Fork Sequential Consistency is Blocking

Christian Cachin* Idit Keidar† Alexander Shraer†

May 14, 2008

Abstract

We consider an untrusted server storing shared data on behalf of clients. We show that no storage access protocol can on the one hand preserve sequential consistency and wait-freedom when the server is correct, and on the other hand always preserve fork sequential consistency.

1 Introduction

We examine an online collaboration facility providing storage and data sharing functions for remote clients that do not communicate directly [3, 4, 13, 14]. Specifically, we consider a server that implements single-writer multi-reader registers. The storage server may be faulty, potentially exhibiting Byzantine faults [10, 8, 11, 2]. When the server is correct, strong liveness, namely *wait-freedom* [5], should be guaranteed, as a client editing a document does not want to be dependent on another client, which could even be in a different timezone [14]. In addition, although read/write operations of different clients may occur concurrently, consistency of the shared data should be provided. Specifically, we consider a service that, when the server is correct, provides *sequential consistency*, which ensures that clients have the same *view* of the order of read/write operations, which also respects the local order of operations occurring at each client [7]. Sequential consistency provides clients with a convenient abstraction of a shared storage space. It allows for more efficient implementations than stronger consistency conditions such as linearizability [6], especially when the system is not synchronized [1].

In executions where the server is faulty, liveness obviously cannot be guaranteed. Moreover, with a Byzantine server, ensuring sequential consistency is also impossible [2]. Still, it is possible to guarantee weaker semantics, in particular so-called *forking* consistency notions [8, 10]. These ensure that whenever the server causes the views of two clients to differ in a single operation, the two clients never again see each other's updates after that. In other words, if an operation appears in the views of two clients, these views are identical up to this operation.

Originally, *fork-linearizability* was considered [8, 10, 2]. In this paper, we examine the weaker *fork sequential consistency* condition, recently introduced by Oprea and Reiter [11], who showed that this new condition is sufficient for certain applications. However, to date, no fork-sequentially-consistent storage protocol has been proposed. In fact, Oprea and Reiter suggested this as a future research direction [11]. Furthermore, Cachin et al. [2] showed that the stronger notion of fork-linearizability does not allow for wait-free implementations, but conjectured that such implementations might be possible with fork sequential consistency. Surprisingly, we prove here that no storage access protocol can provide fork sequential consistency at all times and also be sequentially consistent and wait-free whenever the server is correct. This generalizes the impossibility result of Cachin et al. [2], and requires a more elaborate proof.

*IBM Research, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. cca@zurich.ibm.com

†Department of Electrical Engineering, Technion, Haifa 32000, Israel. {idish@ee, shralex@tx}.technion.ac.il

In this paper we require only sequentially consistent semantics when the server is correct. Though one may also consider stronger semantics, such as linearizability, for this case, as our goal is to prove an impossibility result, it suffices to address sequential consistency. Our impossibility result a fortiori rules out the existence of protocols with stronger semantics as well.

2 Definitions

System model. We consider an asynchronous distributed system consisting of n clients $C_1, \ldots, C_n$, a server S, and asynchronous FIFO reliable channels between the clients and S (there is no direct communication between clients). The clients and the server are collectively called *parties*. System components are modeled as deterministic I/O Automata [9]. An automaton has a state, which changes according to *transitions* that are triggered by *actions*. A *protocol* P specifies the behaviors of all parties. An execution of P is a sequence of alternating states and actions, such that state transitions occur according to the specification of system components.

All clients follow the protocol, and any number of clients can fail by crashing. The server might be faulty and deviate arbitrarily from the protocol, exhibiting so-called "Byzantine" faults [12]. A party that does not fail in an execution is *correct*. The protocol emulates a *shared functionality* F to the clients, defined analogously to shared-memory objects.

Events, operations, and histories. Clients interact with the functionality F via *operations* provided by F. As operations take time, they are represented by two *events* occurring at the client, an *invocation* and a *response*. An operation is *complete* if it has a response. For a sequence of events σ, *complete*(σ) is the maximal subsequence of σ consisting only of complete operations.

A *history* is a sequence of requests and responses of F occurring in an execution. 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. 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. 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 $o <_\pi o'$. For a sequence of events π, the subsequence of π consisting of events occurring at client C_i is denoted by $\pi|_{C_i}$. For a sequential π, the prefix of π ending with operation o is denoted by π^o.

An execution is *admissible* if the following two conditions hold: (1) the sequence of events at each client consists of alternating invocations and matching responses, starting with an invocation; and (2) the execution is fair. *Fairness* means, informally, that the execution does not halt prematurely when there are still steps to be taken or messages to be delivered (we refer to the standard literature for a formal definition of admissibility and fairness [9]).

Read/write registers. A functionality F is defined via a *sequential specification*, which indicates the behavior of F in sequential executions.

The basic functionality we consider is a *read/write register* X. A register stores a value v from a domain $\mathcal{X}$ and offers *read* and *write* operations. Initially, a register holds a special value $\perp \notin \mathcal{X}$. When a client C_i invokes a read operation, the register responds with a value v, denoted $read_i(X) \rightarrow v$. When C_i invokes a write operation with value v, denoted $write_i(X, v)$, the response of X is an acknowledgment, denoted by OK. The sequential specification requires that each read operation from X return the value written by the most recent preceding write operation, if there is one, and the initial value otherwise. We assume that the values written to every particular register are unique, i.e., no value is written more than once. This can easily be implemented by including the identity of the writer and a sequence number together with the stored value.

In this paper, we consider *single-writer/multi-reader (SWMR)* registers, where for every register, only a designated writer may invoke the write operation, but any client may invoke the read operation.

Sequential consistency. One of the most important consistency conditions for concurrent access is sequential consistency [7], which preserves the real-time order only for operations by the same client. This is in contrast to linearizability, which must preserve the real-time order for all operations.

Definition 1 (Sequential consistency [7]). A history σ is *sequentially consistent* w.r.t. a functionality F if it can be extended (by appending zero or more response events) to a history σ', and there exists a sequential permutation π of *complete*(σ') such that:

1. For every client C_i, the sequence $\pi|_{C_i}$ preserves the real-time order of σ; and
2. The operations of π satisfy the sequential specification of F.

Intuitively, sequential consistency requires that every operation takes effect at some point and occurs somewhere in the permutation π. This guarantees that every write operation is eventually seen by all clients. In other words, if an operation writes v to a register X, there cannot be an infinite number of subsequent read operations from register X that return a value written to X prior to v.

Wait-freedom. A shared functionality needs to ensure liveness. A common requirement is that clients are able to make progress independently of the actions or failures of other clients. A notion that formally captures this idea is *wait-freedom* [5].

Definition 2 (Wait-free history). A history σ is wait-free if every operation by a correct client in σ is complete.

Fork sequential consistency. The notion of fork sequential consistency [11] requires, informally, that when an operation is observed directly or indirectly by multiple clients, then the history of events occurring before the operation is the same at these clients. For instance, when a client reads a value written by another client, the reader is assured to be consistent with the writer up to its write operation.

Definition 3 (Fork sequential consistency). A history σ is *fork-sequentially-consistent* w.r.t. a functionality F if it can be extended (by appending zero or more response events) to a history σ', such that for each client C_i there exists a subsequence σ_i of *complete*(σ') and a sequential permutation π_i of σ_i such that:

1. All complete operations in $\sigma|_{C_i}$ are contained in σ_i;
2. For every client C_j, the sequence $\pi_i|_{C_j}$ preserves the real-time order of σ;
3. The operations of π_i satisfy the sequential specification of F; and
4. (*No-join*) For every $o \in \pi_i \cap \pi_j$, it holds that $\pi_i^o = \pi_j^o$.

A permutation π_i satisfying these properties is called a *view* of C_i.

Note that a view π_i of C_i contains at least all those operations that either occur at C_i or are apparent from C_i's interaction with F. A fork-sequentially-consistent history in which some permutation π of *complete*(σ') is a possible view of all clients is sequentially consistent.

We are now ready to define a fork-sequentially-consistent storage service. It should guarantee sequential consistency and wait-freedom when the server is correct, and fork sequential consistency otherwise.

Definition 4 (Wait-free fork-sequentially-consistent Byzantine emulation). A protocol P is a wait-free fork-sequentially-consistent Byzantine emulation of a functionality F on a Byzantine server S if P satisfies the following conditions:

1. If S is correct, the history of every admissible execution of P is sequentially consistent w.r.t. F and wait-free; and
2. The history of every admissible execution of P is fork sequentially consistent w.r.t. F.

We show next that wait-free fork-sequentially-consistent Byzantine emulations of SWMR registers are impossible.

3 Impossibility of Wait-Freedom with Fork Sequential Consistency

Theorem 1. *There is no wait-free fork-sequentially-consistent Byzantine emulation of $n \geq 2$ SWMR registers on a Byzantine server S.*

Proof. Towards a contradiction assume that there exists such a protocol P. Then in any admissible execution of P with a correct server, every operation of a correct client completes. We next construct three executions α, β, and γ of P, shown in Figures 1–3. All three executions are admissible, since clients issue operations sequentially, and every message sent between two correct parties is eventually delivered. There are two clients C_1 and C_2, which are always correct, and access two SWMR registers X_1 and X_2. Protocol P describes the asynchronous interaction of the clients with S; this interaction is depicted in the figures only when necessary.

Execution α. In execution α, the server is correct. The execution is shown in Figure 1 and begins with four operations by C_2: first C_2 executes a write operation with value v_1 to register X_2, denoted w_2^1, then an operation reading register X_1, denoted r_2^1, then an operation writing v_2 to X_2, denoted w_2^2, and finally again a read operation of X_1, denoted r_2^2. Since S and C_2 are correct and P is wait-free with a correct server, all operations of C_2 eventually complete.

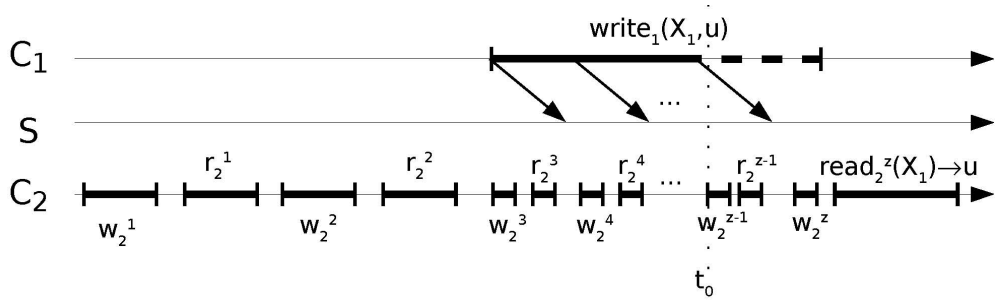

Figure 1: Execution α, where S is correct.

Execution α continues as follows. C_1 starts to execute a single write operation with value u to X_1, denoted w_1. Every time a message is sent from C_1 to S during this operation, and as long as no read operation by C_2 from X_1 returns a value different from $\perp$, the following steps are repeated in order, for $i = 3, 4, \ldots$:

(a) The message from C_1 is delayed by the asynchronous network;

(b) C_2 executes an operation writing v_i to X_2, denoted w_2^i;

(c) C_2 executes an operation reading X_1, denoted r_2^i; and

(d) the delayed message from C_1 is delivered to S.

Note that w_2^i and r_2^i complete by the assumptions that P is wait-free and that S is correct. For the same reason, operation w_1 eventually completes. After w_1 completes, and while C_2 does not read any non-$\perp$ value from X_1, C_2 continues to execute alternating operations w_2^i and r_2^i, writing v_i to X_2 and reading X_1, respectively. This continues until some read returns a non-$\perp$ value. Because S is correct, eventually some read of X_1 is guaranteed to return $u \neq \perp$ by sequential consistency of the execution. We denote the first such read by r_2^z. This is the last operation of C_2 in α. If messages are sent from C_1 to S after the completion of r_2^z, they are not delayed.

Note that the prefix of α up to the completion of r_2^3 is indistinguishable to C_2 from an execution in which no client writes to X_1, and therefore r_2^1, r_2^2, and r_2^3 return the initial value $\perp$. Hence, $z \geq 4$.

We denote the point of invocation of w_2^{z-1} in α by t_0. It is marked by a dotted line. Executions β and γ constructed below are identical to α before t_0, but differ from α starting at t_0.

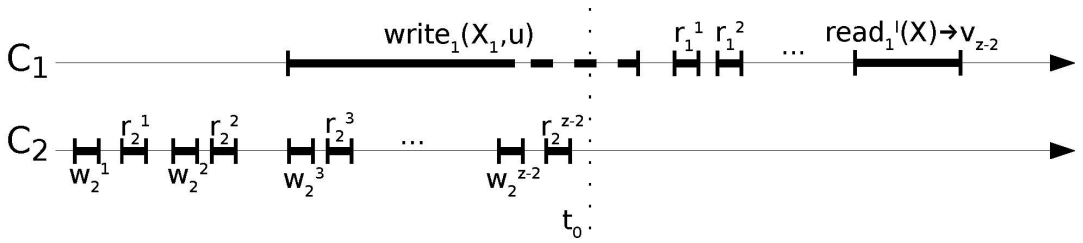

Figure 2: Execution β, where S is correct.

Execution β. We next define execution β, shown in Figure 2, in which the server is also correct. Execution β is identical to α up to the end of r_2^{z-2} (before t_0), but then C_2 halts. In other words, the last two write-read pairs of C_2 in α are missing in β. Operation w_1 is invoked in β like in α and begins after the completion of r_2^2 (notice that r_2^2 is in β since $z \geq 4$). Because the protocol is wait-free with the correct server, operation w_1 completes. Afterwards, C_1 repeatedly reads X_2 until v_{z-2} is returned. Because the execution is sequentially consistent with the correct server, a read of X_2 eventually returns v_{z-2}. We denote the i-th read operation of C_1 by r_1^i and the read operation that returns v_{z-2} by r_1^l.

Execution γ. The third execution γ is shown in Figure 3; here, the server is faulty. Execution γ proceeds just like the common prefix of α and β before t_0, and client C_1 invokes w_1 in the same way as in α and in β. From t_0 onward, the server simulates β to C_1. This is easy because S simply hides from C_1 all operations of C_2 starting with w_2^{z-1}. The server also simulates α to C_2. We next explain how this is done. Notice that in α, the server receives at most one message from C_1 between t_0 and the completion of r_2^z, and this message is sent before t_0 by construction of α. If such a message exists in α, then in γ, which is identical to α before t_0, the same message is sent by C_1. Therefore, the server has all information needed to simulate α to C_2 and r_2^z returns u.

Thus, γ is indistinguishable from α to C_2 and indistinguishable from β to C_1. However, we next show that γ is not fork-sequentially-consistent. Consider the sequential permutation π_2 required by the definition of fork sequential consistency, i.e., the view of C_2. As the real-time order of C_2's operations

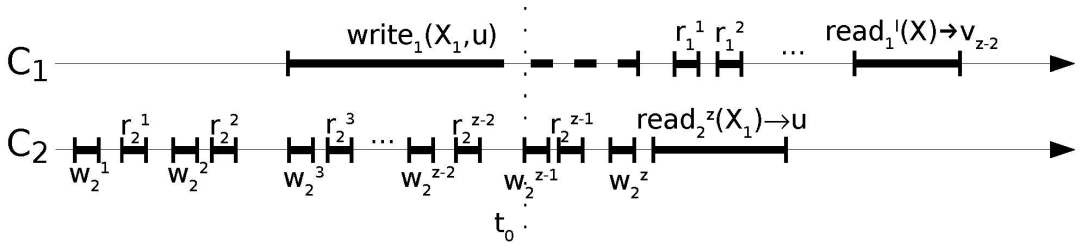

Figure 3: Execution γ, where S is faulty and simulates α to C_2 and β to C_1.

and the sequential specification of the registers must be preserved in π_2, and since r_2^1, ..., r_2^{z-1} return $\perp$ but r_2^z returns u, we conclude that w_1 must appear in π_2 and is located after r_2^{z-1} but before r_2^z. Because w_1 is one of C_1's operations, it also appears in π_1. By the no-join property, the sequence of operations preceding w_1 in π_2 must be the same as the sequence preceding w_1 in π_1. In particular, w_2^{z-1} and w_2^{z-2} appear in π_1 before w_1, and w_2^{z-2} precedes w_2^{z-1}. Since the real-time order of C_1's operations must be preserved in π_1, operation w_1 and, hence, also w_2^{z-1}, appears in π_1 before r_1^l. But since w_2^{z-1} writes v_{z-1} to X_2 and r_1^l reads v_{z-2} from X_2, this violates the sequential specification of X_2 (v_{z-2} is written only by w_2^{z-2}). This contradicts the assumption that P guarantees fork sequential consistency in all executions. □

4 Conclusions

When clients store their data on an untrusted server, strong guarantees should be provided whenever the server is correct, and forking conditions when the server is faulty. Since it was discovered that fork-linearizability does not allow for protocols that are wait-free in all executions where the server is correct [2], the weaker condition of fork sequential consistency was expected to be a promising direction to remedy this shortcoming [2, 11]. In this paper we proved that this is not the case, and in fact, fork sequential consistency suffers from the same limitation.

References

[1] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

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

[3] Collabnet, Inc. Subversion project. `http://subversion.tigris.org/`, Last accessed Apr. 2008.

[4] Google, Inc. Google Docs. `http://docs.google.com/`, Last accessed Apr. 2008.

[5] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[6] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[7] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[8] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symp. on Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

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

[10] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–117, 2002.

[11] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. 20th Intl. Symp. on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 254–268, 2006.

[12] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[13] Wikipedia. List of file systems, distributed file systems section. `http://en.wikipedia.org/wiki/List_of_file_systems#Distributed_file_systems`, Last accessed Apr. 2008.

[14] J. Yang, H. Wang, N. GU, Y. Liu, C. Wang, and Q. Zhang. Lock-free consistency control for web 2.0 applications. In *Proc. 17th Intl. Conference on World Wide Web (WWW)*, 2008.