborative applications that need to synchronize different kinds of state between clients. It consists of a generic server implementation and client-side libraries that implement the SPORC protocol, including the sending, receiving, encryption, and transformation of operations, as well as the necessarily consistency checks and document membership management. To build applications within the SPORC framework, a developer only needs to implement client-side functionality that (i) defines a data type for SPORC operations, (ii) defines how to transform a pair of operations, and (iii) defines how to combine multiple document operations into a single one. The server need not be modified, as it always deals with operations on encrypted data.

6.1 Variants

We implemented two variants of SPORC: a command-line version in which both client and server are stand-alone applications, and a web-based version with a browser-based client and a Java servlet. The command-line version, which we use for later microbenchmarks, is written in approximately 5500 lines of Java code (per SLOCCount [46]) and, for network communication, uses the socket-based RPC library in the open-source release of Google Wave [16]. Because the server's role is limited to ordering and storing client-supplied operations, its basic implementation is simple and only requires approximately 300 lines of code.

The web-based version shares the majority of its code with the command-line variant. The server just encapsulates the command-line server functionality in a Java servlet. The client consists almost entirely of JavaScript code that was automatically generated using the Java-to-JavaScript compiler included with the Google Web Toolkit (GWT) [12]. Network communication uses a combination of the GWT RPC framework, which wraps browser `XmlHttpRequests`, and the GWTEventService [37], which allows the server to push messages to the browser asynchronously through a long-lived HTTP connection (the so-called "Comet" style of web programming). This prototype could be extended with HTML5's offline storage to provide disconnected operation.

The client's use cryptographic module was its only component that could not be translated to JavaScript. JavaScript remains too slow to implement public key cryptography efficiently, and browsers lack both secure storage for cryptographic keys and a secure pseudorandom number generator for key generation. To work around these limitations, we encapsulate our cryptographic module in a Java applet and implement JavaScript-to-Java communication using the LiveConnect API [28] (a strategy employed in [2, 47]). Our experience suggests it would be beneficial for browsers to provide a JavaScript API that supported basic cryptographic primitives.

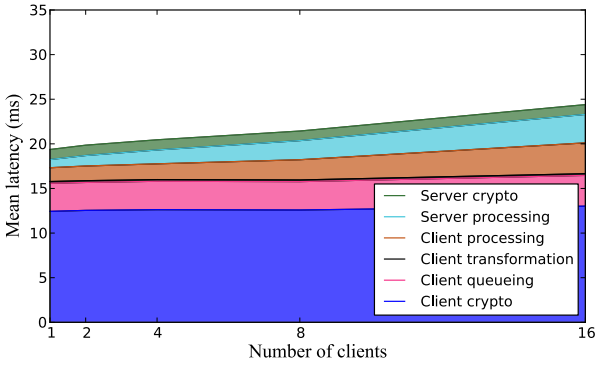
6.2 Building SPORC Applications

To demonstrate the usefulness of our framework, we built two prototype applications: a causally-consistent key-value store and a web-based collaborative text editor. The key-value store keeps a simple dictionary—mapping strings to strings—synchronized across a set of participating clients. To implement it, we defined a data type that represents a list of keys to update or remove. We wrote a simple transformation function that implements a "last writer wins" policy, as well as a composition function that merges two lists of key updates in a straightforward manner. Overall, the application-specific portion of the key-value store only required 280 lines of code.

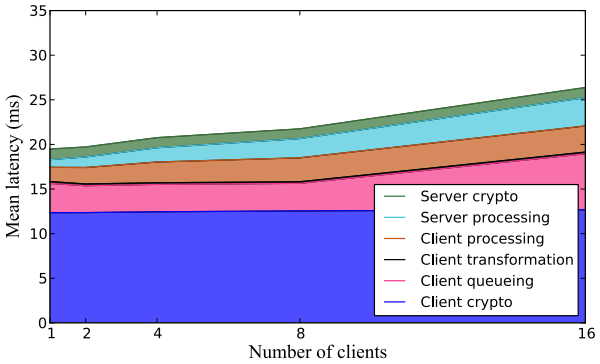
The collaborative editor allows multiple users to modify a text document simultaneously via their web browsers and see each other's changes in near real-time. It provides a user experience similar to Google Docs [14] and EtherPad [13], but, unlike those services, it does not require a trusted server. To implement it, we were able to reuse the data types and the transformation and composition functions from the open-source release of Google Wave. Although Wave is a server-centric OT system without SPORC's level of security and privacy, we were able to adapt its components for our framework with only 550 lines of wrapper code.

7 Experimental Evaluation

The user-facing collaborative applications for which SPORC was designed—*e.g.*, word processing, calendaring, and instant messaging—require latency that is low enough for human users to see each others' updates in real-time. But unlike file or storage systems, their primary goal is not high throughput. In this section, we present the results of several microbenchmarks of our Java-based

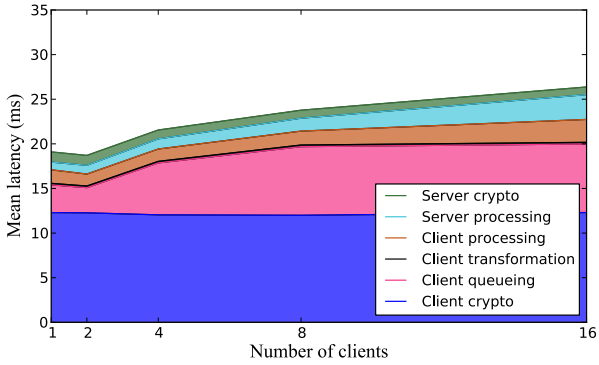


(a) Unloaded key-value store

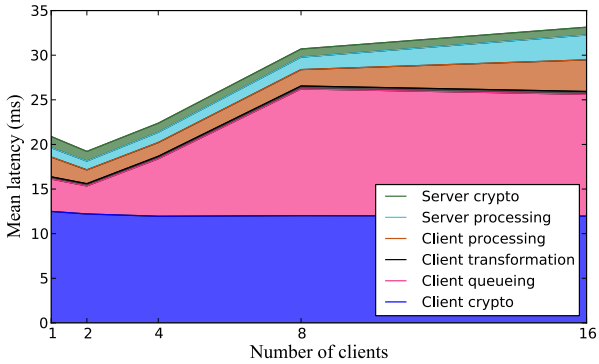


(b) Unloaded text editor

Figure 2: **Latency of SPORC with a single client writer**



(a) Loaded key-value store



(b) Loaded text editor

Figure 3: **Latency of SPORC with all clients issuing writes**

command-line version, to demonstrate SPORC's usefulness for this class of applications.

We performed our experiments on a cluster of five commodity machines, each with eight 2.3 GHz AMD Opteron cores and 8 GB of RAM, that were connected by gigabit switched Ethernet. In each of our experiments, we ran a single server instance on its own machine, along with varying numbers of client instances. To scale our system to moderate numbers of clients, in many of our experiments, we ran multiple client instances on each machine. We ran all the experiments under the OpenJDK Java VM (version IcedTea6 1.6). For RSA signatures, however, we used the Network Security Services for Java (JSS) library from the Mozilla Project [29] because, unlike Java's default cryptography library, it is implemented in native code and offers considerably better performance.

Latency. To measure SPORC's latency, we conducted three minute runs with between one and sixteen clients for both key-value and text editor operations. We tested our system under both low-load conditions, where only one of the clients submitted new operations (once every 200 ms), and high-load conditions, where all of the clients were writers. We measured latency by computing the mean time that an operation was "in flight": from the time that it was generated by the sender's application-level code, until the time it was delivered to the recipient's application.

Under low-load conditions with only one writer, we would expect the load on each client to remain constant as the number of clients increases, because each additional client does not add to the total number of operations in flight. We would, however, expect to see server latency increase modestly, as the server has to send operations to increasing numbers of clients. Indeed, as shown in Figure 2, the latency due to server processing increased from under 1 ms with one client to over 3 ms with sixteen clients, while overall latency increased modestly from approximately 19 ms to approximately 25 ms.[7]

On the other hand, when every client is a writer, we would expect the load on each client to increase with the number of clients. As expected, Figure 3 shows that with sixteen clients under loaded conditions, overall latency is higher: approximately 26 ms for key-value operations and 33 ms for the more expensive text-editor operations. The biggest contributor to this increase is client queueing, which is primarily the time that a client's received operations spend in its incoming queue before being processed. Queueing delay begins at around 3 ms for one

[7]Figure 2 also shows small increases in the latency of client processing and queuing when the number of clients was greater than four. These increases are most likely due to the fact that, when we conducted experiments with more than four clients, we ran multiple client instances per machine.

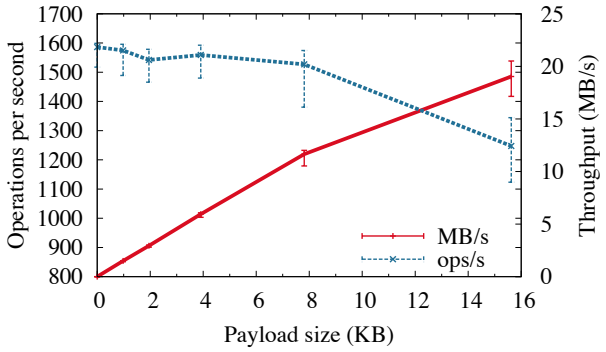


Figure 4: **Server throughput as a function of payload size.**

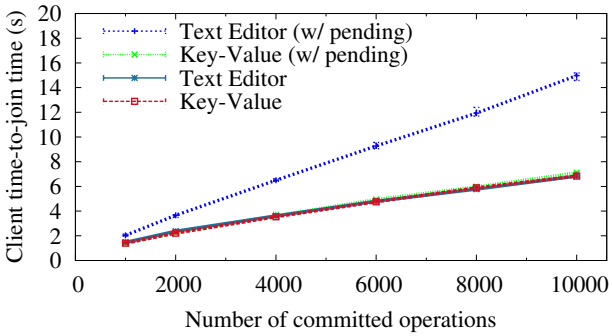


Figure 5: **Client time-to-join given a variable length history**

client and then increases steadily until it levels off at approximately 8 ms for the key-value application and 14 ms for the text editor. Despite this increase, Figure 3 demonstrates that SPORC successfully supports real-time collaboration for moderately-sized groups, even under load. As these experiments were performed on a local-area network, a wide-area deployment of SPORC would see an increase in latency that reflects the correspondingly higher network round-trip-time.

Figures 2 and 3 also show that client-side cryptographic operations account for a large share of overall latency. This occurs because SPORC performs a 2048-bit RSA signature on every outgoing operation and because Mozilla JSS, while better than Java's cryptography built-in library, still requires about 10 ms to compute a single signature. Using an optimized implementation of a more efficient signature scheme, such as ESIGN, could improve the latency of signatures by nearly two orders of magnitude [24].

Server throughput. We measured the server's maximum throughput by saturating the server with operations using 100 clients. These particular clients were modified to allow them to have more than one operation in flight at a time. Figure 4 shows server throughput as a function of payload size, measured in terms of both operations per second and MB per second. Each data point was computed by performing a three minute run of the system and then taking the median of the mean throughput of each one second interval. The error bars represent the 5th and 95th percentiles. The figure shows that, as expected, when payload size increases, the number of operations per second decreases, because each operation requires more time to process. But, at the same time, data throughput (MB/s) increases, because the processing overhead per byte decreases.

Client time-to-join. Because our current implementation lacks the checkpoints of Section 5.1, when a client joins the document, it must first download each individual operation in the committed history. To evaluate the cost of joining an existing document, we first filled the history with varying numbers of operations. Then, we measured the time it took for a new client to receive the shared decryption key and download and process all of the committed operations. We performed two kinds of experiments: one where the client started with an empty local state, and a second in which the client had 2000 pending operations that had yet to be submitted to the server. The purpose of the second test was to measure how long it would take for a client that had been working offline for some length of time to synchronize with the current state of the document. Synchronization requires the client to transform its pending operations past the committed operations that the client has not seen; thus, it is more costly than joining a document with an empty local state. Notably, since the-fork recovery algorithm sketched in Section 5.3 relies on the same mechanism that is used to synchronize clients that have been offline—it treats operations after the fork as if they were pending uncommitted operations—this test also sheds light on the cost of recovering from a fork.

Figure 5 shows time-to-join as a function of history size. Each data point represents the median of ten runs, and the error bars correspond to the 10th and 90th percentiles. We find that time-to-join is linear in the number of committed operations. It takes a client with an empty local state approximately one additional second to join a document for every additional 1000 committed operations.

In addition, the figure shows that the time-to-join with a significant number of pending operations varies greatly by application. In the key-value application, the transformation function is cheap, because it is effectively a no-op if the given operations do not affect the same keys. As a result, the cost of transforming 2000 operations adds little to the time-to-join. By contrast, the text editor's more complex transformation function adds a non-trivial, although still acceptable, amount of overhead.

8 Related Work

Real-time "groupware" collaboration systems have adapted classic distributed systems techniques for time-stamping and ordering (*e.g.*, [4, 5, 20]), but have also introduced novel techniques to automatically resolve

conflicts between concurrent operations in an intention-preserving manner (*e.g.*, [11, 18, 33, 38, 39, 40, 41, 42]). These techniques form the basis of SPORC's client synchronization mechanism and allow it to support slow or disconnected networks. Several systems also use OT to implement undo functionality (*e.g.*, [32, 33]), and SPORC's fork recovery algorithm draws upon these approaches. Furthermore, as an alternative to OT, Bayou [43] allows applications to specify conflict detection and merge protocols to reconcile concurrent operations. Most of these protocols focus on decentralized settings and use n-way reconciliation, but several well-known systems use a central server to simplify synchronization between clients (including Jupiter [30] and Google Wave [44]). SPORC also uses a central server for ordering and storage, but allows the server to be untrusted. Secure Spread [3] presents several efficient message encryption and key distribution architectures for such client-server group collaboration settings. But unlike SPORC, it relies on trusted servers that can generate keys and re-encrypt messages as needed.

Traditionally, distributed systems have defended against potentially malicious servers by replicating functionality and storage over multiple servers. Protocols, such as Byzantine fault tolerant (BFT) replicated state machines [9, 21, 48] or quorum systems [1, 26], can then guarantee safety and liveness, provided that some fraction of these servers remain non-faulty. Modern approaches optimize performance by, for example, concurrently executing independent operations [19], permitting client-side speculation [45], or supporting eventual consistency [35]. BFT protocols face criticism, however, because when the number of correct servers falls below a certain threshold (typically two-thirds), they cannot make progress.

Subsequently, variants of fork consistency protocols (*e.g.*, [7, 27, 31]) have addressed the question of how much safety one can achieve with a single untrusted server. These works demonstrate that server *equivocation* can always be detected unless the server permanently forks the clients into groups that cannot communicate with each other. SUNDR [24] and FAUST [8] use these fork consistency techniques to implement storage protocols on top of untrusted servers. Other systems, such as A2M [10] and TrInc [22], rely on trusted hardware to detect server equivocation. BFT2F [23] combines techniques from BFT replication and SUNDR to achieve fork* consistency with higher fractions of faulty nodes than BFT can resist. SPORC borrows from the design of BFT2F in its use of hash chains to limit equivocation, but unlike BFT2F or any of these other systems, SPORC allows disconnected operation and enables clients to recover from server equivocation, not just detect it.

Like SPORC, two very recent systems, Venus [34] and Depot [25], allow clients to use a cloud resource without having to trust it, and they also support some degree of disconnected operation. Venus provides strong consistency in the face of a potentially malicious server, but does not support applications other than key-value storage. Furthermore, unlike SPORC, it requires the majority of a "core set" of clients to be online in order to achieve most of its consistency guarantees. In addition, although members may be added dynamically to the group editing the shared state, it does not allow access to be revoked, nor does it provide a mechanism for distributing encryption keys. Depot, on the other hand, does not rely on the availability of a "core set" of clients and supports varied applications. Moreover, similar to SPORC, it allows clients to recover from malicious forks using the same mechanism that it uses to keep clients synchronized. But rather than providing a means for reconciling conflicting operations as SPORC does with OT, Depot relies on the application for conflict resolution. Because Depot treats clients and servers identically, it can also tolerate faulty clients, in addition to faulty servers. Unlike SPORC, however, Depot does not consider dynamic access control or confidentiality.

9 Conclusion

Our original goal for SPORC was to design a general framework for web-based group collaboration that could leverage cloud resources, but not be beholden to them for privacy guarantees. This goal leads to a design in which servers only store encrypted data, and each client maintains its own local copy of the shared state. But when each client has its own copy of the state, the system must keep them synchronized, and operational transformation provides a way do to so. OT enables optimistic updates and automatically reconciles clients' conflicting states.

Supporting applications that need timely commits requires a central server. But if we do not trust the server to preserve data privacy, we should not trust it to commit operations correctly either. This requirement led us to employ fork* consistency techniques to allow clients to detect server equivocation about the order of committed operations. But beyond the benefits that each provides independently, this work shows that OT and fork* consistency complement each other well. Whereas prior systems that enforced fork* consistency alone were only able to detect malicious forks, by combining fork* consistency with OT, SPORC can recover from them using the same mechanism that keeps clients synchronized.

In addition to these conceptual contributions, we present a membership management architecture that provides dynamic access control and key distribution with an untrusted server, even in the face of concurrency. Finally, we also demonstrate the flexibility of our design by implementing two applications: a causally-consistent key-value store and a browser-based collaborative text editor.

Acknowledgments. We thank Siddhartha Sen, Jinyuan Li, Alma Whitten, Alexander Shraer, and Christian Cachin for their insights. We also thank our shepherd, Lidong Zhou, and the anonymous reviewers for their helpful comments. This research was supported by funding from Google and the NSF CAREER grant CNS-0953197.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. SOSP*, Oct. 2005.

[2] B. Adida. Helios: Web-based open-audit voting. In *Proc. USENIX Security*, Aug. 2008.

[3] Y. Amir, C. Nita-rotaru, J. Stanton, and G. Tsudik. Secure spread: An integrated architecture for secure group communication. *IEEE Trans. Dependable and Secure Computing*, 2:248–261, 2005.

[4] P. Bernstein, N. Goodman, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comp. Systems*, 9(3): 272–314, Aug. 1991.

[6] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *Advances in Cryptology – CRYPTO*, Aug. 2005.

[7] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. PODC*, Aug. 2007.

[8] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. Dependable Systems and Networks (DSN)*, June 2009.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, Feb. 1999.

[10] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, Oct. 2007.

[11] C. Ellis and S. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[12] Google. Google Web Toolkit (GWT). `http://code.google.com/webtoolkit/`, 2010.

[13] Google. EtherPad. `http://etherpad.com/`, 2010.

[14] Google. Google Docs. `http://docs.google.com/`, 2010.

[15] Google. Government requests directed to Google and YouTube. `http://www.google.com/governmentrequests/`, 2010.

[16] Google. Google Wave federation protocol. `http://code.google.com/p/wave-protocol/`, 2010.

[17] M. Handley and J. Crowcroft. Network text editor (NTE): A scalable shared text editor for MBone. In *Proc. SIGCOMM*, Oct. 1997.

[18] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. ICDCS*, May 1993.

[19] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *Proc. Dependable Systems and Networks (DSN)*, June 2004.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Programming Language Systems*, 4(3), 1982.

[22] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. NSDI*, Apr. 2009.

[23] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, Apr. 2007.

[24] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.

[26] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. STOC*, May 1997.

[27] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. PODC*, July 2002.

[28] Mozilla Project. LiveConnect. `https://developer.mozilla.org/en/LiveConnect`, 2010.

[29] Mozilla Project. Network security services for Java (JSS). `https://developer.mozilla.org/En/JSS`, 2010.

[30] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proc. UIST*, Nov. 1995.

[31] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. Symposium on Distributed Computing (DISC)*, Sept. 2006.

[32] A. Prakash and M. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Computer-Human Interaction*, 4(1):295–330, Dec. 1994.

[33] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. CSCW*, Nov. 1996.

[34] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. ACM CCSW*, Oct. 2010.

[35] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *Proc. NSDI*, Apr. 2009.

[36] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[37] S. Strohschein. GWTEventService. `http://code.google.com/p/gwteventservice/`, 2010.

[38] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in distributed collaborative environment. In *Proc. Conf. Supporting Group Work (GROUP)*, Nov. 1997.

[39] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. CSCW*, Nov. 1998.

[40] C. Sun, X. Jia, Y. Yang, and Y. Zhang. A generic operation transformation schema for consistency maintenance in realtime cooperative editing systems. In *Proc. Conf. Supporting Group Work (GROUP)*, Nov. 1997.

[41] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Computer-Human Interaction*, 5(1):64–108, 1998.

[42] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. In *Proc. CSCW*, Nov. 2004.

[43] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, Dec. 1995.

[44] D. Wang and A. Mah. Google wave operational transformation. `http://www.waveprotocol.org/whitepapers/operational-transform`, Apr. 2010.

[45] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proc. NSDI*, Apr. 2009.

[46] D. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2010.

[47] T. D. Wu. The secure remote password protocol. In *Proc. NDSS*, Mar. 1998.

[48] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, Oct. 2003.