IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Composing Ordered Sequential Consistency using Leading Updates

Kfir Lev-Ari, Edward Bortnikov, Idit Keidar and Alexander Shraer

CCIT Report #902
September 2016

THE ANDREW & ERNA VITERBI FACULTY OF ELECTRICAL ENGINEERING
TECHNION—ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 3200003, ISRAEL

CCIT Report #902 September 2016

Composing Ordered Sequential Consistency using Leading Updates

Kfir Lev-Ari[1], Edward Bortnikov[2], Idit Keidar[1,2], and Alexander Shraer[3]

[1]Viterbi Department of Electrical Engineering, Technion, Haifa, Israel
[2]Yahoo Research, Haifa, Israel
[3]Google, Mountain View, CA, USA

Abstract

We define *ordered sequential consistency* (OSC), a correctness criterion for concurrent objects, which captures the typical behavior of many real-world services, e.g., ZooKeeper, etcd, Chubby, Doozer, and Consul. A straightforward composition of OSC objects is not necessarily OSC. To remedy this, we recently implemented a composition framework that injects dummy updates in specific scenarios. We prove that injecting such updates, which we call here *leading updates*, enables correct OSC composition.

We generalize OSC to define *G-OSC*, a generic criterion for concurrent objects, which encompasses a range of criteria, including sequential consistency and linearizability.

1 Introduction

The backends of large-scale distributed applications, e.g., social networks, web search engines, content providers, and ecommerce platforms, rely on coordination services such as ZooKeeper [8], etcd [5], Chubby [6], Doozer [4], and Consul [3]. A coordination service facilitates maintaining shared state in a consistent and fault-tolerant manner. Such services are commonly used for inter-process coordination (e.g., global locks and leader election), service discovery, configuration and metadata storage, and more. They offer an abstraction of shared memory objects.

In this work we define a correctness criterion named *Ordered Sequential Consistency* (OSC), which captures the semantics of such services [3, 4, 5, 6, 8]. These coordination services provide so-called "strong consistency" for updates and some weaker semantics for reads. They are replicated for high-availability, and each client submits requests to one of the replicas. Reads are not atomic so that they can be served fast, i.e., locally by any of the replicas, whereas update requests are serialized via a quorum-based protocol based on Paxos [11], e.g., Zab [9] or Raft [14]. Since reads are served locally, they can be somewhat stale but nevertheless represent a valid system state.

In the literature, these services' guarantees are described as linearizable, (or atomic), [7] writes and FIFO order for each client [1, 8]. This definition is not accurate in two ways: First, linearizability of updates has no meaning when no operation reads the written values. Second, this definition enables reading from a future write, which obviously does not occur in any real-world service. Our notion of OSC defines, for the first time, the precise guarantees of existing coordination services. OSC is stronger than sequential consistency, but weaker than linearizability.

Although supporting OSC semantics instead of atomicity of all operations enables fast local reads, this makes services *non-composable*: correct OSC coordination services may fail to provide the same level of consistency when combined [2, 13], as we exemplify in Section 4. Intuitively, the problem arises because OSC, similarly to sequential consistency [10], allows reads to occur "in the past", which can introduce cycles in the history of updates.

In a companion systems paper [13] we present ZooNet, a system for modular composition of coordination services, which addresses this challenge: Consistency is achieved on the client side by judiciously adding synchronization requests called *leading updates*. The key idea is to place a "barrier" that limits how far in the past reads can be served from. ZooNet does so by adding a "leading" update request prior to a read request whenever the read is addressed to a different service than the previous one accessed by the same client. We provide here the theoretical underpinnings for the algorithm implemented in ZooNet.

Proving the correctness of ZooNet is made possible by the OSC definition that we present in this paper. For example, had the coordination services allowed reads from the future, the composition would not have been correct. Interestingly, Vitenberg and Friedman [17] showed that sequential consistency, when combined with any local (i.e., composable) property continues to be non-composable. Our approach circumvents this impossibility result since leading updates is not a local property.

Updates have a key role in coordination services; for example, a shared lock is acquired by the first client to update the lock. Therefore, coordination services maintain a total order of updates that preserves their real-time order. In our formal definitions, the fact that strong consistency is required specifically for updates and not for reads plays no role. Indeed, we generalize our OSC definition to consider an arbitrary subset of an object's operations (not necessarily updates), and require the stronger consistency notion for this subset only. This generalization, called G-OSC, captures a range of criteria, including sequential consistency and linearizability.

We further model TSO [15, 16] as a composition of G-OSC objects, with strong consistency required for memory barriers. Thus, our result implies that TSO can be made sequentially consistent by adding leading memory barriers whenever a new object is accessed.

We organize the paper as follows: Section 2 presents our model. Section 3 provides background on coordination services, motivating our work. Section 4 defines OSC, and proves that OSC is not composable. Section 5 defines the leading updates property satisfied by adding update requests as in ZooNet, and shows that any composition of multiple OSC coordination services that preserves leading updates is OSC. Section 6 generalizes our notion of OSC to capture a range of criteria. Section 7 discusses the necessity of excluding "reads from the future". Section 8 concludes the paper.

2 Model and Notation

In this section we define the model and basic notions.

We use a standard shared memory execution model [7], where a set ϕ of sequential *processes* access shared *objects* from some set X. An object has a name label, a value, and a set of *operations* used for manipulating and reading its value. An operation's execution is delimited by two events, *invoke* and *response*.

A *history* σ is a sequence of operation invoke and response events. An invoke event of operation *op* is denoted i_{op}, and the matching response event is denoted r_{op}. For two events $e_1, e_2 \in \sigma$, we denote $e_1 <_\sigma e_2$ if e_1 precedes e_2 in σ, and $e_1 \leq_\sigma e_2$ if $e_1 = e_2$ or $e_1 <_\sigma e_2$. For two operations *op* and *op'* in σ, *op precedes op'*, denoted $op <_\sigma op'$, if $r_{op} <_\sigma i_{op'}$, and $op \leq_\sigma op'$ if $op = op'$ or $op <_\sigma op'$. Two operations are *concurrent* if neither precedes the other.

For a history σ, *complete(σ)* is the sequence obtained by removing operations with no response events from σ. A history is *sequential* if it begins with an invoke event and consists of an alternating sequence of invoke and response events, s.t. each invoke is followed by the matching response.

For $p \in \phi$, the *process subhistory* $\sigma|p$ of a history σ is the subsequence of σ consisting of events of process p. The *object subhistory* σ_x for an object $x \in$ X is similarly defined. A history σ is *well-formed* if for each process $p \in \phi$, $\sigma|p$ is sequential. For the rest of our discussion, we assume that all histories are well-formed. The order of operations in $\sigma|p$ is called the *process order of p*.

For the sake of our analysis, we assume that each subhistory σ_x starts with a dummy initialization update of x to a dedicated initial value v_0, denoted $di_x(v_0)$, and that there are no concurrent operations with $di_x(v_0)$ in σ_x.

The *sequential specification* of an object x is a set of allowed sequential histories in which all events are associated with x. For example, the sequential specification of a read-write object is the set of sequential histories in which each read operation returns the value written by the last update operation that precedes it.

In order to define OSC, for simplicity we classify operations into two categories: *update operations* that read and change the object's value (using any type of write, e.g., an unconditional write, a general read-modify-write, etc.), and *read operations* that only read the object value.

3 Coordination Services – Background

Coordination services are deployed on a collection of servers. In asynchronous systems, in order to overcome f failures, $2f + 1$ servers are required [12], and therefore the common deployment size is either three or five servers.

The servers maintain shared state that is accessible to client processes; each process is connected to one server at a time, and it performs operations on the shared state, (such as updates and reads), by sending requests to the server it is connected to. The updates' strong consistency is achieved by a protocol that allows a quorum, (typically a majority), of the servers

reach consensus on the next accepted update operation; examples of such protocols include Paxos [11] and its variants, e.g., Zab [9] and Raft [14]. Waiting for a majority to agree upon each request entails long latency, and therefore reads offer weaker semantics, which allow servers to immediately respond based on their local states.

In Figure 1 we see a coordination service that has three servers, and three processes, each connected to a different server. Process 3 sends to Server_1 a request to update x to be 1. The update propagates to Server_3. Since two out of three servers comprise a majority, they agree on the update, and both servers set $x = 1$. Process 1 reads x from Server_2 to which the update did not propagate yet, therefore the read returns 0. Thus, process 1 reads "from the past".

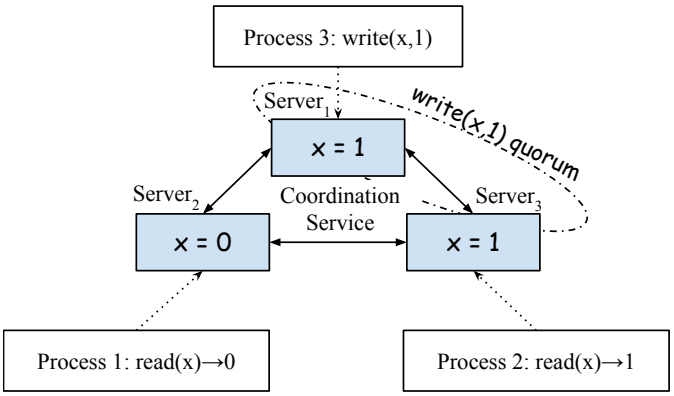

Figure 1: Coordination service with 3 servers and a shared state in which a value is assigned to x, serving 3 processes.

If a process switches servers during execution, (e.g., due to a connectivity problem, or a server failure), the coordination service maintains the process' execution correctness by not allowing it to connect to a server that is less up-to-date than the previous server of that process [8]. For example, if in Figure 1 process 2 disconnects from Server_3, it cannot connect to Server_2 until the update of x, (which process 2 saw), propagates to Server_2.

Each coordination service maintains multiple objects, e.g., ZooKeeper maintains a file-system-like structure, in which files and directories are objects. In the next section we define OSC, a correctness criterion satisfied for a collection of objects managed by the same service instance. However, when we combine multiple service instances, OSC is not guaranteed. Specifically, we show that such composition may lead to a scenario in which different processes see different update orders.

Nevertheless, there is strong motivation for composing coordination services, particularly in large scale deployments. Given that the update latency depends on communication between servers, if a coordination service runs over WAN, updates become slow. By offering a correct composition of coordination services, we enable local updates that are both fast and consistent; see ZooNet [13] for further discussion.

4 Ordered Sequential Consistency

In Section 4.1 we define ordered sequential consistency, and in Section 4.2 we prove that OSC is not composable.

4.1 Definition

We now define our correctness criterion for coordination services.

Definition 1 (OSC)**.** *A history σ is* ordered sequentially consistent *if there exists a history σ' that can be created by adding zero or more response events to σ, and there is a sequential permutation π of complete(σ'), called a* serialization *of σ, satisfying the following:*

OSC$_1$ (sequential specification): $\forall x \in \mathrm{X}$*, π_x belongs to the sequential specification of x.*

OSC$_2$ (process order): *For two operations o and o', if $\exists p \in \phi : o <_{\sigma|p} o'$ then $o <_\pi o'$.*

OSC$_3$ (real-time order of updates): $\forall x \in \mathrm{X}$*, for an update operation uo and an operation o s.t. $uo, o \in \sigma_x$, if $o <_\sigma uo$ then $o <_\pi uo$.*

An object is OSC if all of its histories are OSC.

Intuitively, OSC_3 does not rule out "reads from the past", in the sense that a read may return an old value, (i.e., a read operation can be serialized after any update that precedes it in σ). On the other hand, in contrast to the standard definition of sequential consistency, OSC_3 disallows "reads from the future" – a read may not return a value before it is written. The significance of the latter is illustrated in Section 7 below.

If we examine OSC's properties, we see that OSC_1 and OSC_3 are *local* properties – they are defined per object. In Section 4.2, we will show that OSC_2 is not local, in the sense that a composition of objects that (locally) satisfy OSC_2 does not necessarily (globally) satisfy OSC_2.

Note that like linearizability and sequential consistency, OSC defines a serialization π for every history, encompassing all objects. Due to OSC_3's real-time order of updates, OSC is stronger than sequential consistency, and since real-time order of reads is not required, it is weaker than linearizability.

4.2 Non-Composability

We now show that OSC is not a local property, i.e., not composable.

Theorem 1. *There exists a history of OSC objects that is not OSC.*

Proof. In Figure 2 we depict a history σ that is not OSC and consists of two OSC objects. Process 1 first writes 5 to x, and then reads from y "in the past", seeing y's initial value 0. Process 2 writes 5 to y and then reads 0 from x, which is x's initial value.

First we observe that σ_x (and by symmetry, also σ_y) satisfies OSC properties: σ_x has the following permutation:

$$\pi_x = di(x,0), (read(x) \to 0),\ write(x,5).$$

OSC_1: π_x belongs to the sequential specification of x.

OSC_2: there are no process order relations.

OSC_3: $di(x,0) <_\sigma write(x,5)$, and $di(x,0) <_{\pi_x} write(x,5)$ as needed. There is no additional requirement on the order $write(x,5)$ and $(read(x) \to 0)$.

Assume towards contradiction that a serialization π of σ exists. W.l.o.g. $write(x,5) <_\pi write(y,5)$. Given that $write(y,5) <_{\sigma|p_2} (read(x) \to 0)$, and that π satisfies OSC_2, then $write(y,5) <_\pi (read(x) \to 0)$. We get:

$$write(x,5) <_\pi write(y,5) <_\pi (read(x) \to 0),$$

a contraction to OSC_1.

In Theorem 1 and Figure 2, we showed the non-composability of two OSC services, one maintaining x and another maintaining y. If x and y belong to the same OSC service, then the scenario in Figure 2 does not arise, because OSC orders updates on all objects together.

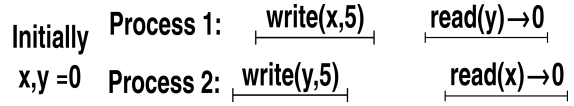

Figure 2: Inconsistent composition of two OSC objects x and y: there is no common serialization.

5 Composability via Leading Updates

In Section 5.1 we define a total order on operations in histories of compositions of OSC objects. This order always satisfies OSC_1 and OSC_3, but not necessarily OSC_2. In Section 5.2 we define the leading updates property, and prove that in histories that satisfy it, the total order defined in Section 5.1 does satisfy OSC_2. This implies that any composition of OSC objects that satisfies leading updates is OSC.

5.1 Π-Order on Operations

Given a history σ of OSC objects, we define a strict total order on all operations in σ. We begin by defining the future set of an update operation:

Definition 2 (Update future set)**.** *Given a history σ of OSC objects, a serialization π_x of σ_x, and an update operation $uo \in \sigma_x$, the* future set of uo in π_x *is* $F^{\pi_x}(uo) \triangleq \{o \in \pi_x | uo \leq_{\pi_x} o\}$.

We now define an update operation's first response event to be the earliest response event of an operation in its future set.

Definition 3 (First response event)**.** *Given a history σ of OSC objects, a serialization π_x of σ_x, and an update operation $uo \in \sigma_x$, the* first response event of uo in π_x*, denoted $fr_\sigma^{\pi_x}(uo)$, is the earliest response event in σ of an operation in $F^{\pi_x}(uo)$.*

We make two observations regarding first responses.

Observation 1. *Given a history σ of OSC objects and an update operation $uo \in \sigma_x$, for every serialization π_x of σ_x, $i_{uo} <_\sigma fr_\sigma^{\pi_x}(uo)$.*

Proof. By definition, $fr_\sigma^{\pi_x}(uo)$ is a response event in σ of an operation o s.t. $uo \leq_{\pi_x} o$. If $fr_\sigma^{\pi_x}(uo) <_\sigma i_{uo}$, i.e., $r_o <_\sigma i_{uo}$, then $o <_\sigma uo$, a contradiction to OSC_3.

We now observe that the order of updates in a serialization corresponds to the order of their first response events in that serialization:

Observation 2. *Let σ be a history of OSC objects, and let π_x be a serialization of σ_x for some x. For two update operations $o, o' \in \pi_x$, if $o <_{\pi_x} o'$, then $fr_\sigma^{\pi_x}(o) \leq_\sigma fr_\sigma^{\pi_x}(o')$.*

Proof. Since $o <_{\pi_x} o'$, we get $F^{\pi_x}(o') \subset F^{\pi_x}(o)$. By Definition 3, $fr_\sigma^{\pi_u}(o')$ is a response event of an operation $o_1 \in F^{\pi_x}(o')$, and therefore $o_1 \in F^{\pi_x}(o)$. Thus, $fr_\sigma^{\pi_x}(o)$ is either $fr_\sigma^{\pi_x}(o')$ or an earlier response event in σ.

To define our strict total order on operations we begin with updates:

Definition 4 (Update Π-order)**.** *Let σ be a history of OSC objects. Let $\Pi = \{\pi_x\}_{x \in \text{X}}$ be a set of serializations of $\{\sigma_x\}_{x \in \text{X}}$. For two update operations $uo_1, uo_2 \in \sigma$, we define their update Π-order, denoted $<_{u\Pi}$, as follows:*

(<) If $uo_1, uo_2 \in \sigma_x$ for some $x \in \text{X}$, then $uo_1 <_{u\Pi} uo_2$ iff $uo_1 <_{\pi_x} uo_2$;

(fr) otherwise, $\exists x, y \in \mathrm{X}, x \neq y : uo_1 \in \sigma_x$ *and* $uo_2 \in \sigma_y$*, then* $uo_1 <_{u\Pi} uo_2$ *iff* $fr_\sigma^{\pi_x}(uo_1) <_\sigma fr_\sigma^{\pi_y}(uo_2)$.

Lemma 1. *For a history* σ *of OSC objects and a set of serializations* $\Pi = \{\pi_x\}_{x\in\mathrm{X}}$ *of* $\{\sigma_x\}_{x\in\mathrm{X}}$*, update* Π*-order is a strict total order on updates in* σ*.*

Proof. Irreflexivity, antisymmetry, and comparability follow immediately from the definition of $<_{u\Pi}$. We show that $<_{u\Pi}$ satisfies transitivity.

Let uo_1, uo_2, and uo_3 be three update operations s.t. $uo_1 <_{u\Pi} uo_2 <_{u\Pi} uo_3$; we need to prove that $uo_1 <_{u\Pi} uo_3$. We consider four cases according to the condition by which each of the pairs is ordered:

(<,<) If $\exists x \in \mathrm{X}\ uo_1, uo_2, uo_3 \in \sigma_x$, then $uo_1 <_{\pi_x} uo_2 <_{\pi_x} uo_3$ implies $uo_1 <_{\pi_x} uo_3$, and thus $uo_1 <_{u\Pi} uo_3$.

(<,fr) If $\exists x, y \in \mathrm{X}, x \neq y : uo_1 <_{\pi_x} uo_2$, $uo_3 \in \sigma_y$, and $fr_\sigma^{\pi_x}(uo_2) <_\sigma fr_\sigma^{\pi_y}(uo_3)$, by Observation 2, $fr_\sigma^{\pi_x}(uo_1) \leq_\sigma fr_\sigma^{\pi_x}(uo_2)$, therefore $fr_\sigma^{\pi_x}(uo_1) <_\sigma fr_\sigma^{\pi_y}(uo_3)$, and $uo_1 <_{u\Pi} uo_3$.

(fr,<) If $\exists x, y \in \mathrm{X}, x \neq y : uo_1 \in \sigma_x$, $uo_2 <_{\pi_y} uo_3$, and $fr_\sigma^{\pi_x}(uo_1) <_\sigma fr_\sigma^{\pi_y}(uo_2)$, by Observation 2, $fr_\sigma^{\pi_y}(uo_2) \leq_\sigma fr_\sigma^{\pi_y}(uo_3)$. We get $fr_\sigma^{\pi_x}(uo_1) <_\sigma fr_\sigma^{\pi_y}(uo_3)$, therefore $uo_1 <_{u\Pi} uo_3$.

(fr,fr) If $\exists x, y, z \in \mathrm{X}, x \neq y, y \neq z : uo_1 \in \sigma_x$, $uo_2 \in \sigma_y$, and $uo_3 \in \sigma_z$, this means that $fr_\sigma^{\pi_x}(uo_1) <_\sigma fr_\sigma^{\pi_y}(uo_2)$ and $fr_\sigma^{\pi_y}(uo_2) <_\sigma fr_\sigma^{\pi_z}(uo_3)$. By transitivity of $<_\sigma$, $fr_\sigma^{\pi_x}(uo_1) <_\sigma fr_\sigma^{\pi_z}(uo_3)$. If $z \neq x$, then $uo_1 <_{u\Pi} uo_3$. If $z = x$, by the contrapositive of Observation 2, $uo_1 <_{\pi_x} uo_3$, and $uo_1 <_{u\Pi} uo_3$.

We extend $<_{u\Pi}$ to a weak total order in the usual way: $o_1 \leq_{u\Pi} o_2$ if $o_1 <_{u\Pi} o_2$ or $o_1 = o_2$. For a history σ, a serialization π_x of σ_x, and an operation o in σ_x, the *last update before* o *in* π_x, denoted $lu_{\pi_x}(o)$, is the latest update operation in the prefix of π_x that ends with o. Note that since every history starts with a dummy initialization, every read is preceded by at least one update and so $lu_{\pi_x}(o)$ is well-defined. We use last updates to extend the update Π-order to a strict total order on all operations in σ.

Definition 5 (Π-order)**.** *Let* σ *be a history of OSC objects. Let* $\Pi = \{\pi_x\}_{x\in\mathrm{X}}$ *be a set of serializations of* $\{\sigma_x\}_{x\in\mathrm{X}}$*. For two operations* $o_1, o_2 \in \sigma$ *on objects* x*,* y*, resp., we define* Π*-order, denoted* $<_\Pi$*, as follows:*

($lu_{\pi_x}(o_1) \neq lu_{\pi_y}(o_2)$*) if the last updates before* o_1 *and* o_2 *are different, then* $o_1 <_\Pi o_2$ *iff* $lu_{\pi_x}(o_1) <_{u\Pi} lu_{\pi_y}(o_2)$*;*

($lu_{\pi_x}(o_1) = lu_{\pi_y}(o_2)$*) otherwise,* $x = y$*, and* $o_1 <_\Pi o_2$ *iff* $o_1 <_{\pi_x} o_2$*.*

We now observe that $<_\Pi$ generalizes all the serializations $\pi_x \in \Pi$:

Observation 3. *Let* σ *be a history of OSC objects, and* $\pi_x \in \Pi$ *a serialization of* σ_x *for some* x*. For two operations* $o_1, o_2 \in \pi_x$*, if* $o_1 <_{\pi_x} o_2$ *then* $o_1 <_\Pi o_2$*.*

Proof. Since $o_1 <_{\pi_x} o_2$, then $lu_{\pi_x}(o_1) \leq_{\pi_x} lu_{\pi_x}(o_2)$. If $lu_{\pi_x}(o_1) = lu_{\pi_x}(o_2)$ then by Definition 5, $o_1 <_\Pi o_2$. Otherwise, by Definition 4, $lu_{\pi_x}(o_1) <_{u\Pi} lu_{\pi_x}(o_2)$ and by Definition 5, $o_1 <_\Pi o_2$.

Lemma 2. *For a history* σ *of OSC objects,* Π*-order is a strict total order on all operations.*

Proof. Irreflexivity, antisymmetry, and comparability follow immediately from the definition of $<_\Pi$. We show that $<_\Pi$ satisfies transitivity.

Let o_1, o_2, and o_3 be three operations on objects x, y, z, resp., s.t. $o_1 <_\Pi o_2 <_\Pi o_3$; we need to prove that $o_1 <_\Pi o_3$.

For every o_i and o_j, by Definition 5, $o_i <_\Pi o_j$ implies $lu_{\pi_i}(o_i) \leq_{u\Pi} lu_{\pi_j}(o_j)$. By transitivity of $\leq_{u\Pi}$ (Lemma 1), we get from $lu_{\pi_x}(o_1) \leq_{u\Pi} lu_{\pi_y}(o_2) \leq_{u\Pi} lu_{\pi_z}(o_3)$ that $lu_{\pi_x}(o_1) \leq_{u\Pi} lu_{\pi_z}(o_3)$.

If $lu_{\pi_x}(o_1) <_{u\Pi} lu_{\pi_z}(o_3)$ then by Definition 5 $o_1 <_\Pi o_3$. If $lu_{\pi_x}(o_1) = lu_{\pi_z}(o_3)$, then by $lu_{\pi_x}(o_1) \leq_{u\Pi} lu_{\pi_y}(o_2) \leq_{u\Pi} lu_{\pi_z}(o_3)$ we get $lu_{\pi_x}(o_1) = lu_{\pi_y}(o_2) = lu_{\pi_z}(o_3)$, and $x = y = z$. Therefore by $o_1 <_\Pi o_2 <_\Pi o_3$ and Definition 5, $o_1 <_{\pi_x} o_2 <_{\pi_x} o_3$, and thus by Definition 5 $o_1 <_\Pi o_3$.

Note that Π-order is always defined for compositions of OSC objects. Since it generalizes all the serializations π_x (Observation 3), it preserves OSC_1 and OSC_3. Nevertheless, OSC_2 is not guaranteed. For example, consider the history σ depicted in Figure 2, for which

$$\pi_x = di(x,0), (read(x) \to 0),\ write(x,5),\ \text{and}\ \pi_y = di(y,0), (read(y) \to 0),\ write(y,5).$$

The first response event of each update operation is the response event of that update in σ. Given that $r_{di(x,0)} <_\sigma r_{di(y,0)}$, we get

$$di(x,0) <_\Pi (read(x) \to 0) <_\Pi di(y,0) <_\Pi (read(y) \to 0) <_\Pi write(y,5) <_\Pi write(x,5).$$

This means that $<_\Pi$ does not maintain the process order as required for OSC_2. In the next section we show that by adding leading updates, we ensure that Π-order satisfies also OSC_2.

5.2 OSC Composition via Leading Updates

We say that in a history σ there are *leading updates* if for every read operation ro by a process p in σ, the last operation of p before ro is on the same object. This means that between every two read operations of different objects by the same process in σ, there is an update operation to the second.

In ZooNet [13], this property is achieved by adding dummy updates whenever a client reads from a different object than the one it last accessed. This is illustrated in Figure 3, where we add dummy updates to the example of Figure 2.

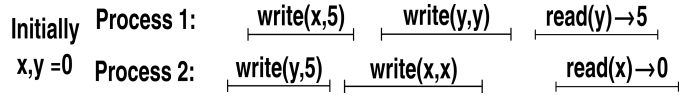

Figure 3: Consistent composition of two objects x and y: adding dummy updates prior to reads on new objects ensures OSC.

The history in Figure 3 is OSC with the following serialization:

$$di(x,0),\ di(y,0),$$

$$write(y,5),\ write(x,x),\ read(x) \to 0, write(x,5),\ write(y,y),\ read(y) \to 5.$$

We next prove that adding leading updates allows for correct OSC composition.

Theorem 2. *If a history σ of OSC objects has leading updates, then σ is OSC.*

Proof. Let $\Pi = \{\pi_x\}_{x \in \text{X}}$ be a set of serializations of $\{\sigma_x\}_{x \in \text{X}}$, and let π be the sequential permutation of σ defined by $<_\Pi$. We now prove that π satisfies OSC.
OSC_1 and **OSC_3** follow immediately from Observation 3.

We prove $\mathbf{OSC}_2$. Let o_1 and o_2 be two operations for which $\exists p \in \phi : o_1 <_{\sigma|p} o_2$. We now show that $o_1 <_\Pi o_2$.

We start by proving the claim for two consecutive operations in $\sigma|p$. If both operations are on the same object, then by Observation 3, $o_1 <_\Pi o_2$, as needed. Otherwise, $\exists x, y \in \mathrm{X}, x \neq y : o_1 \in \sigma_x, o_2 \in \sigma_y$, and o_1 immediately precedes o_2 in $\sigma|p$. By leading updates, since o_1 and o_2 are not on the same object, o_2 is an update operation and hence $lu_{\pi_y}(o_2) = o_2$.

By definition, $fr_\sigma^{\pi_x}(lu_{\pi_x}(o_1)) \leq_\sigma r_{o_1}$. Since $r_{o_1} <_\sigma i_{o_2}$, and by Observation 1, $i_{o_2} <_\sigma fr_\sigma^{\pi_y}(o_2)$, we get that $fr_\sigma^{\pi_x}(lu_{\pi_x}(o_1)) <_\sigma fr_\sigma^{\pi_y}(o_2)$. By Definition 4, $lu_{\pi_x}(o_1) <_{u\Pi} o_2$, and by Definition 5, $o_1 <_\Pi o_2$.

Thus, every two consecutive operations o^i, o^{i+1} in $\sigma|p$ satisfy $o^i <_\Pi o^{i+1}$. By Lemma 2, $<_\Pi$ is a strict total order on all operations, and therefore by transitivity, we get $o_1 <_\Pi o_2$.

6 Generalization and Relationship to other Conditions

In Section 6.1 we define G-OSC, a generalization of OSC. In Section 6.2 we show that sequential consistency and linearizability are special cases of G-OSC, and in Section 6.3 we show that a composition of G-OSC objects can represent TSO.

6.1 Generalizing OSC

The OSC definition differentiates between update and read operations, requiring real-time order only for the former. The focus on updates follows their role in coordination services, which maintain a total order of updates that preserves their real-time order. Nevertheless, in the formal definitions, the fact that the real-time ordered set of operations happens to be updates plays no role. We observe that we can choose an arbitrary subset A of the objects' operations instead of updates. Note that A is a subset of the operations defined on all objects in X, which are not necessarily the same for all objects. OSC can then be generalized as follows:

Definition 6 (G-OSC(A))**.** *A history σ is* general OSC w.r.t. a subset A of the objects' operations *if there exists a history σ' that can be created by adding zero or more response events to σ, and there is a sequential permutation π of complete(σ'), called a* serialization *of σ, satisfying the following:*

> *OSC*$_1$ (sequential specification), *as in Definition 1.*
>
> *OSC*$_2$ (process order), *as in Definition 1.*
>
> *G-OSC*$_3$ (real-time order of A): $\forall x \in \mathrm{X}$*, for an operation $o \in A$ and an operation o' s.t. $o, o' \in \sigma_x$, if $o' <_\sigma o$ then $o' <_\pi o$.*

6.2 G-OSC, Linearizability, and Sequential Consistency

Linearizability and sequential consistency are both special cases of G-OSC. For linearizability, we let A consist of all of the objects' operations. On the other hand, if $A = \emptyset$, then OSC_3 is null, and G-OSC only follows the sequential specification and process order of an object. Thus, sequential consistency is simply G-OSC($\emptyset$).

Our composition is based on adding leading A-s. In case $A = \emptyset$, i.e., in sequentially consistent objects, one needs to add a special "sync" operation, which is not exposed to clients and is used for allowing correct composition. In addition, we assume that each subhistory σ_x starts with a dummy initialization operation that belongs to A. Similarly to sync operations, such dummy updates are not exposed to clients, hence in the object's public interface, A can remain $\emptyset$.

6.3 G-OSC and TSO

TSO: background TSO ("Total Store Order") [15, 16] is a widely used memory model, for example, implemented in x86 processors.

TSO models a system where each process' updates are stored in a local buffer, and applied on the shared memory at some point in the future. A read of object x then returns either the value of the most recent buffered update on x by the same process, or x's value from the shared memory, in case there is no such buffered operation. TSO is not sequentially consistent because it allows read operations to be served by shared memory prior to applying updates that precede them in process order. That is, for a history σ, an update operation o_1 on x, and a read operation o_2 on $y \neq x$, $o_1 <_{\sigma|p} o_2$ does not imply that o_1 is applied on the shared memory before o_2 reads from the shared memory. This contradicts the process order, and thus contradicts sequential consistency.

For example, the history depicted in Figure 2 satisfies TSO – it is possible for $write(x, 5)$ of Process 1 to respond before x is set to 5 in shared memory, and symmetrically for $write(y, 5)$ to respond before y is set to 5. Both processes then read the old values – Process 1 reads $y = 0$, and Process 2 reads $x = 0$.

Since TSO diverges from sequential consistency only due to operations on different objects, each object x in TSO by itself satisfies sequential consistency. In addition, for cases in which a process needs to force process order on its updates and reads on different objects, TSO defines special *memory barrier* operations, that prevent such reordering. For simplicity, we focus our discussion on the general memory barrier. This barrier guarantees that all prior operations of the same process are applied to shared memory before the barrier.

Modeling TSO using G-OSC We observe that a single TSO object can be modeled as G-OSC({barrier}): it is sequentially consistent and enforces real-time order between every barrier and every other operation. A composition of TSO objects is therefore, in particular, a composition of G-OSC({barrier}) objects. By modeling TSO this way we can conclude that adding leading barriers every time a process performs an operation on a new object ensures correct TSO composition:

Corollary 1. *A composition of TSO objects with leading barriers is sequentially consistent.*

7 Why not Read from the Future?

OSC_3 requires that the serialization order between an update and any other operation follow their real-time order. Since the serialization satisfies the object's sequential specification, this disallows "reads from the future".

Requiring real-time order only between update operations would cause two problems: first, it will allow impractical histories; second, it would give rise to the anomaly depicted in Figure 4, where process 1 reads $y = 5$ and then updates x to 5, while process 2 reads $x = 5$ and only then updates y to 5, creating a non-serializable dependency cycle. In this case, adding updates anywhere in the history does not guarantee a correct composition.

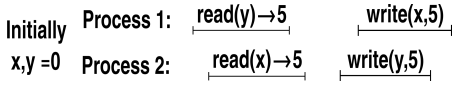

Figure 4: Inconsistent composition of two objects x and y where reading from the future is possible. Note that adding updates to the history cannot make it consistent.

8 Conclusions

In this work we defined OSC, which accurately captures the semantics of coordination services, which are broadly deployed nowadays in backends of large-scale distributed systems. OSC does not require reads to maintain their real-time order, and so OSC is a weaker property than linearizability, and like sequential consistency, non-composable in itself. Non-composability precludes multi-data-center deployments that are both consistent and efficient. We showed a way to compose OSC objects correctly using a simple non-local property called leading updates. Composability of coordination services enables low-latency local updates, while having global consistency among services.

In addition, we showed that OSC can be generalized, and that G-OSC captures a range of criteria, including sequential consistency, linearizability and TSO; the latter implies a sufficient condition on adding memory barriers so as to ensure sequential consistency.

Acknowledgments

We thank Alexey Gotsman for helpful comments on an earlier draft. Kfir Lev-Ari is supported in part by the Hasso-Plattner Institute (HPI) Research School. This work was partially supported by the Israeli Ministry of Science.

References

[1] etcd3 API - Guarantees Provided. `https://coreos.com/etcd/docs/latest/api_v3.html`. [Online; accessed 1-June-2016].

[2] Camille Fournier: Building a Global, Highly Available Service Discovery Infrastructure with ZooKeeper. `http://whilefalse.blogspot.co.il/2012/12/building-global-highly-available.html`, 2016. [Online; accessed 1-Jan-2016].

[3] Consul – a tool for service discovery and configuration. Consul is distributed, highly available, and extremely scalable. `https://www.consul.io/`, 2016. [Online; accessed 1-Jan-2016].

[4] Doozer – a highly-available, completely consistent store for small amounts of extremely important data. `https://github.com/ha/doozerd`, 2016. [Online; accessed 1-Jan-2016].

[5] etcd – a highly-available key value store for shared configuration and service discovery. `https://coreos.com/etcd/`, 2016. [Online; accessed 1-Jan-2016].

[6] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1298455.1298487`.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. `doi:10.1145/78969.78972`.

[8] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA,

USA, 2010. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[9] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society. URL: http://dx.doi.org/10.1109/DSN.2011.5958223, doi:10.1109/DSN.2011.5958223.

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. doi:10.1109/TC.1979.1675439.

[11] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. URL: http://doi.acm.org/10.1145/279227.279229, doi:10.1145/279227.279229.

[12] Leslie Lamport. Lower bounds for asynchronous consensus. Technical report, July 2004. URL: https://www.microsoft.com/en-us/research/publication/lower-bounds-for-asynchronous-consensus/.

[13] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/lev-ari.

[14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[15] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_27, doi:10.1007/978-3-642-03359-9_27.

[16] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[17] Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings*, pages 92–105, 2003. URL: http://dx.doi.org/10.1007/978-3-540-39989-6_7, doi:10.1007/978-3-540-39989-6_7.