Ordered Sequential Consistency: Composition and Correctness

Kfir Lev-Ari

Ordered Sequential Consistency: Composition and Correctness

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Kfir Lev-Ari

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5777 Haifa July 2017

This research was carried out under the supervision of Prof. Idit Keidar, in the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion.

The results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period. The most up-to-date versions of which are found in this thesis.

The generous financial help of the Technion is gratefully acknowledged.

Acknowledgements

First, I would like to thank my advisor, Prof. Idit Keidar. Thank you for being *highly available* (24/7), for *tolerating my faults*, and for the *consistent* guidance throughout the inconsistent world of distributed systems. I am more than grateful for the opportunity to work under your supervision. It was an awesome (*Paxos*-like) journey, in which I got the chance to grasp (and *accept*) new concepts, to *propose* new ideas, to *learn* a lot from others, and to *lead* the way when I was ready. To the anonymous reader, note that there is a *consensus* that Idit is the best advisor one could have.

I would also like to thank Edward Bortnikov, Prof. Gregory Chockler, and Alexander Shraer, for very productive collaborations, as well as for the great time we had together on many different occasions, such as internships, conferences, work-related-and-not-related meetings, avocados, and one crazy cab drive back from the airport. Whenever I've talked with you, you guys were always willing to share thoughts based on your experience in order to help me get things right, as well as to avoid obstacles that you stumble upon in your journey (whether related or unrelated to my PhD), and I thank you for that. Eventually, we've managed to *compose* great things together ☺ .

I thank the super-smart-crazy-these-guys-scare-me members of Idit's research group: Naama Kraus, Noam Shalev, Alexander (Sasha) Spiegelman, Dani Shaket, Hagar Porat, Itay Tsabary, and Alon Berger, (as well as for past members who I had the pleasure to meet), for their valuable comments, suggestions, water cooler talks, and (mainly) shared complaints. Specially, I wish to thank Dani and Naama for introducing me to the group, and I thank Sasha for our joint efforts while working on a cool yet unpublished paper. I would like to thank other Technion PhD students who I had the pleasure to learn from and speak with, such as Maya Arbel, and my office roommates Ofir Shwartz and Aran Bergman. I had very interesting conversations with you all along the way. You guys really rock! (and jazz)

Last but not least, I would like to thank my family. I thank my father and mother for always being supportive, saying (prior to my bachelor degree) that I can go and learn whatever I want, but they'll help me financially only if it'll be computer science, law, or medicine 😄. I thank all the grandparents of Adam, Liam, and Ethan, for helping us with them while I'm working around the clock in order to meet (yet another) deadline. I thank my wonderful and special wife Anat, for supporting me in every step of the way and for believing in me.

List of Publications

Journal Paper

1. Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Composing ordered sequential consistency. *Inf. Process. Lett.*, 123:47–50, 2017. doi: 10.1016/j.ipl.2017.03.004. URL `https://doi.org/10.1016/j.ipl.2017.03.004`

Conference Papers

1. Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, pages 251–264, 2016. URL `https://www.usenix.org/conference/atc16/technical-sessions/presentation/lev-ari`

2. Kfir Lev-Ari, Gregory Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 356–370, 2015. doi: 10.1007/978-3-662-48653-5_24. URL `https://doi.org/10.1007/978-3-662-48653-5_24`

3. Kfir Lev-Ari, Gregory Chockler, and Idit Keidar. On correctness of data structures under reads-write concurrency. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 273–287, 2014. doi: 10.1007/978-3-662-45174-8_19

4. Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 739–753, 2016. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya` **(Note: not included in the thesis)**

Contents

List of Figures

Abstract

We define ordered sequential consistency (OSC), a generic criterion for concurrent objects. We show that OSC encompasses a range of criteria, from sequential consistency to linearizability, and captures the typical behavior of real-world coordination services, such as ZooKeeper. A straightforward composition of OSC objects is not necessarily OSC, e.g., a composition of sequentially consistent objects is not sequentially consistent. We define a global property we call leading ordered operations, and prove that it enables correct OSC composition.

A direct implication of our OSC composition global property is the ability to compose coordination services. Such services, like ZooKeeper, etcd, Doozer, and Consul are increasingly used by distributed applications for consistent, reliable, and high-speed coordination. When applications execute in multiple geographic regions, coordination service deployments trade-off between performance, (achieved by using independent services in separate regions), and consistency. We present a system design for modular composition of services that addresses this trade-off. We implement ZooNet, a prototype of this concept over ZooKeeper. ZooNet allows users to compose multiple instances of the service in a consistent fashion, facilitating applications that execute in multiple regions. In ZooNet, clients that access only local data suffer no performance penalty compared to working with a standard single ZooKeeper. Clients that use remote and local ZooKeepers show up to 7x performance improvement compared to consistent solutions available today.

Linearizability, (which is a special case of OSC), is one of the most common correctness criteria for shared objects. We present a comprehensive methodology for proving linearizability and related criteria of concurrent data structures. We exemplify our methodology by using it to give a roadmap for proving linearizability of the popular Lazy List implementation of the concurrent set abstraction. Correctness is based on our key theorem, which captures sufficient conditions for linearizability. In contrast to prior work, our conditions are derived directly from the properties of the data structure in sequential runs, without requiring the linearization points to be explicitly identified.

Abbreviations and Notations

RCU	:	read-copy-update
RTO	:	real time order
CAS	:	compare-and-swap
RO	:	read-only
OSC	:	ordered sequential consistency

Chapter 1

Introduction

The common design nowadays for backends of large-scale applications is built upon using external services as building blocks. Due to the distributed nature of these applications, (i.e., multiple servers that serve multiple clients concurrently), one of the most frequent and important building blocks in use is one that is responsible for maintaining correct executions. E.g., an online application that sells concert tickets uses a global lock service to guarantee that no ticket is sold twice. We call these global services "coordination services".

As the scale of applications increases and crosses the data-center boundaries, (e.g., serving clients all over the world via multiple data-centers), application designers face performance challenges that arise from the physical distance between the backend servers, (e.g., what is the best global lock servers' deployment alternative w.r.t. the application servers locations?). Addressing these challenges often requires knowledge of the building blocks' internals, and in the case of coordination services' internals, answering this question is far from being trivial.

The goal of this thesis is to simplify reasoning about concurrent algorithms' correctness, with a focus on coordination services and their performance on global scale deployment.

1.1 Thesis Structure

The presented thesis collates our published papers. The first two papers present our base-point analysis approach for proving correctness of concurrent algorithms, and the last two papers present our ordered sequential consistency (OSC) definition, and implications on real-world services.

The first paper introduces the foundations of our base-point analysis approach for proving correctness of concurrent data structures. In addition, we define regular data structures, and state sufficient conditions for regularity, linearizability, and sequential consistency, (all w.r.t. base points of read-only operations). Last, we prove the correctness of Lazy List

under reads-write concurrency.

In the second paper we generalize base-point analysis to any type of data structure and provide a constructive road-map for proving data structures' linearizability. We exemplify our approach using the same Lazy List, this time addressing writes concurrency as well.

The third paper presents a design, an implementation, and an evaluation of a consistent composition of ZooKeepers, which we call ZooNet. In ZooNet, clients that access only local data suffer no performance penalty compared to working with a standard single ZooKeeper. Clients that use several ZooKeeper instances, i.e., remote and local ZooKeepers, show up to 7.5x performance improvement compared to consistent solutions available today.

The fourth paper presents OSC, a criterion that generalizes linearizability and sequential consistency. In addition, we define a global property called leading A operations, and prove that OSC objects can be composed via leading As.

1.2 Brief Scientific Background

1.2.1 Model for Analyzing Concurrent Objects

We use a standard shared memory execution model [49], 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*. For example, a read-write register r is an object with an initial value 0, and two operations: (1) $get(r) \rightarrow 0$ - returns r's value; and (2) $set(r,5)$ - modifies r's value.

A *history* h 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 $e_1, e_2 \in h$, we denote $e_1 <_h e_2$ if e_1 precedes e_2 in h, and $e_1 \leq_h e_2$ if $e_1 = e_2$ or $e_1 <_h e_2$.

For two operations op and op' in h, *op precedes op'*, denoted $op <_h op'$, if $r_{op} <_h i_{op'}$, and $op \leq_h op'$ if $op = op'$ or $op <_h op'$. Two operations are *concurrent* if neither precedes the other.

For a history h, *complete(h)* is the sequence obtained by removing all operations with no response events from h. 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. See Figure 1.1 for history examples.

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

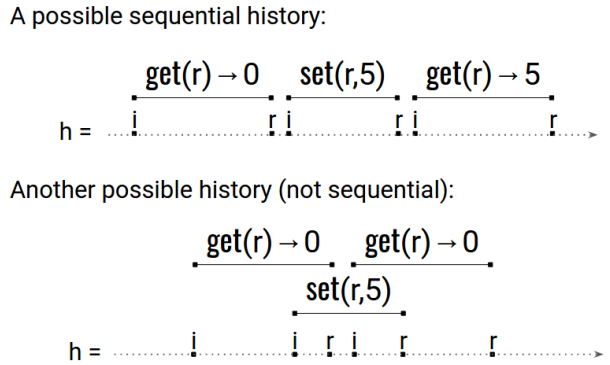


Figure 1.1: Possible histories of a read-write register r.

We refer to an operation that changes the object's value as an *update operation*. In order to simplify the discussion of initialization, we assume that every history begins with a dummy (initializing) update operation. A *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 or the initial value if no such operation exists, as depicted in Figure 1.2

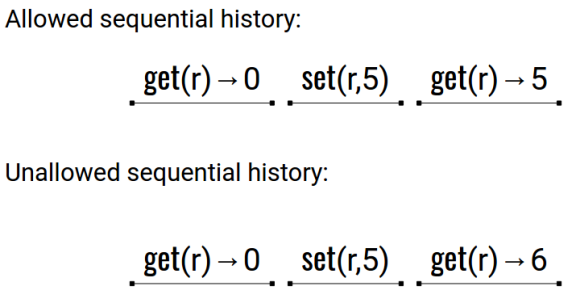


Figure 1.2: A history of a read-write register r belongs to r's sequential specification if each read operation in the history returns the last written value.

A *sequential permutation* π of an history h is a sequential history such that (1) all objects in π start with the same initial value as in h; and (2) π consists of the same operations as h.

1.2.2 Sequential Consistency

A history h is *sequentially consistent* [53] if there exists a history h' that can be created by adding zero or more response events to h, and there is a sequential permutation π of complete(h'), such that:

1. for every object $x \in \pi$, π_x belongs to the sequential specification of x.
2. for two operations o and o', if $\exists p \in \phi : o <_{h|p} o'$ then $o <_\pi o'$. I.e., every pair of operations that belong to the same process, appear in the same order in h and in π.

A sequentially consistent history example is depicted in Figure 1.3, a non-sequential

consistent history in Figure 1.4.

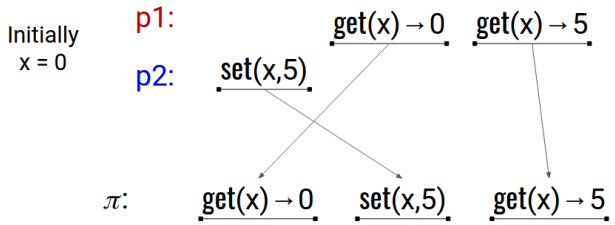


Figure 1.3: A sequentially consistent history h of two processes p_1 and p_2. π is a sequential permutation that belongs to x's sequential specification, in which the program order is identical to the order in h. h is not linearizable because every sequential permutation that follows the operations' real time order contradicts the sequential specification of x.

An object x is *sequentially consistent* if for every history h of x, h is sequentially consistent.

1.2.3 Linearizability

A history h is *linearizable* [49] if there exists a history h' that can be created by adding zero or more response events to h, and there is a sequential permutation π of complete(h'), such that:

1. for every object $x \in \pi$, π_x belongs to the sequential specification of x.
2. for two operations o and o', if $o <_h o'$ then $o <_\pi o'$. I.e., every pair of operations that are not interleaved in h, appear in the same order in h and in π.

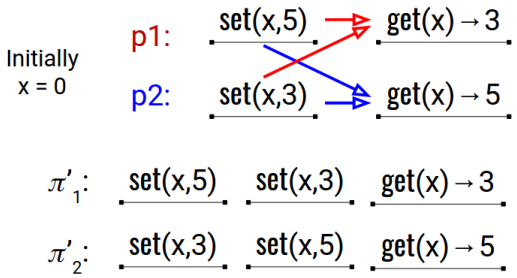


Figure 1.4: The history h of two processes p_1 and p_2 is not sequentially consistent, since there is no sequential permutation that belongs to the sequential specification of x, in which the program order is identical to the order in h (the colored arrows indicate the required sequential ordering).

See Figure 1.5 for an example of linearizable history, and Figure 1.3 for a history that is not linearizable. An object x is *linearizable*, also called *atomic*, if for every history h of x, h is linearizable.

1.2.4 Coordination Services

Coordination services are used for maintaining shared state in a consistent and fault-tolerant manner. Fault tolerance is achieved using replication, which is usually done by running a

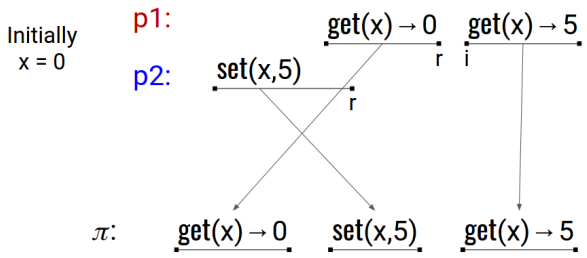


Figure 1.5: A linearizable history of two processes p_1 and p_2. π is a sequential permutation that belongs to x sequential specification, in which the operations real time order is identical to the order in h.

quorum-based state-machine replication protocol such as Paxos [55] or its variants [51, 68].

In Paxos, the history of state updates is managed by a set of servers called *acceptors*, s.t. every update is voted on by a quorum (majority) of acceptors (as depicted in Figure 1.6). One acceptor serves as *leader* and manages the voting process. In addition to acceptors, Paxos has *learners*, which are light-weight services that do not participate in voting and get notified of updates after the quorum accepts them.

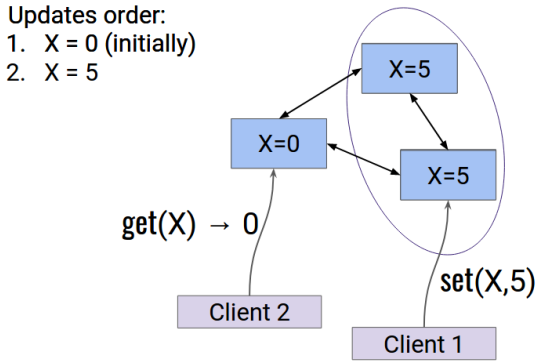


Figure 1.6: Example of two clients using a coordination service concurrently. Client 1 sets the shared variable x's value to 5, and Client 2 reads x. A majority of acceptors is required for the update to take place, whereas reads are served locally.

Coordination services are typically built on top of an underlying key-value store and offer read and update (read-modify-write) operations. The updates are linearizable, i.e., all acceptors and learners see the same sequence of updates and this order conforms to the real-time order of the updates. The read operations are sequentially consistent. A client can thus read a stale value that has already been overwritten by another client (as shown in Figure 1.7). These weaker semantics are chosen in order to allow every learner or acceptor to serve reads locally.

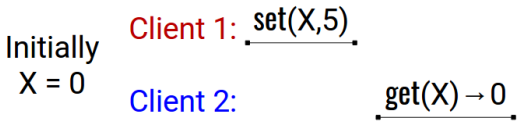


Figure 1.7: A possible history of a coordination service x. Client 2 "reads from the past", by not seeing the update of Client 1. This is possible since reads are served locally.

Chapter 2

Paper: On Correctness of Data Structures under Reads-Write Concurrency

Kfir Lev-Ari, Gregory V. Chockler, Idit Keidar: "On Correctness of Data Structures under Reads-Write Concurrency". Distributed Computing: 28th International Symposium DISC 2014, Austin, TX, USA, October 12-15, 2014, Proceedings. ed. / Fabian Kuhn. Vol. 8784 1. ed. Springer-Verlag Berlin Heidelberg, 2014. p. 273-287.

In this paper, we create a base-point analysis approach for proving correctness of concurrent data structures, define regular data structures, and prove the correctness of Lazy List under this type of concurrency.

On Correctness of Data Structures under Reads-Write Concurrency *

Kfir Lev-Ari[1], Gregory Chockler[2], and Idit Keidar[1]

[1] *Viterbi Department of Electrical Engineering, Technion, Haifa, Israel*

[2] *CS Department, Royal Holloway University of London, Egham, UK*

Abstract

We study the correctness of shared data structures under reads-write concurrency. A popular approach to ensuring correctness of read-only operations in the presence of concurrent update, is read-set validation, which checks that all read variables have not changed since they were first read. In practice, this approach is often too conservative, which adversely affects performance. In this paper, we introduce a new framework for reasoning about correctness of data structures under reads-write concurrency, which replaces validation of the entire read-set with more general criteria. Namely, instead of verifying that all read shared variables still hold the values read from them, we verify abstract conditions over the shared variables, which we call *base conditions*. We show that reading values that satisfy some base condition at every point in time implies correctness of read-only operations executing in parallel with updates. Somewhat surprisingly, the resulting correctness guarantee is not equivalent to linearizability, rather, it can express a range of conditions. Here we focus on two new criteria: *validity* and *regularity*. Roughly speaking, the former requires that a read-only operation never reaches a state unreachable in a sequential execution; the latter generalizes Lamport's notion of regularity for arbitrary data structures, and is weaker than linearizability. We further extend our framework to capture also linearizability and sequential consistency. We illustrate how our framework can be applied for reasoning about correctness of a variety of implementations of data structures such as linked lists.

*This work was partially supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by the Israeli Ministry of Science, by a Royal Society International Exchanges Grant, and by the Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research Program.

2.1 Introduction

Motivation Concurrency is an essential aspect of computing nowadays. As part of the paradigm shift towards concurrency, we face a vast amount of legacy sequential code that needs to be parallelized. A key challenge for parallelization is verifying the correctness of the new or transformed code. There is a fundamental tradeoff between generality and performance in state-of-the-art approaches to correct parallelization. General purpose methodologies, such as transactional memory [48, 74] and coarse-grained locking, which do not take into account the inner workings of a specific data structure, are out-performed by hand-tailored fine-grained solutions [66]. Yet the latter are notoriously difficult to develop and verify. In this work, we take a step towards mitigating this tradeoff.

It has been observed by many that correctly implementing concurrent modifications of a data structure is extremely hard, and moreover, contention among writers can severely hamper performance [71]. It is therefore not surprising that many approaches do not allow write-write concurrency; these include the *read-copy-update (RCU)* approach [64], flat-combining [47], coarse-grained readers-writer locking [38], and pessimistic software lock-elision [23]. It has been shown that such methodologies can perform better than ones that allow write-write concurrency, both when there are very few updates relative to queries [64] and when writes contend heavily [47]. We focus here on solutions that allow only read-read and read-write concurrency.

A popular approach to ensuring correctness of read-only operations in the presence of concurrent updates, is *read-set validation*, which checks before the operation returns that no shared variables have changed since they were first read by the operation. In practice, this approach is often too conservative, which adversely affects performance. For example, when traversing a linked list, it suffices to require that the last read node is connected to the rest of the list; there is no need to verify the values of other traversed nodes, since the operation no longer depends on them. In this paper, we introduce a new framework for reasoning about correctness of concurrent data structures, which replaces validation of the entire read-set with more general conditions: instead of verifying that all read shared variables still hold the values read from them, we verify abstract conditions over the variables. These are captured by our new notion of *base conditions*.

Roughly speaking, a *base condition* of a read-only operation at time t, is a predicate over shared variables, (typically ones read by the operation), that captures the local state the operation has reached at time t. Base conditions are defined over sequential code. Intuitively, they represent invariants that the read-only operation relies upon in sequential executions. We show that the operation's correctness in a concurrent execution depends on whether these invariants are preserved by update operations executed concurrently with the read-only

one. We capture this formally by requiring each state in every read-only operation to have a *base point* of some base condition, that is, a point in the execution where the base condition holds. In the linked list example – it does not hurt to see old values in one section of the list and new ones in another section, as long as we read every next pointer consistently with the element it points to. Indeed, this is the intuition behind the famous hand-over-hand locking (lock-coupling) approach [28, 70].

Our framework yields a methodology for verifiable reads-write concurrency. In essence, it suffices for programmers to identify base conditions for their sequential data structure's read-only operations. Then, they can transform their sequential code using means such as readers-writer locks or RCU, to ensure that read-only operations have base points when run concurrently with updates.

It is worth noting that there is a degree of freedom in defining base conditions. If coarsely defined, they can capture the validity of the entire read-set, yielding coarse-grained synchronization as in snapshot isolation and transactional memories. Yet using more precise observations based on the data structure's inner workings can lead to fine-grained base conditions and to better concurrency. Our formalism thus applies to solutions ranging from validation of the entire read-set [40], through multi-versioned concurrency control [30], which has read-only operations read a consistent snapshot of their entire read-set, to fine-grained solutions that hold a small number of locks, like hand-over-hand locking.

Overview of Contributions This paper makes several contributions that arise from our observation regarding the key role of base conditions. We observe that obtaining base points of base conditions guarantees a property we call *validity*, which specifies that a concurrent execution does not reach local states that are not reachable in sequential ones. Intuitively, this property is needed in order to avoid situations like division by zero during the execution of the operation. To avoid returning old values, we restrict the locations of the base points that can potentially have effect on the return value of a read-only operation *ro* to coincide with the return event of an update operation which either immediately precedes, or is executed concurrently with *ro*. Somewhat surprisingly, this does not suffice for the commonly-used correctness criterion of *linearizability (atomicity)* [49] or even *sequential consistency* [53]. Rather, it guarantees a correctness notion weaker than linearizability, similar to Lamport's *regularity* semantics for registers, which we extend here for general objects for the first time.

In Section 2.2, we present a formal model for shared memory data structure implementations and executions, and define correctness criteria. Section 2.3 presents our methodology for achieving regularity and validity: We formally define the notion of a base condition, as well as base points, which link the sequentially-defined base conditions to concurrent execu-

tions. We assert that *base point consistency* implies validity, and that the more restricted base point condition, which we call *regularity base point consistency*, implies regularity. We proceed to exemplify our methodology for a standard linked list implementation, in Section 2.4. In Section 2.5 we turn to extend the result for linearizability. We introduce a criterion called *linearizability base point consistency* - a direct generalisation of regularity base point consistency that restricts the base points of non-overlapping read-only operation to respect their real-time order.

Comparison with Other Approaches The regularity correctness condition was introduced by Lamport [54] for registers. To the best of our knowledge, the regularity of a data structure as we present in this paper is a new extension of the definition.

Using our methodology, proving correctness relies on defining a base condition for every state in a given sequential implementation. One easy way to do so is to define base conditions that capture the entire read-set, i.e., specify that there is a point in the execution where all shared variables the operation has read hold the values that were first read from them. But often, such a definition of base conditions is too strict, and spuriously excludes correct concurrent executions. Our definition generalizes it and thus allows for more parallelism in implementations.

Opacity [43] defines a sufficient condition for validity and linearizability, but not a necessary one. It requires that every transaction see a consistent snapshot of all values it reads, i.e., that all these values belong to the same sequentially reachable state. We relax the restriction on shared states busing base conditions.

Snapshot isolation [29] guarantees that no operation ever sees updates of concurrent operations. This restriction is a special case of the possible base points that our base point consistency criterion defines, and thus also implies our condition for the entire read-set.

In this paper we focus on correctness conditions that can be used for deriving a correct data structure that allows reads-write concurrency from a sequential implementation. The implementation itself may rely on known techniques such as locking, RCU [64], pessimistic lock-elision [23], or any combinations of those, such as RCU combined with fine-grained locking [24]. There are several techniques, such as flat-combining [47] and read-write locking [38], that can naturally expand such an implementation to support also write-write concurrency by adding synchronization among update operations.

Algorithm designers usually prove linearizability of by identifying a serialization point for every operation, showing the existence of a specific partial ordering of operations [35], or using rely-guarantee reasoning [79]. Our approach simplifies reasoning – all the designer needs to do now is identify a base condition for every state in the existing sequential implementation, and show that it holds under concurrency. This is often easier than finding

and proving serialization points, as we exemplify. In essence, we break up the task of proving data structure correctness into a generic part, which we prove once and for all, and a shorter, algorithm-specific part. Given our results, one does not need to prove correctness explicitly (e.g., using linearization points or rely-guarantee reasoning, which typically result in complex proofs). Rather, it suffices to prove the much simpler conditions that read-only operations have base points and linearizability follows from our theorems. Another approach that simplifies verifiable parallelization is to re-write the data structure using primitives that guarantee linearizability such as LLX and SCX [33]. Whereas the latter focuses on non-blocking concurrent data structure implementations using their primitive, our work is focused on reads-write concurrency, and does not restrict the implementation; in particular, we target lock-based implementations as well as non-blocking ones.

2.2 Model and Correctness Definitions

We consider a shared memory model where each process performs a sequence of operations on shared data structures. The data structures are implemented using a set $X = \{x_1, x_2, ...\}$ of shared variables. The shared variables support atomic read and write operations (i.e., are atomic registers), and are used to implement more complex data structures. The values in the x_i's are taken from some domain $\mathcal{V}$.

2.2.1 Data Structures and Sequential Executions

A *data structure implementation* (algorithm) is defined as follows:

- A set of states, $\mathcal{S}$, were a *shared state* $s \in \mathcal{S}$ is a mapping $s : X \to \mathcal{V}$, assigning values to all shared variables. A set $\mathcal{S}_0 \subseteq \mathcal{S}$ defines *initial states*.
- A set of operations representing methods and their parameters. For example, $find(7)$ is an operation. Each *operation op* is a state machine defined by:
  - A set of local states $\mathcal{L}_{op}$, which are usually given as a set of mappings l of values to local variables. For example, for a local state l, $l(y)$ refers to the value of the local variable y in l. $\mathcal{L}_{op}$ contains a special initial local state $\perp \in \mathcal{L}_{op}$.
  - A deterministic transition function $\tau_{op}(\mathcal{L}_{op} \times \mathcal{S}) \to Steps \times \mathcal{L}_{op} \times \mathcal{S}$ where $step \in Steps$ is an atomic transition label, which can be *invoke*, $a \leftarrow$ *read*(x_i), *write*(x_i,v), or *return*(v):
    - An *invoke* changes the initial local state $\perp$ into another local state, and does not change the shared state.
    - A $write(x_i, v)$ changes the local state and changes the value of shared variable $x_i \in X$ to v.

* A $a \leftarrow read(x_i)$ reads the value of one variable $x_i \in X$ from the shared state and changes the local state accordingly (i.e., stores the value of x_i in a local variable a).
* A $return(v)$ ends the operation by changing the local state to $\perp$ and returning v to the calling process. It does not change the shared state.

Note that there are no atomic read-modify-write steps. Invoke and return steps interact with the application while read and write steps interact with the shared memory.

We assume that every operation has an isolated state machine, which begins executing from local state $\perp$.

For a transition $\tau(l, s) = \langle step, l', s' \rangle$, l determines the step. If *step* is an invoke, return, or write step, then l' is uniquely defined by l. If *step* is a read step, then l' is defined by l and s, specifically, $read(x_i)$ is determined by $s(x_i)$. Since only write steps can change the content of shared variables, $s = s'$ for invoke, return, and read steps.

For the purpose of our discussion, we assume the entire shared memory is statically allocated. This means that every read step is defined for every shared state in $\mathcal{S}$. One can simulate dynamic allocation in this model by writing to new variables that were not previously used. Memory can be freed by writing a special value, e.g., "invalid", to it.

A state consists of a local state l and a shared state s. By a slight abuse of terminology, in the following, we will often omit either shared or local component of the state if its content is immaterial to the discussion.

A *sequential execution of an operation* from a shared state $s_i \in \mathcal{S}$ is a sequence of transitions of the form:

$$\overset{\perp}{s_i},\ invoke,\ \overset{l_1}{s_i},\ step_1,\ \overset{l_2}{s_{i+1}},\ step_2,\ \dots,\ \overset{l_k}{s_j},\ return_k,\ \overset{\perp}{s_j},$$

where $\tau(l_m,\ s_n) = \langle step_m,\ l_{m+1},\ s_{n+1} \rangle$. The first step is invoke, ensuing steps are read or write steps, and the last step is a return step.

A *sequential execution of a data structure* is a (finite or infinite) sequence μ:

$$\mu = \overset{\perp}{s_1},\ O_1,\ \overset{\perp}{s_2},\ O_2,\ \dots,$$

where $s_1 \in \mathcal{S}_0$ and every $\overset{\perp}{s_j},\ O_j,\ \overset{\perp}{s_{j+1}}$ in μ is a sequential execution of some operation. If μ is finite, it can end after an operation or during an operation. In the latter case, we say that the last operation is *pending* in μ. Note that in a sequential execution there can be at most one pending operation.

A *read-only operation* is an operation that does not perform write steps in any execution. All other operations are *update operations*.

A state is *sequentially reachable* if it is reachable in some sequential execution of a data structure. By definition, every initial state is sequentially reachable. The *post-state* of an invocation of operation o in execution μ is the shared state of the data structure after o's return step in μ; the *pre-state* is the shared state before o's invoke step. Recall that read-only operations do not change the shared state and execution of update operations is serial. Therefore, every pre-state and post-state of an update operation in μ is sequentially reachable. A state st' is sequentially reachable from a state st if there exists a sequential execution fragment that starts at st and ends at st'.

In order to simplify the discussion of initialization, we assume that every execution begins with a dummy (initializing) update operation that does not overlap any other operation.

2.2.2 Correctness Conditions for Concurrent Data Structures

A *concurrent execution fragment of a data structure* is a sequence of interleaved states and steps of different operations, where state consists of a set of local states $\{l_i, ..., l_j\}$ and a shared state s_k, where every l_i is a local state of a pending operation. A *concurrent execution of a data structure* is a concurrent execution fragment of a data structure that starts from an initial shared state. Note that a sequential execution is a special case of concurrent execution.

For example, the following is a concurrent execution fragment that starts from a shared state s_i and invokes two operations: O_A and O_B. The first operation takes a write step, and then O_B takes a read step. We subscript every step and local state with the operation it pertains to.

$$\underset{s_i}{\emptyset},\ invoke_A(),\ \underset{s_i}{\{l_{1,A}\}},\ write_A(x_i, v),\ \underset{s_{i+1}}{\{l_{2,A}\}},\ invoke_B(),\ \underset{s_{i+1}}{\{l_{2,A},\ l_{1,B}\}},\ a \leftarrow read_B(x_i),\ \underset{s_{i+1}}{\{l_{2,A}, l_{2,B}\}}.$$

In the remainder of this paper we assume that for all concurrent executions μ of the date structure, and any two update operations uo_1 and uo_2 invoked in μ, uo_1 and uo_2 are not executed concurrently to each other (i.e., either uo_1 is invoked after uo_2 returns, or vice versa).

For an execution σ of data structure *ds*, the *history* of σ, denoted H_σ, is the subsequence of σ consisting of the invoke and return steps in σ (with their respective return values). For a history H_σ, *complete(H_σ)* is the subsequence obtained by removing pending operations, i.e., operations with no return step, from H_σ. A history is *sequential* if it begins with an invoke step and consists of an alternating sequence of invoke and return steps.

A data structure's correctness in sequential executions is defined using a *sequential specification*, which is a set of its allowed sequential histories.

Given a correct sequential data structure, we need to address two aspects when defining its correctness in concurrent executions. As observed in the definition of opacity [43] for memory transactions, it is not enough to ensure serialization of completed operations, we must also prevent operations from reaching undefined states along the way. The first aspect relates to the data structure's external behavior, as reflected in method invocations and responses (i.e., histories):

Linearizability A history H_σ is *linearizable* [49] if there exists H'_σ that can be created by adding zero or more return steps to H_σ, and there is a sequential permutation π of complete(H'_σ), such that: (1) π belongs to the sequential specification of *ds*; and (2) every pair of operations that are not interleaved in σ, appear in the same order in σ and in π. A data structure *ds* is *linearizable*, also called *atomic*, if for every execution σ of *ds*, H_σ is linearizable.

Regularity We next extend Lamport's regular register definition [54] for data structures (we do not discuss regularity for executions with concurrent update operations, which can be defined similarly to [72]). A data structure *ds* is *regular* if for every execution σ of *ds*, and every read-only operation $ro \in H_\sigma$, if we omit all other read-only operations from H_σ, then the resulting history is linearizable.

Sequential Consistency A history H_σ is *sequentially consistent* [53] if there exists H'_σ that can be created by adding zero or more return steps to H_σ, and there is a sequential permutation π of complete(H'_σ), such that: (1) π belongs to the sequential specification of *ds*; and (2) every pair of operations that belong to the same process, appear in the same order in σ and in π. A data structure *ds* is *sequentially consistent*, if for every execution σ of *ds*, H_σ is sequentially consistent.

Validity The second correctness aspect is ruling out bad cases like division by zero or access to uninitialized data. It is formally captured by the following notion of *validity*: A data structure is *valid* if every local state reached in an execution of one of its operations is sequentially reachable. We note that, like opacity, validity is a conservative criterion, which rules out bad behavior without any specific data structure knowledge. A data structure that does not satisfy our notion of validity may still be correct in a weaker sense, e.g., if allowed to abort an operation, which encountered a sequentially unreachable state. We do not address such an alternative notions of correctness in our discussion.

2.3 Base Conditions, Validity and Regularity

2.3.1 Base Conditions and Base Points

Intuitively, a base condition establishes some link between the local state an operation reaches and the shared variables the operation has read before reaching this state. It is given as a predicate Φ over shared variable assignments. Formally:

Definition 2.3.1 (Base Condition)**.** Let l be a local state of an operation *op*. A predicate Φ over shared variables is a *base condition* for l if every sequential execution of *op* starting from a shared state s such that $\Phi(s) = true$, reaches l.

For completeness, we define a base condition for $step_i$ in an execution μ to be a base condition of the local state that precedes $step_i$ in μ.

Consider a data structure consisting of an array of elements v and a variable $lastPos$, whose last element is read by the function *readLast*. An example of an execution fragment of *readLast* that starts from state s_1 (depicted in Figure 2.1) and the corresponding base conditions appear in Algorithm 1. The *readLast* operation needs the value it reads from $v[tmp]$ to be consistent with the value of $lastPos$ that it reads into tmp because if $lastPos$ is newer than $v[tmp]$, then $v[tmp]$ may contain garbage.

(a) s_1 **(b)** s_2

Figure 2.1: Two shared states satisfying the same base condition $\Phi_3 : lastPos = 1 \wedge v[1] = 7$.

local state	***base condition***	**Function** *readLast*()
$l_1 : \{\}$	$\Phi_1 : true$	$tmp \leftarrow \textbf{\textit{read}}(lastPos)$
$l_2 : \{tmp = 1\}$	$\Phi_2 : lastPos = 1$	$res \leftarrow \textbf{\textit{read}}(v[tmp])$
$l_3 : \{tmp = 1, res = 7\}$	$\Phi_3 : lastPos = 1 \wedge v[1] = 7$	$\textbf{\textit{return}}(res)$

Algorithm 1: The local states and base conditions of readLast when executed from s_1. The shared variable $lastPos$ is the index of the last updated value in array v. See Algorithm 3 for corresponding update operations.

The predicate $\Phi_3 : lastPos = 1 \wedge v[1] = 7$ is a base condition of l_3 because l_3 is reachable from any shared state in which $lastPos = 1$ and $v[1] = 7$ (e.g., s_2 in Figure 2.1), by executing lines 1-2. The base conditions for every possible local state of *readLast* are detailed in Algorithm 2.

We now turn to define base points of base conditions, which link a local state with base condition Φ to a shared state s where $\Phi(s)$ holds.

Definition 2.3.2 (Base Point)**.** Let μ be a concurrent execution, ro be a read-only operation executed in μ, and Φ_t be a base condition of the local state and step at index t in

Shared variables: $lastPos$, $\forall i \in \mathbb{N} : v[i]$

base condition	***step***
$\Phi_1 : true$	$tmp \leftarrow \textbf{\textit{read}}(lastPos)$
$\Phi_2 : lastPos = tmp$	$res \leftarrow \textbf{\textit{read}}(v[tmp])$
$\Phi_3 : lastPos = tmp \wedge v[tmp] = res$	$\textbf{\textit{return}}(res)$

Algorithm 2: ReadLast operation. The shared variable $lastPos$ is the index of the last updated value in array v. See Algorithm 3 for the corresponding update operation.

μ. An execution fragment of ro in μ has a *base point* for point t with Φ_t, if there exists a sequentially reachable post-state s in μ, called a *base point of* t, such that $\Phi_t(s)$ holds.

Note that together with Definition 2.3.1, the existence of a base point s implies that t is reachable from s in all sequential runs starting from s.

We say that a data structure *ds* satisfies *base point consistency* if every point t in every execution of every read-only operation ro of *ds* has a base point with some base condition of t.

The possible base points of read-only operation ro are illustrated in Figure 2.2. To capture real-time order requirements we further restrict base point locations.

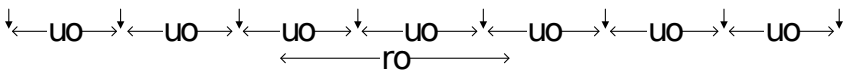


Figure 2.2: Possible locations of ro's base points.

Definition 2.3.3 (Regularity Base Point)**.** A base point s of a point t of ro in a concurrent execution μ is a *regularity base point* if s is the post-state of either an update operation executed concurrently with ro in μ or of the last update operation that ended before ro's invoke step in μ.

The possible regularity base points of a read-only operation are illustrated in Figure 2.3. We say that a data structure *ds* satisfies *regularity base point consistency* if every return step t in every execution of every read-only operation ro of *ds* has a regularity base point with a base condition of t. Note that the base point location is only restricted for the return step, since the return value is determined by its state.

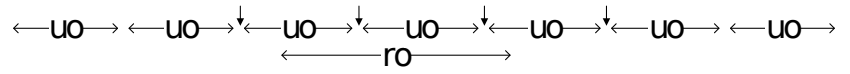


Figure 2.3: Possible locations of ro's regularity base points.

In Algorithm 3 we see two versions of an update operation: *writeSafe* guarantees the existence of a base point for every local state of *readLast* (Algorithm 1), and *writeUnsafe* does not. As shown in Section 2.3.2, *writeUnsafe* can cause a concurrent *readLast* operation interleaved between its two write steps to see values of $lastPos$ and $v[lastPos]$ that do not satisfy *readLast*'s return step's base condition, and to return an uninitialized value.

Function *writeSafe(val)*	**Function** *writeUnsafe(val)*
$i \leftarrow \boldsymbol{read}(lastPos)$	$i \leftarrow \boldsymbol{read}(lastPos)$
$\boldsymbol{write}(v[i+1], val)$	$\boldsymbol{write}(lastPos, i+1)$
$\boldsymbol{write}(lastPos, i+1)$	$\boldsymbol{write}(v[i+1], val)$

Algorithm 3: Unlike *writeUnsafe*, *writeSafe* ensures a regularity base point for every local state of *readLast*; it guarantees that any concurrent *readLast* operation sees values of *lastPos* and $v[tmp]$ that occur in the same sequentially reachable post-state. It also has a single visible mutation point (as defined in Section 2.5), and hence linearizability is established.

2.3.2 Satisfying the Regularity Base Point Consistency

Let us examine the possible concurrent executions an invocation *ro* of *readLast* (Algorithm 1) and an invocation *uo* of *writeSafe* (Algorithm 3) with parameter 80 starting from s_1 (Figure 2.1). There are four possible interleavings of write steps of *uo* and read steps of *ro* starting from s_1 shown in Algorithm 4. In each of them, *ro* returns 7, and s_1 is the base point of its last local state.

μ_1 :	μ_2 :	μ_3 :	μ_4 :
$\boldsymbol{read_{ro}}(lastPos)$	$\boldsymbol{read_{ro}}(lastPos)$	$read_{uo}(lastPos)$	$read_{uo}(lastPos)$
$read_{uo}(lastPos)$	$read_{uo}(lastPos)$	$write_{uo}(v[2], 80)$	$write_{uo}(v[2], 80)$
$write_{uo}(v[2], 80)$	$write_{uo}(v[2], 80)$	$\boldsymbol{read_{ro}}(lastPos)$	$\boldsymbol{read_{ro}}(lastPos)$
$\boldsymbol{write_{uo}}(lastPos, 2)$	$read_{ro}(v[1])$	$\boldsymbol{write_{uo}}(lastPos, 2)$	$read_{ro}(v[1])$
$read_{ro}(v[1])$	$return_{ro}(7)$	$read_{ro}(v[1])$	$return_{ro}(7)$
$return_{ro}(7)$	$\boldsymbol{write_{uo}}(lastPos, 2)$	$return_{ro}(7)$	$\boldsymbol{write_{uo}}(lastPos, 2)$

Algorithm 4: Four interleaved executions of invocation *ro* of *readLast* and invocation *uo* of *writeSafe* that start from s_1.

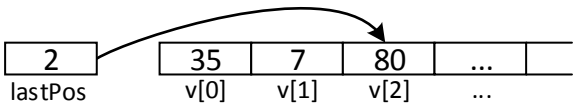


Figure 2.4: The shared state s'_1. It is the post-state after executing *writeSafe* or *writeUnsafe* from s_1 (Figure 2.1) with initial value 80.

Now let us examine a concurrent execution consisting of *readLast* and *writeUnsafe* (Algorithm 3), in which *readLast* reads a value from *lastPos* right after *writeUnsafe* writes to it. In Algorithm 5 we see such an execution that starts from s_1. The last local state of *ro* is $l'_3 = \{tmp = 2, res = 99\}$. Neither s_1 and s'_1 (Figure 2.4) satisfies $\Phi'_3 : lastPos = 2 \wedge v[2] = 99$, meaning that l'_3 does not have a base point with Φ'_3.

Below we show that this is not an artifact of our choice of a base condition – we prove that for every base condition Φ'_3 of l'_3, both $\Phi'_3(s_1)$ and $\Phi'_3(s'_1)$ are false.

Lemma 2.3.4. *A data structure that has both writeUnsafe and readLast is not regularity base point consistent.*

$read_{seq}(lastPos)$
$\boldsymbol{write_{seq}}(lastPos, 2)$
$\boldsymbol{read_{ro}}(lastPos)$
$read_{ro}(v[2])$
$return_{ro}(99)$
$write_{seq}(v[2], 80)$

Algorithm 5: A possible concurrent execution consisting of *readLast* and *writeUnsafe*, starting from s_1.

Proof. Given the execution of Algorithm 5 that starts from the shared state s_1 and ends in shared state s_1', we assume by contradiction that there is such a base condition of $l_3' = \{tmp = 2, res = 99\}$ that is satisfied by s_1 or s_1'. By the definition of base condition, if we execute *readLast* sequentially from a shared state that satisfies l_3''s base condition, we reach l_3'. But if we execute *readLast* from s_1 we reach $l_3 : \{tmp = 1, res = 7\}$ and if we execute from s_1' we reach $l_3'' : \{tmp = 2, res = 80\}$. A contradiction. ■

2.3.3 Deriving Correctness from Base Points

We start by proving that the base point consistency implies validity.

Theorem 2.1 (Validity)**.** *If a data structure* ds *satisfies base point consistency, then* ds *is valid.*

Proof. In order to prove that *ds* is valid, we need to prove that for every execution μ of *ds*, for any operation $op \in \mu$ of *ds*, every local state is sequentially reachable. If μ is a sequential execution then the claim holds. If *op* is an update operation, since every update operation is executed sequentially starting from a sequentially reachable post-state, then every local state of *op* is sequentially reachable. Now we prove for *op* that is a read-only operation in concurrent execution μ. Given that the data structure satisfies the base point consistency, every local state l of every read-only operation in μ has a base point s_{base}. In order to show that l is sequentially reachable, we build a sequential execution μ' that starts from the same initial state as μ and consists of the same update operations that appear in μ until s_{base}. Then we add a sequential execution of *op*. Since s_{base} is a base point of l, l is reached in μ' and therefore is sequentially reachable. ■

We now prove that the regularity base point consistency implies regularity.

Lemma 2.3.5. *Let μ be a concurrent execution of a data structure* ds*. Let ro be a read-only operation of* ds *executed in μ, which returns v. If* ds *satisfies regularity base point consistency then there exists a sequentially reachable shared state s in μ such that: (1) s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked; and (2) when executing ro from s, its return value is equal to v.*

Proof. Let l be the local state that precedes ro's return step. Since τ is deterministic, its return value v is fully determined by l, and every execution of ro that reaches l returns v. Given that *ds* satisfies regularity base point consistency, l, which is the last local state of ro, has a regularity base point for some base condition Φ of l. Let s denote a base point of l for Φ in μ. By the definition of a regularity base point, the shared state s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked, and $\Phi(s)$ is true. By the definition of base condition Φ, we get that l is reached in ro's sequential execution from s, that is, when ro is sequentially executed from s, its return value is v. ■

Theorem 2.2 (Regularity)**.** *If a data structure* ds *satisfies regularity base point consistency, then* ds *is regular.*

Proof. In order to prove that *ds* is regular, we need to show that for every concurrent execution μ of *ds* with history H_μ, for any read-only operation $ro \in H_\mu$, if we omit all other read-only operations from H_μ, the resulting history H_μ^{ro} is linearizable. Recall that update operations are executed sequentially.

If μ includes only update operations then μ vacuously satisfies the condition. Otherwise, let ro be a read-only operation in μ. If ro is pending in μ, we build a sequential history by removing ro's invocation from H_μ^{ro}, which is allowed by the definition of linearizability.

Consider now a read-only operation ro that returns in μ. Since every return step of ro has a regularity base point in μ, by Lemma 2.3.5, we get that there is a shared state s in μ from which ro's sequential execution returns the same value as in μ, and s is the post-state of some update operation that is either concurrent with ro or is the last before ro is invoked. We build a sequential execution μ_{seq}^{ro} from the sequence of update operations in μ with ro added at point s. Then μ_{seq}^{ro} is a sequential execution of *ds*, which belongs to the sequential specification. Every pair of operations that are not interleaved in μ appear in the same order in μ_{seq}^{ro}. Therefore, H_μ^{ro} is linearizable. ■

2.4 Using Our Methodology

We now demonstrate the simplicity of using our methodology. Based on Theorems 2.1 and 2.2 above, the proof for correctness of a data structure (such as a linked list) becomes almost trivial. We look at three linked list implementations – one assuming managed memory, (i.e., automatic garbage collection, Algorithm 6), one using read-copy-update methodology (Algorithm 7), and one using hand-over-hand locking (Algorithm 8).

For Algorithm 6, we first prove that the listed predicates are indeed base conditions, and next we prove that it satisfies the base point consistency and the regularity base point

consistency. By doing so, and based on Theorems 2.1 and 2.2, we get that the algorithm satisfy both validity and regularity.

Consider a linked list node stored in local variable n (we assume the entire node is stored in n, including the value and *next* pointer). Here, $head \stackrel{*}{\Rightarrow} n$ denotes that there is a set of shared variables $\{head, n_1, ..., n_k\}$ such that $head.next = n_1 \wedge n_1.next = n_2 \wedge ... \wedge n_k = n$, i.e., that there exists some path from the shared variable *head* to n. Note that n is the only element in this predicate that is associated with a specific read value. We next prove that this defines base conditions for Algorithm 6.

Lemma 2.4.1. *In Algorithm 6, Φ_i defined therein is a base condition of the i^{th} step of* readLast.

Proof. For Φ_1 the claim is vacuously true. For Φ_2, let l be a local state where *readLast* is about to perform the second read step in *readLast*'s code, meaning that $l(next) \neq \perp$. Note that in this local state both local variables n and *next* hold the same value. Let s be a shared state in which $head \stackrel{*}{\Rightarrow} l(n)$. Every sequential execution from s iterates over the list until it reaches $l(n)$, hence the same local state where $n = l(n)$ and $next = l(n)$ is reached.

For Φ_3, Let l be a local state where *readLast* has exited the while loop, hence $l(n).next = \perp$. Let s be a shared state such that $head \stackrel{*}{\Rightarrow} l(n)$. Since $l(n)$ is reachable from *head* and $l(n).next = \perp$, every sequential execution starting from s exits the while loop and reaches a local state where $n = l(n)$ and $next = \perp$. ■

Lemma 2.4.2. *In Algorithm 6, if a node n is read during concurrent execution μ of* readLast*, then there is a shared state s in μ such that n is reachable from* head *in s and readLast is pending.*

Proof. If n is read in operation *readLast* from a shared state s, then s exists concurrently with *readLast*. The operation *readLast* starts by reading *head*, and it reaches n.

Thus, n must be linked to some node n' at some point during *readLast*. If n was connected (or added) to the list while n' was still reachable from the head, then there exists a state where n is reachable from the head and we are done. Otherwise, assume n is added as the next node of n' at some point after n' is already detached from the list. Nodes are only added via *insertLast*, which is not executed concurrently with any *remove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. ■

The following lemma, combined with Theorem 2.2 above, guarantees that Algorithm 6 satisfies regularity.

Lemma 2.4.3. *Every local state of* readLast *in Algorithm 6 has a regularity base point.*

Proof. We show regularity base points for predicates Φ_i, proven to be base points in Lemma 2.4.1.

```
Function remove(n)
    p ← ⊥
    next ← read(head.next)
    while next ≠ n
        p ← next
        next ← read(p.next)
    write(p.next, n.next)

Function insertLast(n)
    last ← readLast()
    write(last.next, n)
```

Base conditions:

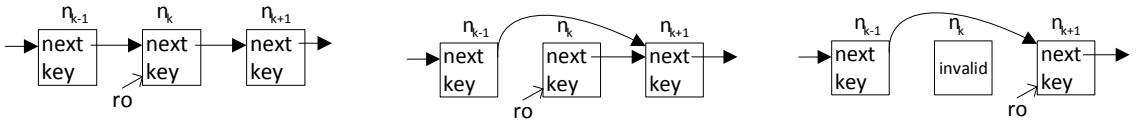
Φ_1 : *true*

Φ_2 : $head \overset{*}{\Rightarrow} n$

Φ_3 : $head \overset{*}{\Rightarrow} n$

```
Function readLast()
    n ← ⊥
    next ← read(head.next)
    while next ≠⊥
        n ← next
        next ← read(n.next)
    return(n)
```

Algorithm 6: A linked list implementation in a memory-managed environment. For simplicity, we do not deal with boundary cases: we assume that a node can be found in the list prior to its deletion, and that there is a dummy head node.



(a) *rcuRemove(n_k)*'s pre-state. **(b)** The shared state during n_k's removal. **(c)** *rcuRemove(n_k)*'s post-state.

Figure 2.5: Shared states in a concurrent execution consisting of *rcuRemove(n_k)* and *rcuReadLast* (*ro*).

The claim is vacuously true for Φ_1. We now prove for Φ_2 and Φ_3 : $head \overset{*}{\Rightarrow} n$. By Lemma 2.4.2 we get that there is a shared state s where $head \overset{*}{\Rightarrow} n$ and *readLast* is pending. Note that n's next field is included in s as part of n's value. Since both update operations - *remove* and *insertLast* - have a single write step, every shared state is a post-state of an update operation. Specifically this means that s is a sequentially reachable post-state, and because *readLast* is pending, s is one of the possible regularity base points of *readLast*. ■

RCU Read-copy-update [64] is a synchronization strategy that aims to reduce read operations' synchronization overhead as much as possible, while risking a high synchronization overhead for update operations. The idea is that only update operations require locks, and the writes mutate the data structure in a way that ensures that concurrent readers always see a consistent view. Additionally, writers do not free data while it is used by readers. Note that RCU does not allow write-write concurrency.

RCU is commonly used via primitives that resemble readers-writer locks [25]: *rcuReadLock* and *rcuReadUnLock*. There are other primitives that encapsulate list traversal, but we do not use them in our example since we wish to illustrate the general approach. Instead, we use primitives that are commonly used for creating RCU-protected non-list data structures (such as arrays and trees): *rcuWrite(p, v)* (originally called *rcuAssignPointer*), and *rcuRead(p)* (originally called *rcuDereference*) [63].

In Algorithm 7, *rcuWrite* is a write step that changes the next pointer of n's predecessor,

```
Function rcuRemove(n)
    p ← ⊥
    next ← read(head.next)
    while next ≠ n
        p ← next
        next ← read(p.next)
    rcuWrite(p.next, n.next)
    rcuWaitForReaders()
    write(n, invalid)

Function insertLast(n)
    last ← readLast()
    write(last.next, n)
```

```
Base conditions:  Function rcuReadLast()
                      rcuReadLock()
                      n ←⊥
        Φ1 : true     next ← rcuRead(head.next)
                      while next ≠⊥
                          n ← next
    Φ2 : head ⇒* n        next ← rcuRead(n.next)
                      rcuReadUnlock()
    Φ3 : head ⇒* n    return(n)
```

Algorithm 7: An RCU linked list implementation. For simplicity, we do not deal with boundary cases: we assume that a node can be found in the list prior to its deletion, and that there is a dummy head node.

and it occurs between the shared states (a) and (b) in Figure 2.5. The invalidation of n takes place once all read-only operations that use n no longer hold a reference to it, as guaranteed by $rcuWaitForReaders()$. The latter happens between the shared states of (b) and (c). The *rcuReadLast* operation holds at most a single reference to list node at a given time, and our base condition links *head* to it. We see in Figure 2.5 that invalid nodes are unreachable from *head* in sequentially reachable post-states. Thus, the base condition $head \overset{*}{\Rightarrow} n$ implies that ro never holds a pointer to an invalid node.

The correctness of the base conditions annotated in Algorithm 7 follows the same reasoning as Lemma 2.4.1, and hence we omit it here. We now prove that Algorithm 7 satisfies regularity base point consistency, and therefore by Theorems 2.1 and 2.2, Algorithm 7 satisfies validity and regularity.

Lemma 2.4.4. *In Algorithm 7, if a node n is read during concurrent execution μ of* rcuReadLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

Proof. If n is read in operation *rcuReadLast* from a shared state s, then s exists concurrently with *rcuReadLast*. The operation *rcuReadLast* starts by reading *head*, and it reaches n.

Thus, n must be linked to some node n' at some point during *rcuReadLast*. If n was connected (or added) to the list while n' was still reachable from the head, then there exists a state where n is reachable from the head and we are done. Otherwise, assume n is added as the next node of n' at some point after n' is already detached from the list. Nodes are only added via *insertLast*, which is not executed concurrently with any *rcuRemove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. ■

Lemma 2.4.5. *Every local state of* rcuReadLast *in Algorithm 7 has a regularity base point.*

Proof. We show regularity base points for predicates Φ_i, proven to be base points in Lemma 2.4.1. The claim is vacuously true for Φ_1.

We now prove for Φ_2 and Φ_3 : $head \stackrel{*}{\Rightarrow} n$. Every read step is encapsulated by *rcuRead*, and is surrounded by *rcuReadLock* and *rcuReadUnlock*. These calls guarantee that as long as the reader holds a reference to the value it read using *rcuRead*, the value cannot be changed by the write step of *rcuRemove* that removes a node from the list. In addition, *rcuRemove* waits for all readers to forget a node before invalidating it, and invalidates it only after the node is not reachable. Therefore, it is guaranteed that every node that is read is valid. In addition, Lemma 2.4.4 guarantees that there is a shared state s where $head \stackrel{*}{\Rightarrow} n$ and *rcuReadLast* is pending. Note that n's next field is included in s as part of n's value. Since the invalidation is not visible to the readers, the post-state of *rcuRemove* and the shared state after *rcuWaitForReaders()* are indistinguishable to the readers. The operation *insertLast* has one write step have a single write step and therefore it is always found between two sequentially reachable shared states.

In conclusion, every shared state is a post-state of an update operation from every reader perspective. Specifically this means that s is a sequentially reachable post-state, and because *rcuReadLast* is pending, s is one of the possible regularity base points of *rcuReadLast*. ■

hand-over-hand locking In *hand-over-hand locking*, a data structure is traversed by holding a lock to the next node in the traversal before unlocking the previous one.

In Algorithm 8 we give a linked list implementation using hand-over-hand locking. The locks used therein are readers-writer locks [62], where write locks are exclusive and multiple threads can obtain read locks concurrently. We define a lock for every shared variable $x_i \in X$, and extend the model with $lock(x_i)$ and $unlock(\{x_{i_1}, x_{i_2}, ...\})$ steps. The correctness of the base conditions annotated in Algorithm 8 follows the same reasoning as Lemma 2.4.1, and hence we omit it here. The reachable post-states in Figure 2.5 are (a) and (c). State (b) does not occur in this implementation since ro cannot access n concurrently with an update operation that holds n's lock. In the following lemma we prove that Algorithm 8 satisfies regularity base point consistency.

Lemma 2.4.6. *In Algorithm 8, if a node n is read during concurrent execution μ of* hohReadLast*, then there is a state where the shared state is s in μ such that n is reachable from* head *in s and ro is pending.*

Proof. If n is read in operation *hohReadLast* from a shared state s, then s exists concurrently with *hohReadLast*. The operation *hohReadLast* starts by reading *head*, and it reaches n.

Thus, n must be linked to some node n' at some point during *hohReadLast*. If n was connected (or added) to the list while n' was still reachable from the head, then there exists

```
Function hohRemove(n)
    p ← ⊥
    lock(head.next)
    next ← read(head.next)
    while next ≠ n
        p ← next
        lock(p.next)
        unlock(p)
        next ← read(p.next)
    write(p.next, n.next)
    lock(n)
    invalidate(n)
    unlock(n, p)

Function insertLast(n)
    last ← readLast()
    write(last.next, n)
```

Base conditions:

Φ_1 : *true*

Φ_2 : $head \overset{*}{\Rightarrow} n$

Φ_3 : $head \overset{*}{\Rightarrow} n$

```
Function hohReadLast()
    n ← ⊥
    lock(head.next)
    next ← read(head.next)
    while next ≠⊥
        n ← next
        lock(n.next)
        next ← read(n.next)
        unlock(n)
    unlock(next)
    return(n)
```

Algorithm 8: A linked list implementation using hand-over-hand locking. For simplicity, we do not deal with boundary cases: we assume that a node can be found in the list prior to its deletion, and that there is a dummy head node.

a state where n is reachable from the head and we are done. Otherwise, assume n is added as the next node of n' at some point after n' is already detached from the list. Nodes are only added via *insertLast*, which is not executed concurrently with any *hohRemove* operation. This means nodes cannot be added to detached elements of the list. A contradiction. ■

Lemma 2.4.7. *Every local state of* hohReadLast *in Algorithm 8 has a regularity base point.*

Proof. We show regularity base points for predicates Φ_i, proven to be base points in Lemma 2.4.1. The claim is vacuously true for Φ_1.

We now prove for Φ_2 and Φ_3 : $head \overset{*}{\Rightarrow} n$. In *hohReadLast*, the reader reads a node only after locking it. Thus, the invalidation of that node is not visible to the readers due to the locking that *hohRemove* performs before any write step, meaning that the post-state of *hohRemove* and the shared state after the first write step of *hohRemove* are indistinguishable to the readers. Therefore, the reader only sees valid nodes. In addition, Lemma 2.4.6 guarantees that there is a shared state s where $head \overset{*}{\Rightarrow} n$ and *hohReadLast* is pending. Note that n's next field is included in s as part of n's value.

The operation *insertLast* has one write step have a single write step and therefore it is always found between two sequentially reachable shared states.

In conclusion, every shared state is a post-state of an update operation from every reader perspective. Specifically this means that s is a sequentially reachable post-state, and because *hohReadLast* is pending, s is one of the possible regularity base points of *hohReadLast*. ■

2.5 Linearizability

We first show that regularity base point consistency is insufficient for linearizability. In Figure 2.6 we show an example of a concurrent execution where two read-only operations ro_1 and ro_2 are executed sequentially, and both have regularity base points. The first operation, ro_1, reads the shared variable *first name* and returns Joe, and ro_2 reads the shared variable *surname* and returns Doe. An update operation uo updates the data structure concurrently, using two write steps. The return step of ro_1 is based on the post-state of uo, whereas ro_2's return step is based on the pre-state of uo. There is no sequential execution of the operations where ro_1 returns Joe and ro_2 returns Doe.

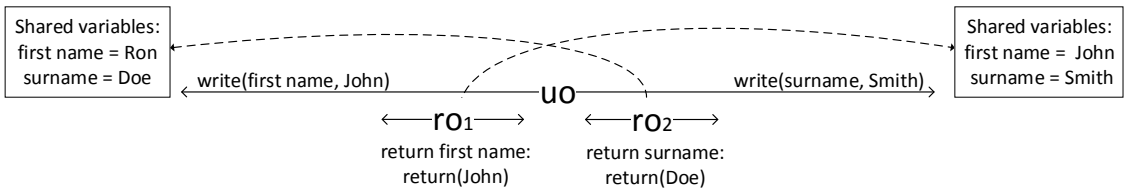


Figure 2.6: Every local state of ro_1 and ro_2 has a regularity base point, and still the execution is not linearizable. If ro_1 and ro_2 belong to the same process, then the execution is not even sequentially consistent.

Thus, an additional condition is required for linearizability. We suggest *linearizability base point consistency* - a condition that adds a restriction to the possible locations of the regularity base points, and is suffice for linearizability.

2.5.1 Linearizability Base Point Consistency

Recall that in order to satisfy linearizability, a data structure needs to guarantee that for every concurrent execution μ there is an equivalent sequential execution μ_{seq} such that the order between non-interleaved operations in μ is preserved in μ_{seq}. One way to ensure this is to determine that the order between the base points has to follow the order of non-interleaved read-only operations.

Definition 2.5.1 (Linearizability Base Point)**.** A base point s of a point t of ro in a concurrent execution μ is a *linearizability base point* if s is the post-state of either an update operation executed concurrently with ro in μ or of the last update operation that ended before ro's invoke step in μ.

The possible linearizability base points of a read-only operation are illustrated in Figure 2.7. We say that a data structure ds satisfies *linearizability base point consistency* if every return step t in every execution of every read-only operation ro of ds has a linearizability base point with a base condition of t.

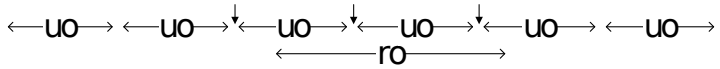


Figure 2.7: Possible locations of ro's linearizability base points.

Notice that base point consistency, regularity base point consistency and linearizability base point consistency, are a sequence in which each condition is a subset of the previous one in terms of possible base point locations. This construction of criteria for data structures correctness follow the construction of Lamport for safe, regular and atomic registers [54]. The connection between regular and linearizable data structures, (as defined by regularity and linearizability base point consistency), reflects the one between regular and atomic registers. Notice that safe data structure can be defined in the same sense.

Theorem 2.3 (Linearizability)**.** *If a data structure* ds *satisfies linearizability base point consistency, then* ds *is linearizable.*

Proof. Given a concurrent execution μ of ds, we create a total ordering σ of operations in μ as follows:

- The order of the update operations in σ is the same as their order in μ.
- Let ro be a read-only operation in μ which returns v. Since ds satisfies linearizability base point consistency, by Lemma 2.3.5, there exists a sequentially reachable state s, which is a post-state of an update operation uo such that uo is either concurrent with ro, or returns before ro is invoked, and the return value of ro is v if it is executed sequentially from s. We therefore, insert immediately the return step of uo, and (2) completes before any uo' which follows uo in σ is invoked. If two read-only operations ro_1 and ro_2 share the same post-state s, and ro_1 returns before ro_2 is invoked in μ, then ro_1 is placed before ro_2 in σ. Otherwise, the order between them is arbitrary provided they are inserted sequentially one after the other, and none of them is inserted after uo' is invoked.

Since by Lemma 2.3.5, the return value of each ro is the same as the one that will be returned in a sequential execution of ro if it is invoked after the write operation immediately preceding ro in σ, σ is a valid sequential execution of ds.

It remains to show that σ preserves the relative order of each pair of non-overlapping operations in μ. First, it is easy to see that the order of each pair of operations op_1 and op_2 such that either both op_1 and op_2 are updates, or exactly one of them is an update, and the the other one is read-only is the same in both μ and σ.

Let ro_1 and ro_2 be two complete read-only operations in μ such that ro_1 returns before ro_2 is invoked in μ; and ro_1's base point s_1 is distinct from ro_2's base point s_2 such that s_1 and s_2 are post-states of update operations uo_1 and uo_2 respectively.

Assume by way of contradiction that ro_2 precedes ro_1 in σ. By construction of σ, uo_2, (resp., uo_1), is the last update operation preceding ro_2 (resp., ro_1). Also, by construction, uo_2 must precede uo_1 in μ, and their respective post-states s_2 and s_1 are base points of ro_2 and ro_1 respectively. However, since s_2 is reached earlier than s_1, by linearizability base point consistency, ro_2 must precede ro_1 in μ, which is a contradiction.

We conclude that σ is a valid sequential execution of *ds*, which preserves the order of all non-overlapping operations in μ. Therefore, the history H of σ belongs to the sequential specification of *ds*, and preserves the order of all non-overlapping operations in μ. Hence, H is a linearization of μ. ■

2.6 Sequential Consistency

Some systems use the correctness criterion of sequential consistency [53], which relaxes linearizability by not requiring *real time order* (RTO) between operations of different processes.

Note that sequential consistency and regularity are incomparable: Regularity does not impose RTO on read-only operations even if they belong to the same process, while in sequential consistency, the RTO of read-only operations of the same process is preserved. On the other hand, regularity enforces the RTO between an update operation and every other operation, while sequential consistency allows re-ordering of operations executed by different processes.

We say that a data structure *ds* satisfies *sequentially base point consistency* if it satisfies the base point consistency, and for every concurrent execution μ in which a read-only operation ro_1 of *ds* precedes a read-only operation ro_2 of *ds* and both belong to the same process, the return step of ro_1 has a base point in μ that precedes or equals to ro_2's return step's base point in μ.

We now prove that the sequentially base point consistency condition ensures sequential consistency.

Lemma 2.6.1. *Let μ be a concurrent execution such that: (1) μ starts from a sequentially reachable post-state s; and (2) every return step of every read-only operation in μ has a base point ; and (3) for every read-only operation ro_1 that precedes a read-only operation ro_2 of the same process, the return step of ro_1 has a base point in μ that precedes or equals to ro_2's return step's base point in μ.*

Then there is a sequential execution μ_{seq} such that: (1) μ_{seq} and μ contain the same operations; and (2) for every process, all its operations appear in the same order in μ_{seq} and in μ.

Proof. We build a sequential execution μ_{seq} in the following way: (1) μ_{seq} starts from the same shared state s as μ. I t is given that s is sequentially reachable. (2) All update operations in μ appear in the same order in μ_{seq}. (3) Every read-only operation ro in μ is executed in μ_{seq} from a post-state that is a base point of the return step of ro. It is given that for operations of the same process, different base points appear in the execution in the same order as the operations do. Therefore if there are multiple possibilities for a base point, the operation is executed from the base point according to the that order. Read-only operations of the same process that have the same base point are executed from it at the same order in μ_{seq} as in μ. (4) The order of read-only operations that do not belong to the same process and are executed from the same base point is arbitrary.

Since only update operations can change the shared state and their sequential order is the same in both executions, every update operation is executed in μ_{seq} from the same shared state as in μ. By the definitions of base point and base condition we get that every read-only operation in μ_{seq} returns the same value in μ_{seq} as in $\mu - ro$ is executed from a shared state that is a base point of its return step, and the last local state determines ro's return value. ■

Theorem 2.4 (Sequential consistency)**.** *If a data structure* ds *satisfies sequentially base point consistency, then* ds *is sequentially consistent.*

Proof. Let μ be a concurrent execution of *ds*. By Lemma 2.6.1 we get that there is a sequential execution μ_{seq}, such that $H_{\mu_{seq}}$ is a permutation of complete(H_μ) that belongs to the sequential specification of *ds* and keeps the RTO of operations that belong to the same process in μ. Thus *ds* is sequentially consistent. ■

Acknowledgements

We thank Naama Kraus, Dahlia Malkhi, Yoram Moses, Dani Shaket, Noam Shalev, and Sasha Spiegelman for helpful comments and suggestions.

Chapter 3

Paper: A Constructive Approach for Proving Data Structures' Linearizability

Kfir Lev-Ari, Gregory V. Chockler, Idit Keidar: "A Constructive Approach for Proving Data Structures' Linearizability". Distributed Computing 29th International Symposium, DISC 2015 Tokyo, Japan, October 7–9, 2015 Proceedings. ed. / Yoram Moses. Vol. 9363 Springer-Verlag, 2015. p. 356–370.

In this paper we generalize base-point analysis to any type of data structure and provide a constructive road-map for proving correctness of data structures (exemplify via Lazy List).

A Constructive Approach for Proving Data Structures' Linearizability *

Kfir Lev-Ari[1], Gregory Chockler[2], and Idit Keidar[1]

[1] *Viterbi Department of Electrical Engineering, Technion, Haifa, Israel*

[2] *CS Department, Royal Holloway University of London, Egham, UK*

Abstract

We present a comprehensive methodology for proving correctness of concurrent data structures. We exemplify our methodology by using it to give a roadmap for proving linearizability of the popular Lazy List implementation of the concurrent set abstraction. Correctness is based on our key theorem, which captures sufficient conditions for linearizability. In contrast to prior work, our conditions are derived directly from the properties of the data structure in sequential runs, without requiring the linearization points to be explicitly identified.

3.1 Introduction

While writing an efficient concurrent data structure is challenging, proving its correctness properties is usually even more challenging. Our goal is to simplify the task of proving correctness. We present a methodology that offers algorithm designers a constructive way to analyze their data structures, using the same principles that were used to design them in the first place. It is a generic appproach for proving handcrafted concurrent data structures' correctness, which can be used for presenting intuitive proofs.

The methodology we present here generalizes our previous work on reads-write concurrency [56], and deals also with concurrency among write operations as well as with any number of update steps per operation (rather than a single update step per operation as in

*This work was partially supported by the Israeli Science Foundation (ISF), the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by a Royal Society International Exchanges Grant IE130802, and by the Randy L. and Melvin R. Berlin Fellowship in the Cyber Security Research Program.

[56]). To do so, we define the new notions of *base point preserving steps*, *commutative steps*, and *critical sequence*. We demonstrate the methodology by proving linearizability of Lazy List [46], as opposed to toy examples in [56].

Our analysis consists of three stages. In the first stage we identify conditions, called *base conditions* [56], which are derived *entirely* by analysis of sequential behavior, i.e., we analyze the algorithm as if it is designed to implement the data structure correctly only in sequential executions. These conditions link states of the data structure with outcomes of operations running on the data structure from these states. More precisely, base conditions tell us what needs to be satisfied by a state of the data structure in order for a sequential execution to reach a specific point in an operation from that state. For example, Lazy List's *contains(31)* operation returns *true* if 31 appears in the list. A possible base condition for returning *true* is "there is an element that is reachable from the head of the list and its value is 31". Every state of Lazy List that satisfies this base condition causes *contains(31)* to return *true*.

In the second stage of our analysis we prove the linearization of update operations, (i.e., operations that might modify shared memory). We state two conditions on update operations that together suffice for linearizability. The first is *commutativity* of steps taken by *concurrent updates*. The idea here is that if two operations' writes to shared memory are interleaved, then these operations must be independent. Such behavior is enforced by standard synchronization approaches, e.g., two-phase locking. The second condition requires that some state reached during the execution of the update operation satisfy base conditions of all the update operation's writes. For example, the update steps of an *add(7)* operation in Lazy List depend on the predecessor and successor of 7 in the list. Indeed, Lazy List's *add(7)* operation writes to shared memory only after locking these nodes, which prevent concurrent operations from changing the two nodes that satisfy the base conditions of *add(7)*'s steps.

In the third stage we consider the relationship between update operations and read-only operations. We first require each update operation to have at most one point in which it changes the state of the data structure in a way that "affects" read-only operations. We capture the meaning of "affecting" read-only operations using base conditions. Intuitively, if an update operation has a point in which it changes something that causes the state to satisfy a base condition of a read-only operation, then we know that this point defines the outcome of the read-only operation. For example, Lazy List's *remove(3)* operation first *marks* the node holding 3, and then detaches it from its predecessor. Since *contains* treats marked nodes as deleted, the second update step does not affect *contains*.

In addition, we require that each read-only operation has a state in the course of its execution that satisfies its base condition. In order to show that such a state exists, we need to examine how the steps that we have identified in the update operations affect the base

conditions of the read-only operations. For example, in Lazy List, *contains(9)* relies on the fact that if a node holding 9 is reachable from the head of the list, then there was some concurrent state in which a node holding 9 was part of the list. We need to make sure that the update operations support this assumption.

The remainder of this paper is organized as follows: Section 3.2 provides formal preliminaries. We formally present and illustrate the analysis approach in Section 3.3. We state and prove our main theorem in Section 3.4. Then, we demonstrate how base point analysis can be used as a roadmap for proving linearizability of Lazy List in Section 3.5.

3.2 Preliminaries

We extend here the model and notions we defined in [56]. Generally speaking, we consider a standard shared memory model [26] with one refinement, which is differentiating between local and shared state, as needed for our discussion.

Each process performs a sequence of operations on shared data structures implemented using a set $X = \{x_1, x_2, ...\}$ of shared variables. The shared variables support atomic operations, such as read, write, CAS, etc. A *data structure implementation* (algorithm) is defined as follows:

- A set $\mathcal{S}$ of *shared states*, some of which are *initial*, where $\underline{s} \in \mathcal{S}$ is a mapping assigning a value to each shared variable.

- A set of operations representing methods and their parameters (e.g., $add(7)$ is an operation). Each *operation op* is a state machine defined by: A set of local states $\mathcal{L}_{op}$, which are given as mappings l of values to local variables; and a deterministic transition function $\tau_{op}(\mathcal{L}_{op} \times \mathcal{S}) \to Steps \times \mathcal{L}_{op} \times \mathcal{S}$ where *Steps* are transition labels, such as *invoke*, *return(v)*, $a \leftarrow$ *read*(x_i), *write*(x_i,*v*), *CAS*(x_i,v_{old},v_{new}), etc.

Invoke and return steps interact with the application, while read and write steps interact with the shared memory and are defined for every shared state. In addition, the implementation may use synchronization primitives (locks, barriers), which constrain the scheduling of ensuing steps, i.e., they restrict the possible *executions*, as we shortly define.

For a transition $\tau(l, \underline{s}) = \langle step, l', \underline{s'} \rangle$, l determines the step. If *step* is an invoke or return, then l' is uniquely defined by l. Otherwise, l' is defined by l and potentially $\underline{s}$. For invoke, return, read and synchronization steps, $\underline{s} = \underline{s'}$. If any of the variables is assigned a different value in $\underline{s}$ than in $\underline{s'}$, then the step is called an *update step*.

A state consists of a local state l and a shared state $\underline{s}$. We omit either the shared or the local component of the state if its content is immaterial to the discussion. A *sequential*

execution of an operation from a shared state $\underline{s_i} \in \mathcal{S}$ is a sequence of transitions of the form:

$$\underset{\underline{s_i}}{\overset{\perp}{}},\ invoke,\ \underset{\underline{s_i}}{\overset{l_1}{}},\ step_1,\ \underset{\underline{s_{i+1}}}{\overset{l_2}{}},\ step_2,\ \dots\ ,\ \underset{\underline{s_j}}{\overset{l_k}{}},\ return_k,\ \underset{\underline{s_j}}{\overset{\perp}{}},$$

where $\perp$ is the operation's initial local sate and $\tau(l_m,\ \underline{s_n}) = \langle step_m,\ l_{m+1},\ \underline{s_{n+1}}\rangle$. The first step is invoke and the last step is a return step.
A *sequential execution of a data structure* is a (finite or infinite) sequence μ:

$$\mu = \underset{\underline{s_1}}{\overset{\perp}{}},\ O_1,\ \underset{\underline{s_2}}{\overset{\perp}{}},\ O_2,\ \dots\ ,$$

where $\underline{s_1} \in \mathcal{S}_0$ and every $\underset{\underline{s_j}}{\overset{\perp}{}},\ O_j,\ \underset{\underline{s_{j+1}}}{\overset{\perp}{}}$ in μ is a sequential execution of some operation. If μ is finite, it can end after an operation or during an operation. In the latter case, we say that the last operation is *pending* in μ. Note that in a sequential execution there can be at most one pending operation.

A *concurrent execution fragment of a data structure* is a sequence of interleaved states and steps of different operations, where each state consists of a set of local states $\{l_i, ..., l_j\}$ and a shared state $\underline{s_k}$, where every l_i is a local state of a *pending* operation, which is an operation that has not returned yet. A *concurrent execution of a data structure* is a concurrent execution fragment that starts from an initial shared state and an empty set of local states. In order to simplify the discussion of initialization, we assume that every execution begins with a dummy (initializing) update operation that does not overlap any other operation. A state $\underline{s'}$ is *reachable from a state* $\underline{s}$ if there exists an execution fragment that starts at $\underline{s}$ and ends at $\underline{s'}$. A state is *reachable* if it is reachable from an initial state.

An operation for which there exists an execution in which it perform update steps is called *update operation*. Otherwise, it is called a *read-only operation*.

A data structure's correctness in sequential executions is defined using a *sequential specification*, which is a set of its allowed sequential executions. A *linearization* of execution μ is a sequential execution μ_l, such that:

- Every operation that is not invoked in μ is not invoked in μ_l.
- Every operation that returns in μ returns also in μ_l and with the same return value.
- μ_l belongs to the data structure's sequential specification.
- The order between non-interleaved operations in μ and μ_l is identical.

A data structure is *linearizable* [49] if each of its executions has a linearization.

3.3 Base Point Analysis

In this section we present key definitions for analyzing and proving correctness using what we call *base point analysis*. We illustrate the notions we define using Lazy List [46], whose pseudo code appears in Algorithm 1.

```
▷ Φ_loc(s, n1, n2, e) : (Head ⇒* n1) ∧ (n1.next = n2) ∧ ¬n1.marked ∧ ¬n2.marked
  ∧ (n1.val < e) ∧ (e ≤ n2.val)

1  Function contains(e)
2      c ← read(Head)
3      while read(c.val) < e
4          c ← read(c.next)
5       ▷ Φ_c : (Head ⇒* c) ∧ (c.val ≥ e)
          ∧ (∄n : (Head ⇒* n) ∧
          (e ≤ n.val < c.val))
6      if read(c.marked)∨read(c.val) ≠ e
7          ▷ Φ_c∧(c.marked ∨c.val ≠ e)
8          return false
9      else
10         ▷ Φ_c ∧ (¬c.marked∧c.val = e)
11         return true

12 Function add(e)
13     ⟨n1, n2⟩ ← locate(e)
14      ▷ Φ_loc(s, n1, n2, e)
15     if read(n2.val) ≠ e
16         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val ≠ e)
17         write(n3, new Node(e, n2))
18         write(n1.next, n3)
19         unlock(n1)
20         unlock(n2)
21         return true
22     else
23         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val = e)
24         unlock(n1)
25         unlock(n2)
26         return false

27 Function locate(e)
28     while true
29         n1 ← read(Head)
30         n2 ← read(n1.next)
31         while read(n2.val) < e
32             n1 ← n2
33             n2 ← read(n2.next)
34         lock(n1)
35         lock(n2)
36         if read(n1.marked) = false ∧
37         read(n2.marked) = false ∧
38         read(n1.next) = n2
39             return ⟨n1, n2⟩
40         else
41             unlock(n1, n2)
42 Function remove(e)
43     ⟨n1, n2⟩ ← locate(e)
44      ▷ Φ_loc(s, n1, n2, e)
45     if read(n2.val) = e
46         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val = e)
47         write(n2.marked, true)
48         write(n1.next, n2.next)
49         unlock(n1)
50         unlock(n2)
51         return true
52     else
53         ▷ Φ_loc(s, n1, n2, e) ∧ (n2.val ≠ e)
54         unlock(n1)
55         unlock(n2)
56         return false
```

Algorithm 1: Lazy List. Base conditions are listed as comments, using Φ_{loc} defined above the functions.

We start by defining *base conditions* [56]. A *base condition* establishes a connection between the local state that an operation reaches and the shared variables the operation has read before reaching this state. It is given as a predicate Φ over shared variable assignments. Formally:

Definition 3.3.1 (Base Condition)**.** Let l be a local state of an operation *op*. A predicate Φ over shared variables is a *base condition* for l if every sequential execution of *op* starting

from a shared state $\underline{s}$ such that $\Phi(\underline{s})$ is true, reaches l.

For completeness, we define a base condition for $step_i$ in an execution μ to be a base condition of the local state that precedes $step_i$ in μ. For example, consider an execution of Lazy List's *contains(31)* operation that returns *true*. A possible base condition for that return step is ϕ : "there is an unmarked node in which $key = 31$, and that node is reachable from the head of the list". Every sequential execution of *contains(31)* from a shared state that satisfied ϕ reaches the same *return true* step. Base conditions for all of Lazy List's update and return steps are annotated in Algorithm 1, and are discussed in detail in Section 3.5.1 below.

For a given base condition, the notion of *base point* [56] links the local state that has base condition Φ to a shared state $\underline{s}$ where $\Phi(\underline{s})$ holds.

Definition 3.3.2 (Base Point)**.** Let *op* be an operation in an execution μ, and let Φ_t be a base condition for the local state at point t in μ. An execution prefix of *op* in μ has a *base point* for point t with Φ_t, if there exists a shared state $\underline{s}$ in μ, called a *base point of* t, such that $\Phi_t(\underline{s})$ holds.

Note that together with Definition 3.3.1, the existence of a base point $\underline{s}$ for point t implies that the step or local state at point t in operation *op* is reachable from $\underline{s}$ in a sequential run of *op* starting from $\underline{s}$. In Figure 3.1 we depict two states of Lazy List: $\underline{s_1}$ is a base point for a *return true* step of *contains(7)*, whereas $\underline{s_2}$ is not.

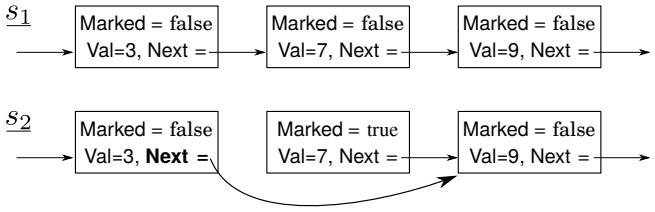


Figure 3.1: Two states of Lazy List (Algorithm 1): $\underline{s_1}$ is a base point for *contains(7)*'s *return true* step, as it satisfies the base condition "there is a node that is reachable from the head of the list, and its value is 7". The shared state $\underline{s_2}$ is not a base point of this step, since there is no sequential execution of *contains(7)* from $\underline{s_2}$ in which this step is reached.

Let $\underline{s_0}$ and $\underline{s_1}$ be two shared states, and let $\underline{s_0}$, st, $\underline{s_1}$ be an execution fragment. We call $\underline{s_0}$ the *pre-state* of step st, and $\underline{s_1}$ the *post-state* of st.

We now define *base point preserving* steps, which are steps under which base conditions are invariant.

Definition 3.3.3 (Base Point Preserving Step)**.** A step st is base point preserving with respect to an operation *op* if for any update or return step b of *op*, for any concurrently reachable pre-state of st, st's pre-state is a base point of b if and only if st's post-state is a base point of b.

An example of a base point preserving step is illustrated in Figure 3.2. In this example, the second write step in Lazy List's *remove* operation is base point preserving for *contains*. Intuitively, since *contains* treats marked nodes as removed, the same return step is reached regardless whether the marked node is detached from the list or reachable from the head of the list.

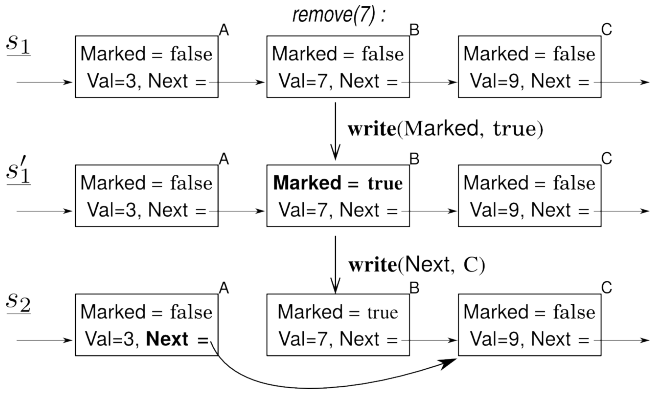


Figure 3.2: Operation *remove(7)* of Lazy List has two write steps. In the first, *marked* is set to *true*. In the second, the *next* field of the node holding 3 is set to point to the node holding 9. If a concurrent *contains(7)* operation sequentially executes from state $\underline{s_1}$, it returns true. If we execute *contains(7)* from $\underline{s_1'}$, i.e., after *remove(7)*'s first write, *contains* sees that 7 is marked, and therefore returns false. If we execute *contains* from state $\underline{s_2}$, after *remove(7)*'s second write, *contains* does not see B because it is no longer reachable from the head of the list, and also returns false. The second write does not affect the return step, since in both cases it returns false.

3.4 Linearizability using Base Point Analysis

We use the notions introduced in Section 3.3 to define sufficient conditions for linearizability. In Section 3.4.1 we define conditions for update operations, and in Section 3.4.2 we define an additional condition on read-only operations, and show that together, our conditions imply linearizability.

3.4.1 Update Operations

We begin by defining the *commutativity* of steps.

Definition 3.4.1 (Commutative Steps)**.** Consider an execution μ of a data structure ds that includes the fragment $a, \underline{s_1}, b, \underline{s_2}$. We say that steps a and b *commute* if $a, \underline{s_1}, b, \underline{s_2}$ in μ can be replaced with $b, \underline{s_1'}, a, \underline{s_2}$, so that the resulting sequence μ' is a valid execution of ds.

We now observe that if two update steps commute, then their resulting shared state is identical for any ordering of these steps along with interleaved read steps.

Observation 3.4.2. Let $\underline{s_0}, a, \underline{s_1}, b, \underline{s_2}$ be an execution fragment of two update steps a and b that commute, then $\underline{s_2}$ is the final shared state in any execution fragment that starts from $\underline{s_0}$ and consists of a, b and any number of read steps (for any possible ordering of steps).

We are not interested in commutativity of all steps, but rather of "critical" steps that modify shared memory or determine return values. This is captured by the following notion:

Definition 3.4.3 (Critical Sequence)**.** The *critical sequence* of an update operation *op* in execution μ is the subsequence of *op*'s steps from its first to its last update step; if *op* takes no update steps in μ, then the critical sequence consists solely of its last read.

For example, if in Lazy List op_1 = *add(2)* and op_2 = *add(47)* concurrently add items in disjoint parts of the list, then all steps in op_1's critical sequence commute with all those in op_2's critical sequence. The same is not true for list traversal steps taken before the critical sequence, since op_2 may or may not traverse a node holding 2, depending on the interleaving of op_1 and op_2's steps. In general, Lazy List uses locks to ensure that the critical steps of two operations overlap only if these operations' respective steps commute. This is our first condition for linearizability of update operations.

Our second requirement from update operations is that each critical sequence begin its execution from a base point of all the operation's update and return steps. Together, we have:

Definition 3.4.4 (Linearizable Update Operations)**.** A data structure *ds* has linearizable update operations if for every execution μ, for every update operation $uo_i \in \mu$:

1. $\forall uo_j \in \mu, i \neq j$, if the critical sequence of uo_j interleaves with the critical sequence of uo_i in μ, then all of uo_i's steps in its critical sequence commute with all of the steps in uo_j's critical sequence, and all the update steps of uo_i and uo_j are base point preserving for uo_j and uo_i respectively.

2. The pre-state of uo_i's critical sequence is a base point for all of uo_i's update and return steps, and moreover, if uo_i is complete in μ, then this state is not a base point for any other possible update step of uo_i.

To satisfy these conditions, before its critical sequence, an update operation takes actions to guarantee that the pre-state of its first update will be a base point for the operation's update and return steps, as depicted in Figure 3.3. For example, any algorithm that follows the two-phase locking protocol [31] satisfies these conditions: operations perform concurrent modifications only if they gain disjoint locks, which means that their steps commute. And in addition, once all locks are obtained by an operation, the shared state is a base point for all of its ensuing steps, i.e., for its critical sequence.

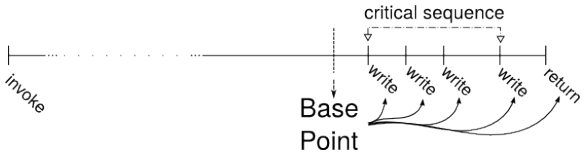


Figure 3.3: The structure of update operations. The steps before the critical sequence ensure that the pre-state of the first update step is a base point for all of the update and return steps.

We now show that every execution that has linearizable update operations and no read-only operations is linearizable.

Lemma 3.4.5. *Let μ be an execution consisting of update operations of some data structure that has linearizable update operations. Let μ' be a sequential execution of all the operations in μ starting from the same initial state as μ such that if some operation op_1's critical sequence ends before the critical sequence of another operation op_2 begins in μ, then op_1 precedes op_2 in μ'. Then μ' is a linearization of μ.*

Proof. By construction, μ' includes only invoke steps from μ, and every two operations that are not interleaved in μ occur in the same order in μ and μ'. It remains to show that every operation has the same return step in μ and μ'.

Denote by μ'_i the prefix of μ' consisting of i operations, and by μ_i the subsequence of μ consisting of the steps of the same i operations. Denote by op_i the i^{th} operation in μ'.

We prove by induction on i that μ'_i is a linearization of μ_i and both executions end in the same final state. As noted above, for linearizability, it suffices to show that all operations that return in both μ'_i and μ_i return the same value.

The first operation in both μ and μ' is a dummy initialization, which returns before all other operations are invoked. Hence, $\mu_1 = \mu'_1$, and their final states are identical.

Assume now that μ'_i is a linearization of μ_i and their final states are the same. The critical sequence of op_{i+1} in μ_{i+1} overlaps the critical sequences of the last zero or more operations in μ_i. We need to show that (1) the execution of op_{i+1} that overlaps these steps in μ_{i+1} yields the same return value and the same final state as a sequential execution of op_{i+1} from the final state of μ_i; and (2) the return values of the operations that op_{i+1} is interleaved with in μ_{i+1} are unaffected by the addition of op_{i+1}'s steps.

(1) By definition 3.4.4, the pre-state $\underline{p}$ of op_{i+1}'s critical sequence in μ_{i+1} is a base point for op_{i+1}'s update and return steps. Note that $\underline{p}$ occurs in μ_{i+1} before any update step of op_{i+1}, and thus it also occurs in μ_i. Thus, the same $\underline{p}$ occurs also in μ_i. All the update steps after $\underline{p}$ in μ_{i+1} belong to operations that have interleaved critical sequences with op_{i+1} in μ_{i+1}, and therefore by definition 3.4.4 their update steps are base point preserving for op_{i+1}. These are the update steps that occur after $\underline{p}$ in μ_i, and so the final state of μ_i is a base point for the update and return steps that op_{i+1} takes in μ_{i+1}.

By the induction hypothesis, the last states of μ_i and μ'_i are identical, and we conclude that op_{i+1} has the same update and return steps in μ_{i+1} and μ'_{i+1}.

In addition, the final states of μ_{i+1} and μ'_{i+1} occur at the end of execution fragments that consist of the same update steps, s.t. if two update steps have different orders in μ_{i+1} and in μ'_{i+1} then they are commute. By Observation 3.4.2 we conclude that the last states of μ_{i+1} and μ'_{i+1} are identical.

(2) If an update step of op_{i+1} occurs in μ_{i+1} before operation op_j's return step, then op_{i+1} has an interleaved critical sequence with op_j. This means that all of op_{i+1}'s update steps are base point preserving for op_j. Thus, the same base points are reached before op_j's critical sequences in μ_i and in μ_{i+1}. By definition 3.4.4, op_j takes the same update and return steps in μ_i and μ_{i+1}. ■

3.4.2 Read-Only Operations

We state two conditions that together ensure linearizability of read-only operations. First, each read-only operation *ro* should have a base point for its return step, which can be either a post-state of some step of operation that is concurrent to *ro*, or the pre-state of *ro*'s invoke step. Second, update operations should have at most one step that is not base point preserving for read-only operations.

In Theorem 3.1 we present a sufficient condition for linearizability. Intuitively, we want the linearizable update operations to satisfy two conditions: (1) the read-only operations should see the update operations as a sequence of single steps that mutate the shared state. To express this relation we use the base point preserving property; and (2) the update operations should guarantee the correctness of the returned values of the read-only operation, as expressed by the return steps' base conditions.

Theorem 3.1. *Let ds be a data structure that has linearizable update operations. If ds satisfies the following conditions, it is linearizable:*

1. *Every update operation of ds has at most one step that is not base point preserving with respect to all read-only operations.*

2. *For every execution μ, for every complete read-only operation $ro \in \mu$, there exists in μ a shared state $\underline{s}$ between the pre-state of ro's invoke step and the pre-state of ro's return step (both inclusive) that is a base-point for ro's return step.*

Proof. For a given execution μ^-, let μ be an execution that is identical to μ^- with the addition that all pending operations in μ^- are allowed to complete. Note that μ also has linearizable update operations. We now show that μ has a linearization, and therefore μ^- has a linearization.

We build a sequential execution μ_{seq} as follows:

1. μ_{seq} starts from the same shared state as μ.

2. We sequentially execute all the update operations that takes steps of their critical sequence in μ in the order of their steps that are not base point preserving for read-only operations, (or the last read step in case all steps are base point preserving). We denote this sequence of steps by $\{ord_i\}$. The update operation that performs ord_i in μ is denoted uo_i.

3. Each read-only operation ro of μ is executed in μ_{seq} after an update operation uo_i such that the post-state of ord_i in μ is a base point for ro, and is either concurrent to ro or the latest step in $\{ord_i\}$ that precedes ro's invoke step. Such a step exists since (1) by our assumption, ro has a base point between its invoke step's pre-state and its return step's pre-state; and (2) every step that is not in $\{ord_i\}$ is base point preserving for ro.

4. The order in μ_{seq} between non-interleaved read-only operations that share the same base point follows their order in μ. The order between interleaved read-only operations that are executed in μ_{seq} from the same base point is arbitrary.

Now, by Lemma 3.4.5, the sequence of update operations in μ_{seq} is a linearization of the sequence of update operations in μ.

Therefore we only need to prove that the order between the read-only operations and other operations that are not interleaved in μ is identical in μ_{seq} and μ, and that each read-only operation has the same return step in both executions.

We observe that:

1. In μ and μ_{seq} the steps of $\{ord_i\}$ appear in the same order, and in both executions each read-only operation is either executed after the same ord_i in both, or is executed concurrently to ord_i in μ and immediately after uo_i in μ_{seq}.

2. Each shared state satisfies the same base conditions since the update steps that appear in a different order in μ and μ_{seq} commute.

Therefore each post-state of ord_i remains a base point in μ_{seq} for the same read-only operations that it was in μ, and thus each read-only operation reaches the same return step as in μ.

Assume towards contradiction that two read-only operations ro_1 and ro_2 have a different order in μ and μ_{seq}, and w.l.o.g. ro_1 precedes ro_2 in μ, and ro_2 precedes ro_1 in μ_{seq}.

Let uo_1 be the update operation that precedes ro_1 in μ_{seq}, and uo_2 be the update operation that precedes ro_2 in μ_{seq}. $uo_2 \neq uo_1$, otherwise ro_1 and ro_2 had the same base

point and their execution order was identical to their order in μ. Since ro_2 precedes ro_1 in μ_{seq}, we conclude that ord_2 occurs before ord_1 in μ. ord_1 takes place in μ as last as one step before uo_1's return step. Therefore ord_2 must appear somewhere before ro_1's return step. But ro_1 precedes ro_2 in μ, meaning that ord_2 is not the latest steps of ord that precedes ro_2's invoke step, in contradiction. ■

3.5 Roadmap for Proving Linearizability

We now prove that Lazy List (Algorithm 1) satisfies the requirements of Theorem 3.1, implying that it is linearizable. We demonstrate the three stages of our roadmap for proving linearizability using base point analysis.

3.5.1 Stage I: Base Conditions

We begin by identifying base conditions for the operations' update and return steps. The base conditions are annotated in comments in Algorithm 1. To do so, we examine the possible sequential executions of each operation.

Add & Remove Let $Head \overset{*}{\Rightarrow} n$ denote that there is a set of shared variables $\{Head, x_1, ..., x_k\}$ such that $Head.next = x_1 \wedge x_1.next = x_2 \wedge ... \wedge x_k = n$, i.e., that there exists some path from the shared variable $Head$ to n. Let $\Phi_{loc}(\underline{s}, n_1, n_2, e)$ be the predicate indicating that in the shared state $\underline{s}$, the place of the key e in the list is immediately after the node n_1, and at or just before the node n_2:

$\Phi_{loc}(\underline{s}, n_1, n_2, e) : Head \overset{*}{\Rightarrow} n_1 \ \wedge n_1.next = n_2 \ \wedge \neg n_1.marked \ \wedge \neg n_2.marked \ \wedge n_1.val < e \wedge \ e \leq n_2.val.$

Observation 3.5.1. $\Phi_{loc}(\underline{s}, n_1, n_2, e)$ is a base condition for the local state of add(e) (remove(e)) after line 14 (resp., 44).

Now, $\Phi_{loc}(\underline{s}, n_1, n_2, e) \wedge n_2.val \neq e$ is a base condition for *add*'s write and *return true* steps and *removes*'s *return false* step. And a base condition for *add*'s *return false* step and *remove*'s write and *return true* steps is $\Phi_{loc}(\underline{s}, n_1, n_2, e) \wedge n_2.val = e$.

Contains First, we define the following predicate:

$$\Phi_c : \ Head \overset{*}{\Rightarrow} c \wedge \ c.val \ \geq \ e \ \wedge (\nexists \, n \ : \ Head \overset{*}{\Rightarrow} \ n \ \wedge e \leq n.val < c.val) \ .$$

In a shared state satisfying Φ_c, c is the node with the smallest value greater than or equal to e in the list. The base condition for *contains*'s *return true* step is $\Phi_c \wedge c.val = e$, and the base condition for *return false* is the predicate $\Phi_c \wedge (c.marked \vee c.val \neq e)$.

These predicates are base conditions since every sequential execution from a shared state satisfying them reaches the same return step, i.e., if c is the node in the list with the smallest value that is greater than or equal to e and is reachable from the head of the list, then after traversing the list and reaching it, the return step is determined according to its value.

3.5.2 Stage II: Linearizability of Update Operations

We next prove that Lazy List has linearizable update operations. Using Definition 3.4.4, it suffices to show the following: (1) each update operation has a base point for its update and return steps, (2) each critical sequence commutes with interleaved critical sequences, and (3) the update steps are base point preserving for operations with interleaved critical sequences.

Base Points for Update and Return Steps

Proof Sketch First we claim that in every execution of an *add* (*remove*) operation, line 14 (44, respectively), is a base point for all the operation's update and return steps.

Claim 1. *Consider the shared state $\underline{s}$ immediately after line 14 (44) of an execution of add(e) (remove(e)). Then $\Phi(\underline{s}, n_1, n_2, e)$ is true.*

Claim 1 can be proven by induction on the steps of an execution. Intuitively, the idea is to show by induction that the list is sorted, and that in each *add* (*remove*) operation, *locate* locks the two nodes and verifies that they are unmarked, and so no other operation can change them and they remain reachable from the head of the list and connected to each other. Formal proofs of this claim were given in [67, 78].

Based on Claim 1 and the observation that after line 14 (44) of an execution of *add(e)* (*remove(e)*) the value of $n_2.val$ persists until n_2 is unlocked, we conclude that the shared state after *locate* returns is a base point for update operations' update and return steps. Since the locked nodes cannot be modified by concurrent operations, the pre-state of the first update step is also a base point for the same steps. In case the update operation has no update steps, the same holds for the last read step.

Commutative and Base Point Preserving Steps

Proof Sketch We now show that the steps of update operations that have interleaved updates are commutative, and that the update steps are base point preserving. Specifically, we examine the steps between the first update step and the last one (or just the last read step in case of an update operation that does no have update steps).

In order to add a key to the list, an update operation locks the predecessor and successor of the new node. For removing a node from the list, the update operation locks the node and it predecessor. This means that every update operation locks the nodes that it changes and the nodes that it relies upon before it verifies its steps' base point. Thus, update operations have concurrent critical sequence only if they access different nodes. Therefore their steps commute, and are base point preserving for one another.

3.5.3 Stage III: Linearizability of Read-Only Operations

The final stage in our proof is to show the conditions stated in Theorem 3.1 hold for each read-only operation.

Single Non-Preserving Step per Update Operation First we show that every update operation of Lazy List has at most one step that is not base point preserving for all read-only operations.

Proof Sketch We only need to consider update steps, since every other step in *add* and *remove* does not modify the shared memory, and therefore does not affect any base condition of *contains*. There are two update steps in an operation. In *add*, the first update step allocates a new (unreachable) node. Nodes that are not reachable from the head of the list do not affect any base condition. Therefore, only the second step, the one that changes the list, is not base point preserving for *contains*.

In *remove*, the first update step marks the removed node, and the second makes the node unreachable from the head of the list. Since marked nodes are treated in every base condition of *contains* as if they are already detached from the list, the second update step does not change the truth value of the base condition of contains. More precisely, if we compare the second update step's pre-state to its post-state, they both satisfy the same base conditions of *contains*'s return steps.

Concurrent Base Points Last, we show that in every execution of *contains*, the return step of *contains* has a base point, and that base point occurs between the pre-state of *contains*'s invoke step and the pre-state of *contains*'s return step.

Proof Sketch When *add* inserts a new value to the list, it locks the predecessor node n and the successor m, and verifies that n and m are not marked and that $n.next = m$.

Since n or m cannot be removed as long as they are locked, and since nodes are removed only when their predecessor is also locked, new nodes are not added to detached parts of the list. This means that every node encountered during a traversal of the list was reachable from the head at some point.

In addition, if *add* inserts a value e, it satisfies $n.val < e < m.val$, since n and m are locked, and no value other than e is inserted between them before e is added (this can be proven by induction on executions).

The execution of *contains(e)* reaches line 6 only after it traverses the list from its head and reaches the first node c whose value v satisfies $e \leq v$. Thus, there is some concurrent shared state $\underline{s}$ that occurs after the invocation of *contains(e)* in which c is unmarked and reachable from the head of the list. State $\underline{s}$ is a base-point of *contains(e)*'s return step.

Acknowledgements

We thank Naama Kraus, Noam Rinetzky and the anonymous reviewers for helpful comments and suggestions.

Chapter 4

Paper: Modular composition of coordination services

Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer : "Modular composition of coordination services". In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16). USENIX Association, Berkeley, CA, USA, 251-264.

In this paper we design, implement, and evaluate ZooKeepers' consistent composition.

Modular Composition of Coordination Services

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

Coordination services like ZooKeeper, etcd, Doozer, and Consul are increasingly used by distributed applications for consistent, reliable, and high-speed coordination. When applications execute in multiple geographic regions, coordination service deployments trade-off between performance, (achieved by using independent services in separate regions), and consistency.

We present a system design for modular composition of services that addresses this trade-off. We implement ZooNet, a prototype of this concept over ZooKeeper. ZooNet allows users to compose multiple instances of the service in a consistent fashion, facilitating applications that execute in multiple regions. In ZooNet, clients that access only local data suffer no performance penalty compared to working with a standard single ZooKeeper. Clients that use remote and local ZooKeepers show up to 7.5x performance improvement compared to consistent solutions available today.

4.1 Introduction

Many applications nowadays rely on coordination services such as ZooKeeper [50], etcd [11], Chubby [34], Doozer [6], and Consul [4]. 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.

When applications span multiple data centers, one is faced with a choice between sacrificing performance, as occurs in a cross data center deployment, and forgoing consistency

by running coordination services independently in the different data centers. For many applications, the need for consistency outweighs its cost. For example, Akamai [75] and Facebook [76] use strongly-consistent globally distributed coordination services (Facebook's Zeus is an enhanced version of ZooKeeper) for storing configuration files; dependencies among configuration files mandate that multiple users reading such files get consistent versions in order for the system to operate properly. Other examples include global service discovery [3], storage of access-control lists [9] and more.

In this work we leverage the observation that, nevertheless, such workloads tend to be highly partitionable. For example, configuration files of user or email accounts for users in Asia will rarely be accessed outside Asia. Yet currently, systems that wish to ensure consistency in the rare cases of remote access, (like [75, 76]), globally serialize all updates, requiring multiple cross data center messages.

To understand the challenge in providing consistency with less coordination, consider the architecture and semantics of an individual coordination service. Each coordination service is typically replicated for high-availability, and clients submit requests to one of the replicas. Usually, update requests are serialized via a quorum-based protocol such as Paxos [55], Zab [51] or Raft [68]. Reads are served locally by any of the replicas and hence can be somewhat stale but nevertheless represent a valid snapshot. This design entails the typical semantics of coordination services [4, 11, 50] – atomic (linearizable [49]) updates and sequentially-consistent [53] reads. Although such weaker read semantics enable fast local reads, this property makes coordination services non-composable: correct coordination services may fail to provide consistency when combined. In other words, a workload accessing *multiple* consistent coordination services may not be consistent, as we illustrate in Section 4.2. This shifts the burden of providing consistency back to the application, beating the purpose of using coordination services in the first place.

In Section 4.3 we present a system design for modular composition of coordination services, which addresses this challenge. We propose deploying a single coordination service instance in each data center, which is shared among many applications. Each application partitions its data among one or more coordination service instances to maximize operation locality. Distinct coordination service instances, either within a data center or geo-distributed, are then composed in a manner that guarantees global consistency. Consistency is achieved on the client side by judiciously adding synchronization requests. The overhead incurred by a client due to such requests depends on the frequency with which that client issues read requests to *different* coordination services. In particular, clients that use a single coordination service do not pay any price.

In Section 4.4 we present ZooNet, a prototype implementation of our modular composition for ZooKeeper. ZooNet implements a client-side library that enables composing multiple

ZooKeeper ensembles, (i.e., service instances), in a consistent fashion, facilitating data sharing across geographical regions. Each application using the library may compose ZooKeeper ensembles according to its own requirements, independently of other applications. Even though our algorithm requires only client-side changes, we tackle an additional issue, specific to ZooKeeper – we modify ZooKeeper to provide better isolation among clients. While not strictly essential for composition, this boosts performance of both stand-alone and composed ZooKeeper ensembles by up to 10x. This modification has been contributed back to ZooKeeper [15] and is planned to be released in ZooKeeper 3.6.

In Section 4.5 we evaluate ZooNet. Our experiments show that under high load and high spatial or temporal locality, ZooNet achieves the same performance as an inconsistent deployment of independent ZooKeepers (modified for better isolation). This means that our support for consistency comes at a low performance overhead. In addition, ZooNet shows up to 7.5x performance improvement compared to a consistent ZooKeeper deployment (the "recommended" way to deploy ZooKeeper across data centers [13]).

We discuss related work in Section 4.6.

In summary, this paper makes the following contributions:

- A system design for composition of coordination services that maintains their semantics.
- A significant improvement to ZooKeeper's server-side isolation and concurrency.
- ZooNet – a client-side library to compose multiple ZooKeepers.

4.2 Background

We discuss the service and semantics offered by coordination services in Section 4.2.1, and then proceed to discuss possible ways to deploy them in a geo-distributed setting in Section 4.2.2.

4.2.1 Coordination Services

Coordination services are used for maintaining shared state in a consistent and fault-tolerant manner. Fault tolerance is achieved using replication, which is usually done by running a quorum-based state-machine replication protocol such as Paxos [55] or its variants [51, 68].

In Paxos, the history of state updates is managed by a set of servers called *acceptors*, s.t. every update is voted on by a quorum (majority) of acceptors. One acceptor serves as *leader* and manages the voting process. In addition to acceptors, Paxos has *learners* (called *observers* in ZooKeeper and *proxies* in Consul), which are light-weight services that do not participate in voting and get notified of updates after the quorum accepts them. In the

context of this paper, acceptors are also (voting) learners, i.e., they learn the outcomes of votes.

Coordination services are typically built on top of an underlying key-value store and offer read and update (read-modify-write) operations. The updates are linearizable, i.e., all acceptors and learners see the same sequence of updates and this order conforms to the real-time order of the updates. The read operations are sequentially consistent, which is a weaker notion similar to linearizability in that an equivalent sequential execution must exist, but it must only preserve the program order of each individual client and not the global real-time order. A client can thus read a stale value that has already been overwritten by another client. These weaker semantics are chosen in order to allow a single learner or acceptor to serve reads locally. This motivates using learners in remote data centers – they offer fast local reads without paying the cost of cross data center voting.

As an aside, we note that some coordination service implementations offer their clients an asynchronous API. This is a client-side abstraction that improves performance by masking network delays. At the server-side, each client's requests are handled sequentially, and so the interaction is well-formed, corresponding to the standard correctness definitions of linearizability and sequential consistency.

Unfortunately, these semantics of linearizable updates and sequentially consistent reads are not composable, i.e., a composition of such services does not satisfy the same semantics. This means that the clients cannot predict the composed system's behavior. As an example, consider two clients that perform operations concurrently as we depict in Figure 4.1. Client 1 updates object x managed by coordination service s_1, and then reads an old version of object y, which is managed by service s_2. Client 2 updates y and then reads an old version of x. While the semantics are preserved at both s_1 and s_2 (recall that reads don't have to return the latest value), the resulting execution violates the service semantics since there is no equivalent sequential execution: the update of y by client 2 must be serialized after the read of y by client 1 (otherwise the read should have returned 3 and not 0), but then the read of x by client 2 appears after the update of x by client 1 and therefore should have returned 5.

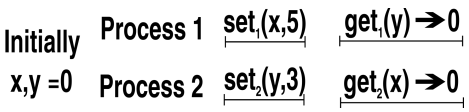
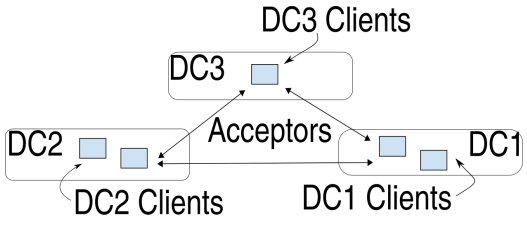


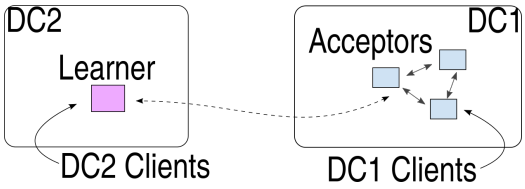
Figure 4.1: Inconsistent composition of two coordination services holding objects x and y: each object is consistent by itself, but there is no equivalent sequential execution.

Alternative	Performance		Correctness	Availability during partitions	
	Updates	Reads		Updates	Reads
`Single Service`	Very slow	Fast	Yes	In majority	Everywhere
`Learners`	Slow	Fast	Yes	In acceptors	Everywhere
`Multiple Services`	Fast	Fast	No	Local	Everywhere
`Modular Composition`	Fast	Fast	Yes	Local	Local

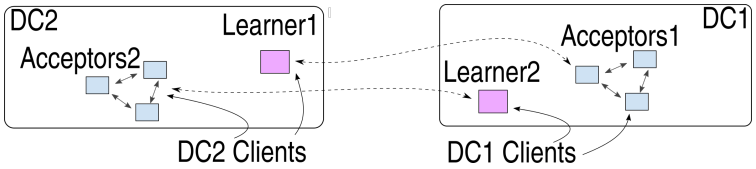
Table 4.1: Comparison of different alternatives for coordination service deployments across data centers. The first three alternatives are depicted in Figure 4.2. Our design alternative, *modular composition*, is detailed in Section 4.3.



(a) Single Service – Coordination service acceptors are deployed in all data centers, no single point of failure.



(b) Learners – Coordination service acceptors are deployed in one data center and learners in all other data centers.



(c) Multiple Services – A single coordination service is deployed in each data center and a learner is deployed in every other data center.

Figure 4.2: Different alternatives for coordination service deployment across data centers.

4.2.2 Cross Data Center Deployment

When coordination is required across multiple data centers over WAN, system architects currently have three main deployment alternatives. In this section we discuss these alternatives with respect to their performance, consistency, and availability in case of partitions. A summary of our comparison is given in Table 4.1.

Alternative 1 – Single Coordination Service A coordination service can be deployed over multiple geographical regions by placing its acceptors in different locations (as done, e.g., in Facebook's Zeus [76] or Akamai's ACMS [75]), as we depict in Figure 4.2a. Using a single coordination service for all operations guarantees consistency.

This setting achieves the best availability since no single failure of a data center takes down all acceptors. But in order to provide availability following a loss or disconnection of any single data center, more than two locations are needed, which is not common.

With this approach, voting on each update is done across WAN, which hampers latency

and wastes WAN bandwidth, (usually an expensive and contended resource). In addition, performance is sensitive to placement of the leader and acceptors, which is frequently far from optimal [73]. On the other hand, reads can be served locally in each partition.

Alternative 2 – Learners A second option is to deploy all of the acceptors in one data center and learners in others, as we depict in Figure 4.2b. In fact, this architecture was one of the main motivations for offering learners (observers) in ZooKeeper [13]. As opposed to acceptors, a learner does not participate in the voting process and it only receives the updates from the leader once they are committed. Thus, cross data center consistency is preserved without running costly voting over WAN. Often, alternatives 1 and 2 are combined, such as in Spanner [37], Megastore [27] and Zeus [76].

The update throughput in this deployment is limited by the throughput of one coordination service, and the update latency in remote data centers is greatly affected by the distance between the learners and the leader. In addition, in this approach we have a single point of failure, i.e., if the acceptors' data center fails or a network partition occurs, remote learners are only able to serve read requests.

Alternative 3 – Multiple Coordination Services In the third approach data is partitioned among several independent coordination services, usually one per data center or region, each potentially accompanied by learners in remote locations, as depicted in Figure 4.2c. In this case, each coordination service processes only updates for its own data partition and if applications in different regions need to access unrelated items they can do so independently and in parallel, which leads to high throughput. Moreover, if one cluster fails all other locations are unaffected. Due to these benefits, multiple production systems [3, 17, 21] follow this general pattern. The disadvantage of this design is that it does not guarantee the coordination service's consistency semantics, as explained in Section 4.2.1.

4.3 Design for Composition

In Section 4.3.1 we describe our design approach and our client-side algorithm for modular composition of coordination services while maintaining consistency. In Section 4.3.2 we discuss the properties of our design, namely correctness (a formal proof is given in an online Technical Report [59]), performance, and availability.

4.3.1 Modular Composition of Services

Our design is based on multiple coordination services (as depicted in Figure 4.2c), to which we add client-side logic that enforces consistency.

Our solution achieves consistency by injecting *sync* requests, which are non-mutating update operations. If the coordination service itself does not natively support such operations, they can be implemented using an update request addressed to a dummy object.

The client-side logic is implemented as a layer in the coordination service client library, which receives the sequential stream of client requests before they are sent to the coordination service. It is a state machine that selectively injects sync requests prior to some of the reads. Intuitively, this is done to bound the staleness of ensuing reads. In Algorithm 1, we give a pseudo-code for this layer at a client accessing multiple coordination services, each of which has a unique identifier.

An injected sync and ensuing read may be composed into a single operation, which we call *synced read*. A synced read can be implemented by buffering the local read request, sending a sync (or non-mutating update) to the server, and serving the read immediately upon receipt of a commit for the sync request. Some coordination services natively support such synced reads, e.g., Consul calls them consistent reads [5]. If all reads are synced the execution is linearizable. Our algorithm only makes some of the reads synced to achieve coordination service's semantics with minimal synchronization overhead.

Since each coordination service orders requests independently, concurrent processing of a client's updates at two coordination services may inverse their order. To avoid such re-ordering (as required, e.g., by ZooKeeper's FIFO program order guarantee), we refrain from asynchronously issuing updates to a new coordination service before responses to earlier requests arrive. Rather, we buffer requests whenever we identify a new coordination service target for as long as there are pending requests to other coordination services. This approach also guarantees that coordination service failures do not introduce gaps in the execution sequence of asynchronous requests.

4.3.2 Modular Composition Properties

We now discuss the properties of our modular composition design.

Correctness

The main problem in composing coordination services is that reads might read "from the past", causing clients to see updates of different coordination services in a different order, as depicted in Figure 4.1. Our algorithm adds sync operations in order to make ensuing reads "read from the present", i.e., read at least from the sync point. We do this every time a client's read request accesses a different coordination service than the previous request. Subsequent reads from the same coordination service are naturally ordered after the first, and so no additional syncs are needed.

```
1: lastService ← nil                          // Last service this client accessed
2: numOutstanding ←0                          // #outstanding requests to lastService
3: onUpdate(targetService, req)
4:    if targetService ≠ lastService then
5:       // Wait until all requests to previous service complete
6:       wait until numOutstanding = 0
7:       lastService ← targetService
8:    numOutstanding++
9:    send req to targetService
10: onRead(targetService, req)
11:   if targetService ≠ lastService then
12:      // Wait until all requests to previous service complete
13:      wait until numOutstanding = 0
14:      lastService ← targetService
15:      numOutstanding++
16:      // Send sync before read
17:      send sync to targetService
18:   numOutstanding++
19:   send req to targetService
20: onResponse(req)
21:   numOutstanding−−
```

Algorithm 1: Modular composition, client-side logic.

In Figure 4.3 we depict the same operations as in Figure 4.1 with sync operations added according to our algorithm. As before, client 1 updates object x residing in service s_1 and then reads y from service s_2. Right before the read, the algorithm interjects a sync to s_2. Similarly, client 2 updates y on s_2, followed by a sync and a read from s_1. Since s_2 guarantees update linearizability and client 1's sync starts after client 2's update of y completes, reads made by client 1 after the sync will retrieve the new state, in this case 3. Client 2's sync, on the other hand, is concurrent with client 1's update of x, and therefore may be ordered either before or after the update. In this case, we know that it is ordered before the update, since client 2's read returns 0. In other words, there exists an equivalent sequential execution that consists of client 2's requests followed by client 1's requests, and this execution preserves linearizability of updates (and syncs) and sequential consistency of read requests, as required by the coordination service's semantics. See [59] for a formal discussion.

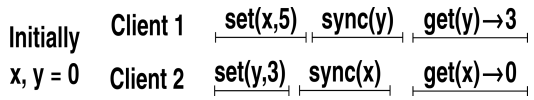


Figure 4.3: Consistent modular composition of two coordination services holding objects x and y (as in Figure 4.1): adding syncs prior to reads on new coordination services ensures that there is an equivalent sequential execution.

Performance

By running multiple independent coordination services, the modular composition can potentially process requests at a rate as high as the sum of the individual throughputs. However, sync requests take up part of this bandwidth, so the net throughput gain depends on the frequency with which syncs are sent.

The number of syncs corresponds to the temporal locality of the workload, since sync is added only when the accessed coordination service changes.

Read latency is low (accessing a local acceptor or learner) when the read does not necessitate a sync, and is otherwise equal to the latency of an update.

Availability

Following failures or partitions, each local coordination service (where a quorum of acceptors remains available and connected) can readily process update and read requests submitted by local clients. However, this may not be the case for remote client requests: If a learner in data center A loses connectivity with its coordination service in data center B, sync requests submitted to the learner by clients in A will fail and these clients will be unable to access the coordination service.

Some coordination services support state that corresponds to active client sessions, e.g., an ephemeral node in ZooKeeper is automatically deleted once its creator's session terminates. Currently, we do not support composition semantics for such session-based state: clients initiate a separate session with each service instance they use, and if their session with one ZooKeeper ensemble expires (e.g., due to a network partition) they may still access data from other ZooKeepers. Later, if the session is re-instated they may fail to see their previous session-based state, violating consistency. A possible extension addressing this problem could be to maintain a single virtual session for each client, corresponding to the composed service, and to invalidate it together with all the client's sessions if one of its sessions terminates.

4.4 ZooNet

We implement *ZooNet*, a modular composition of ZooKeepers. Though in principle, modular composition requires only client-side support, we identified a design issue in ZooKeeper that makes remote learner (observer) deployments slow due to poor isolation among clients. Since remote learners are instrumental to our solution, we address this issue in the ZooKeeper server, as detailed in Section 4.4.1. We then discuss our client-side code in Section 4.4.2.

4.4.1 Server-Side Isolation

The original ZooKeeper implementation stalls reads when there are concurrent updates by other clients. Generally speaking, reads wait until an update is served even when the semantics do not require it. In Section 4.4.1 we describe this problem in more detail and in Section 4.4.1 we present our solution, which we have made available as a patch to ZooKeeper [15] and has been recently committed to ZooKeeper's main repository.

ZooKeeper's Commit Processor

ZooKeeper servers consist of several components that process requests in a pipeline. When an update request arrives to a ZooKeeper server from a client, the server forwards the update to the leader and places the request in a local queue until it hears from the leader that voting on the update is complete (i.e., the leader has *committed* the request). Only at that point can the update be applied to the local server state. A component called *commit processor* is responsible for matching incoming client requests with commit responses received from the leader, while maintaining the order of operations submitted by each client.

In the original implementation of the commit processor, (up to ZooKeeper version 3.5.1-alpha), clients are not isolated from each other: once some update request reaches the head of the request stream, all pending requests by all clients connected to this server stall until a commit message for the head request arrives from the leader. This means that there is a period, whose duration depends on the round-trip latency between the server and the leader plus the latency of quorum voting, during which all requests are stalled. While the commit processor must maintain the order of operations submitted by each client, enforcing order among updates of *different* clients is the task of the leader. Hence, blocking requests of other clients in this situation, only because they were unlucky enough to connect via the same server, is redundant.

In a geo-distributed deployment, this approach severely hampers performance as it does not allow read operations to proceed concurrently with long-distance concurrent updates. In the context of modular composition, it means that syncs hamper read-intensive workloads, i.e., learners cannot serve reads locally concurrently with syncs and updates.

Commit Processor Isolation

We modified ZooKeeper's commit processor to keep a separate queue of pending requests per client. Incoming reads for which there is no preceding pending update by the same client, (i.e., an update for which a commit message has not yet been received), are not blocked. Instead, they are forwarded directly to the next stage of the pipeline, which responds to the client based on the current server state.

Read requests of clients with pending updates are enqueued in the order of arrival in the appropriate queue. For each client, whenever the head of the queue is either a committed update or a read, the request is forwarded to the next stage of the server pipeline. Updates are marked committed according to the order of commit messages received from the leader (the linearization order). For more details, see our ZooKeeper Jira [15].

4.4.2 The ZooNet Client

We prototyped the ZooNet client as a wrapper for ZooKeeper's Java client library. It allows clients to establish sessions with multiple ZooKeeper ensembles and maintains these connections. Users specify the target ZooKeeper ensemble for every operation as a znode path prefix. Our library strips this prefix and forwards the operation to the appropriate ZooKeeper, converting some of the reads to synced reads in accordance with Algorithm 1. Our sync operation performs a dummy update; we do so because ZooKeeper's sync is not a linearizable update [50]. The client wrapper consists of roughly 150 lines of documented code.

4.5 Evaluation

We now evaluate our modular composition concept using the ZooNet prototype. In Section 4.5.1 we describe the environment in which we conduct our experiments. Section 4.5.2 evaluates our server-side modification to ZooKeeper, whereas Section 4.5.3 evaluates the cost of the synchronization introduced by ZooNet's client. Finally, Section 4.5.4 compares ZooNet to a single ZooKeeper ensemble configured to ensure consistency using remote learners (Figure 4.2b).

4.5.1 Environment and Configurations

We conduct our experiments on Google Compute Engine [12] in two data centers, DC1 in eastern US (South Carolina) and DC2 in central US (Iowa). In each data center we allocate five servers: three for a local ZooKeeper ensemble, one for a learner connected to the remote data center, and one for simulating clients (we run 30 request-generating client threads in each data center). Each server is allocated a standard 4 CPU machine with 4 virtual CPUs and 15 GB of memory. DC1 servers are allocated on a 2.3 GHz Intel Xeon E5 v3 (Haswell) platform, while DC2 servers are allocated on a 2.5GHz Intel Xeon E5 v2 (Ivy Bridge). Each server has two standard persistent disks. The Compute Engine does not provide us with information about available network bandwidth between the servers. We use the latest version of ZooKeeper to date, version 3.5.1-alpha.

We benchmark throughput when the system is saturated and configured as in ZooKeeper's original evaluation (Section 5.1 in [50]). We configure the servers to log requests to one disk while taking snapshots on another. Each client thread has at most 200 outstanding requests at a time. Each request consists of a read or an update of 1KB of data. The operation type and target coordination service are selected according to the workload specification in each experiment.

4.5.2 Server-Side Isolation

In this section we evaluate our server-side modification given in Section 4.1. We study the learner's throughput with and without our change. Recall that the learner (observer in ZooKeeper terminology) serves as a fast local read cache for distant clients, and also forwards update requests to the leader.

We experiment with a single ZooKeeper ensemble running three acceptors in DC1 and an observer in DC2. Figure 4.4 compares the learner's throughput with and without our modification, for a varying percentage of reads in the workload. DC1 clients have the same workload as DC2 clients.

Our results show that for read-intensive workloads that include some updates, ZooNet's learner gets up to around 4x higher throughput by allowing concurrency between reads and updates of different clients, and there is 30% up to 60% reduction in the tail latency. In a read-only workload, ZooNet does not improve the throughput or the latency, because ZooKeeper does not stall any requests. In write-intensive workloads, reads are often blocked by preceding pending updates by the same client, so few reads can benefit from our increased parallelism.

Our Jira [15] provides additional evaluation (conducted on Emulab [81]) in which we show that the throughput speedup for local clients can be up to 10x in a single data center deployment of ZooKeeper. Moreover, ZooNet significantly reduces read and write latency in mixed workloads in which the write percentage is below 30 (for reads, we get up to 96% improvement, and for writes up to 89%).

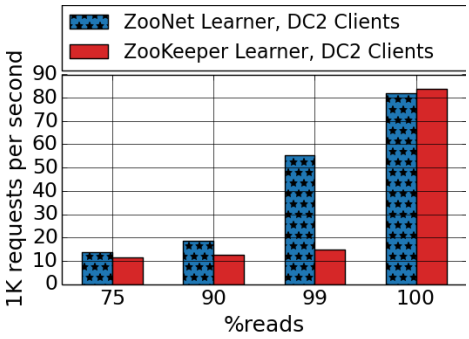


Figure 4.4: Improved server-side isolation. Learner's throughput as a function of the percentage of reads.

4.5.3 The Cost of Consistency

ZooNet is a composition of independent ZooKeepers, as depicted in Figure 4.2c, with added sync requests. In this section we evaluate the cost of the added syncs by comparing our algorithm to two alternatives: (1) Sync-All, where all reads are executed as synced reads, and (2) Never-Sync, in which clients never perform synced reads.

Never-Sync in not sequentially consistent (as illustrated in Figure 4.1). It thus corresponds to the fastest but inconsistent ZooKeeper deployment (Figure 4.2c), with ZooKeeper patched to improve isolation. At the other extreme, by changing all reads to be synced, Sync-All guarantees linearizability for all operations, including reads. ZooNet provides a useful middle ground (supported by most coordination services in the single-data center setting), which satisfies sequential consistency for all operations and linearizability for updates.

As a sanity check, we study in Section 4.5.3 a fully partitionable workload with clients accessing only local data in each data center. In Section 4.5.3 we have DC1 clients perform only local operations, and DC2 clients perform both local and remote operations.

Local Workload

In Figure 4.5 we depict the saturation throughput of DC1 (solid lines) and DC2 (dashed lines) with the three alternatives.

ZooNet's throughput is identical to that of Never-Sync in all workloads, at both data centers. This is because ZooNet sends sync requests only due to changes in the targeted ZooKeeper, which do not occur in this scenario. Sync-All has the same write-only throughput (leftmost data point). But as the rate of reads increases, Sync-All performs more synced reads, resulting in a significant performance degradation (up to 6x for read-only workloads). This is because a read can be served locally by any acceptor (or learner), whereas each synced read, similarly to an update, involves communication with the leader and a quorum.

The read-only throughput of ZooNet and Never-Sync is lower than we expect: since in this scenario the three acceptors in each data center are dedicated to read requests, we would expect the throughput to be 3x that of a single learner (reported in Figure 4.4). We hypothesize that the throughput is lower in this case due to a network bottleneck.

Remote Data Center Access

When clients access remote data, synced reads kick-in and affect performance. We now evaluate the cost of synced reads as a function of workload locality. We define two workload parameters: *local operations*, which represents spatial locality, namely the percentage of requests that clients address to their local data center, and *burst*, which represents the temporal locality of the target ZooKeeper. For simplicity, we consider a fixed burst size,

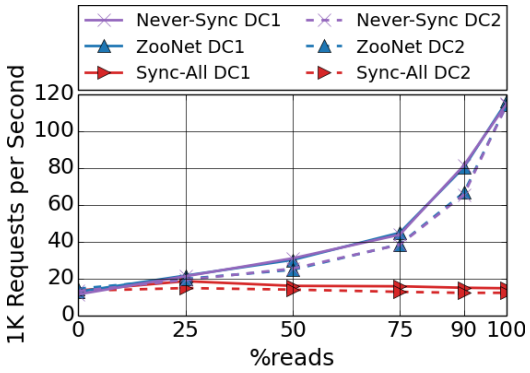


Figure 4.5: Saturated ZooNet throughput at two data centers with local operations only. In this sanity check we see that the performance of Never-Sync is identical to ZooNet's performance when no syncs are needed.

where the client sends *burst* requests to the same ZooKeeper and then chooses a new target ZooKeeper according to the local operations ratio. Note that a burst size of 1 represents the worst-case scenario for ZooNet, while with high burst sizes, the cost of adding syncs is minimized.

Our design is optimized for partitionable workloads where spatial locality is high by definition since clients rarely access data in remote partitions. In ZooKeeper, another factor significantly contributes to temporal locality: ZooKeeper limits the size of each data object (called znode) to 1MB, which causes applications to express stored state using many znodes, organized in a hierarchical manner. ZooKeeper intentionally provides a minimalistic API, so programs wishing to access stored state (e.g., read the contents of a directory or sub-tree) usually need to make multiple read requests to ZooKeeper, effectively resulting in a high burst size.

In Figure 4.6 we compare ZooNet to Sync-All and Never-Sync with different burst sizes where we vary the local operations ratio of DC2 clients. DC1 clients perform 100% local operations. We select three read ratios for this comparison: a write-intensive workload in which 50% of the requests are updates (left column), a read-intensive workload in which 90% of the requests are reads (middle column), and a read-only workload (right column). DC1 clients and DC2 clients have the same read ratio in each test.

Results show that in a workload with large bursts of 25 or 50 (bottom two rows), the addition of sync requests has virtually no effect on throughput, which is identical to that of Never-Sync except in read-intensive workloads, where with a burst of 25 there is a slight throughput degradation when the workload is less than 80% local.

When there is no temporal locality (burst of 1, top row), the added syncs induce a high performance cost in scenarios with low spatial locality, since they effectively modify the workload to become write-intensive. In case most accesses are local, ZooNet seldom adds syncs, and so it performs as well as Never-Sync regardless of the burst size.

All in all, ZooNet incurs a noticeable synchronization cost only if the workload shows no locality whatsoever, neither temporal nor spatial. Either type of locality mitigates this cost.

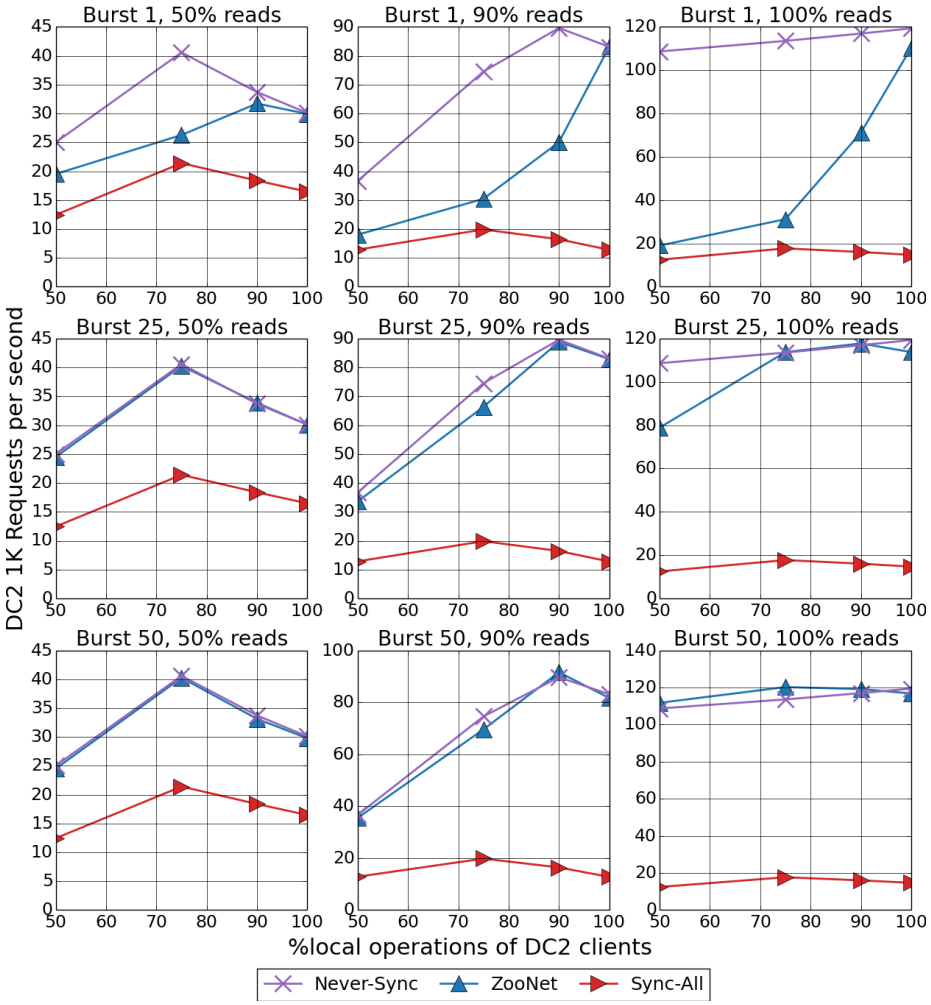


Figure 4.6: Throughput of ZooNet, Never-Sync and Sync-All. Only DC2 clients perform remote operations.

4.5.4 Comparing ZooNet with ZooKeeper

We compare ZooNet with the fastest cross data center deployment of ZooKeeper that is also consistent, i.e., a single ZooKeeper ensemble where all acceptors are in DC1 and a learner is located in DC2 (Figure 4.2b). The single coordination service deployment (Figure 4.2a) is less efficient since: (1) acceptors participate in the voting along with serving clients (or, alternatively, more servers need to be deployed as learners as in [76]); and (2) the voting is done over WAN (see [13] for more details). We patch ZooKeeper with the improvement described in Section 4.4.1 and set the burst size to 50 in order to focus the current discussion on the impact that data locality has on performance.

We measure aggregate client throughput and latency in DC1 and DC2 with ZooKeeper and ZooNet, varying the workload's read ratio and the fraction of local operations of the clients in DC2. We first run a test where all operations of clients in DC1 are local. Figure 4.7a

shows the throughput speedup of ZooNet over ZooKeeper at DC1 clients, and Figure 4.7b shows the throughput speedup for DC2 clients.

Our results show that in write-intensive workloads, DC2 clients get up to 7.5x higher throughput and up to 92% reduction in latency. This is due to the locality of update requests in ZooNet, compared to the ZooKeeper deployment in which each update request of a DC2 client is forwarded to DC1. The peak throughput saturates at the update rate that a single leader can handle. Beyond that saturation point, it is preferable to send update operations to a remote DC rather than have them handled locally, which leads to a decrease in total throughput.

In read-intensive workloads (90% – 99% reads), DC2 clients also get a higher throughput with ZooNet (4x to 2x), and up to 90% reduction in latency. This is due to the fact that in ZooKeeper, a single learner can handle a lower update throughput than three acceptors. In read-only workloads, the added acceptors have less impact on throughput; we assume that this is due to a network bottleneck as observed in our sanity check above (Figure 4.5).

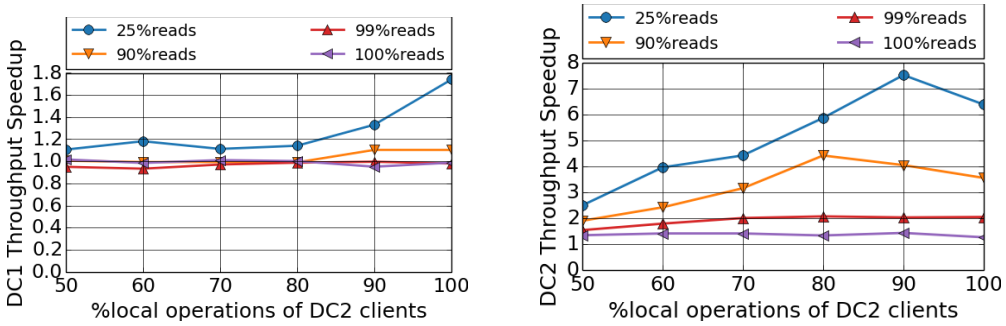
In addition, we see that DC1 clients are almost unaffected by DC2 clients in read-intensive workloads. This is due to the fact that with both ZooKeeper and ZooNet, reads issued by clients in DC2 are handled locally in DC2. The added synced reads add negligible load to the acceptors in DC1 due to the high burst size and locality of requests (nevertheless, they do cause the throughput speedup to drop slightly below 1 when there is low locality). With a write-intensive workload, DC1 clients have a 1.7x throughput speedup when DC2 clients perform no remote operations. This is because remote updates of DC2 clients in ZooKeeper add to the load of acceptors in DC1, whereas in ZooNet some of these updates are local and processed by acceptors in DC2.

Finally, we examine a scenario where clients in both locations perform remote operations. Figure 4.8a shows the throughput speedup of ZooNet over ZooKeeper achieved at DC1 clients, and Figure 4.8b shows the throughput speedup of DC2 clients. All clients have the same locality ratio. Each curve corresponds to a different percentage of reads.

There are two differences between the results in Figure 4.8 and Figure 4.7. First, up to a local operations ratio of 75%, DC1 clients suffer from performance degradation in read-intensive workloads. This is because in the ZooKeeper deployment, all the requests of DC1 clients are served locally, whereas ZooNet serves many of them remotely. This re-emphasizes the observation that ZooNet is most appropriate for scenarios that exhibit locality, and is not optimal otherwise.

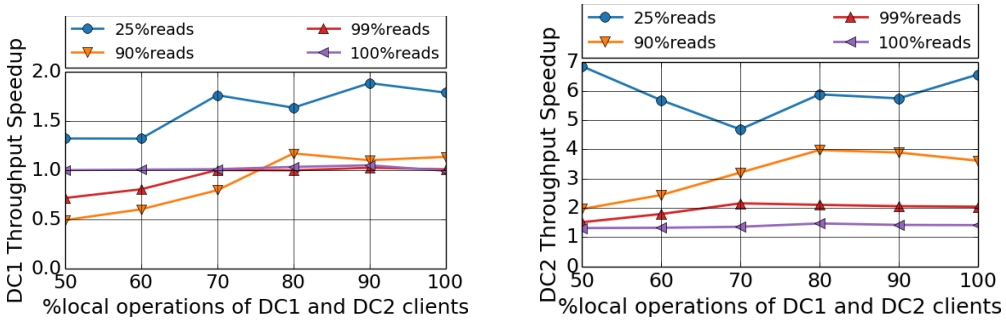
Second, the DC1 leader is less loaded when DC1 clients also perform remote updates (Figure 4.8). This mostly affects write-intensive scenarios (top blue curve), in which the leaders at both data centers share the update load, leading to higher throughput for all clients. Indeed, this yields higher throughput speedup when locality is low (leftmost data

point in Figures 4.8a and 4.8b compared to Figures 4.7a and 4.7b, respectively). As locality increases to 70%–80%, the DC2 leader becomes more loaded due to DC2's updates, making the throughput speedup in Figures 4.7b and Figure 4.8b almost the same, until with 100% local updates (rightmost data point), the scenarios are identical and so is the throughput speedup.



(a) Throughput speedup of DC1 clients. **(b)** Throughput speedup of DC2 clients.

Figure 4.7: Throughput speedup (ZooNet/ZooKeeper). DC1 clients perform only local operations. The percentage of read operations is identical for DC1 clients and DC2 clients.



(a) Throughput speedup of DC1 clients. **(b)** Throughput speedup of DC2 clients.

Figure 4.8: Throughput speedup (ZooNet/ZooKeeper). DC1 clients an DC2 clients have the same local operations ratio as well as read operations percentage.

4.6 Related Work

Coordination services such as ZooKeeper [50], Chubby [34], etcd [11], and Consul [4] are extensively used in industry. Many companies deploying these services run applications in multiple data centers. But questions on how to use coordination services in a mutli-data center setting arise very frequently [3, 10, 19–21], and it is now clear that the designers of coordination services must address this use-case from the outset.

In what follows we first describe the current deployment options in Section 4.6.1 followed by a discussion of previously proposed composition methods in Section 4.6.2.

A large body of work, e.g., [52, 60, 61], focuses on improving the efficiency of coordination services. Our work is orthogonal – it allows combining multiple instances to achieve a single

system abstraction with the same semantics, while only paying for coordination when it is needed.

4.6.1 Multi-Data Center Deployment

In Section 4.2 we listed three prevalent strategies for deploying coordination services across multiple data centers: a single coordination service where acceptors are placed in multiple data centers, a single coordination service where acceptors run in one data center, or multiple coordination services. The choice among these options corresponds to the tradeoff system architects make along three axes: consistency, availability, and performance (a common interpretation of the CAP theorem [8]). Some are willing to sacrifice update speed for consistency and high availability in the presence of data center failures [27, 37, 75, 76]. Others prefer to trade-off fault-tolerance for update speed [13], while others prioritize update speed over consistency [3, 21]. In this work we mitigate this tradeoff, and offer a fourth deployment option whose performance and availability are close to that of the third (inconsistent) option, without sacrificing consistency.

Some systems combine more than one of the deployment alternatives described in Section 4.2. For example, Vitess [18] deploys multiple local ZooKeeper ensembles (as in Figure 4.2c) in addition to a single global ensemble (as in Figure 4.2a). The global ensemble is used to store global data that doesn't change very often and needs to survive a data center failure. A similar proposal has been made in the context of SmartStack, Airbnb's service discovery system [16]. ZooNet can be used as-is to combine the local and global ensembles in a consistent manner.

Multiple studies [69, 82] showed that configuration errors and in particular inconsistencies are a major source of failure for Internet services. To prevent inconsistencies, configuration stores often use strongly consistent coordination services. ACMS [75] is Akamai's distributed configuration store, which, similarly to Facebook's Zeus [76], is based on a single instance of a strongly consistent coordination protocol. Our design offers a scalable alternative where, assuming that the stored information is highly partitionable, updates rarely go through WAN and can execute with low latency and completely independently in the different partitions, while all reads (even of data stored remotely) remain local. We demonstrate that the amortized cost of sync messages is low for such read-heavy systems (in both ACMS and Zeus the reported rate of updates is only hundreds per hour).

4.6.2 Composition Methods

Consul [4], ZooFence [45] and Volery [32] are coordination services designed with the multi-data center deployment in mind. They provide linearizable updates and either linearizable or sequentially consistent reads. Generally, these systems follow the multiple coordination

services methodology (Figure 4.2c) – each coordination service is responsible for part of the data, and requests are forwarded to the appropriate coordination service (or to a local proxy). As explained in Section 4.2, when the forwarded operations are sequentially-consistent reads, this method does not preserve the single coordination service's semantics. We believe that, as in ZooKeeper, this issue can be rectified using our modular composition approach.

ZooFence [45] orchestrates multiple instances of ZooKeeper using a client-side library in addition to a routing layer consisting of replicated queues and executors. Intuitively, it manages local and cross-data center partitions using data replication. Any operation (including reads) accessing replicated data must go through ZooFence's routing layer. This prevents reads from executing locally, forfeiting a major benefit of replication. In contrast, ZooNet uses learners, (which natively exist in most coordination services in the form or proxies or observers), for data replication. This allows local reads, and does not require orchestration of multiple ZooKeeper instances as in ZooFence.

Volery [32] is an application that implements ZooKeeper's API, and which consists of partitions, each of which is an instance of a state machine replication algorithm. Unlike ZooKeeper, all of Volery operations are linearizable (i.e., including reads). In Volery, the different partitions must communicate among themselves in order to maintain consistency, unlike ZooNet's design in which the burden of maintaining consistency among ZooKeepers is placed only on clients. In addition, when compared to ZooKeeper, Volery shows degredated performance in case of a single partition, while ZooNet is identical to ZooKeeper if no remote operations are needed.

In distributed database systems, composing multiple partitions is usually done with protocols such as two-phase commit (e.g., as in [37]). In contrast, all coordination services we are familiar with are built on key-value stores, and expose simpler non-transactional updates and reads supporting non-ACID semantics.

Server-side solutions were also proposed for coordination services composition [14] but were never fully implemented due to their complexity, the intrusive changes they require from the underlying system, as well as the proposed relaxation of coordination service's semantics required to make them work. In this paper we show that composing such services does not require expensive server-side locking and commit protocols among partitions, but rather can be done using a simple modification of the client-side library and can guarantee the standard coordination service semantics.

Acknowledgements

We thank Arif Merchant, Mustafa Uysal, John Wilkes, and the anonymous reviewers for helpful comments and suggestions. We gratefully acknowledge Google for funding our

experiments on Google Cloud Platform. We thank Emulab for the opportunity to use their testbeds. Kfir Lev-Ari is supported in part by the Hasso-Plattner Institute (HPI) Research School. Research partially done while Kfir Lev-Ari was an intern with Yahoo, Haifa. Partially supported by the Israeli Ministry of Science.

Chapter 5

Paper: Composing ordered sequential consistency

Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer : "Composing ordered sequential consistency". Information Processing Letters, Volume 123, July 2017, Pages 47-50, ISSN 0020-0190.

In this paper we introduce Ordered Sequential Consistency, which generalize linearizability and sequential consistency, and prove a sufficient conditions for its composability.

Composing Ordered Sequential Consistency

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*

Abstract

We define *ordered sequential consistency* (OSC), a generic criterion for concurrent objects, which encompasses a range of criteria, from sequential consistency to linearizability. We show that OSC captures the typical behavior real-world coordination services, such as ZooKeeper. A straightforward composition of OSC objects is not necessarily OSC. To remedy this, we recently implemented a composition framework that injects dummy operations in specific scenarios. We prove that injecting such operations, which we call here *leading ordered operations*, enables correct OSC composition.

5.1 Introduction

In this work we define a generic correctness criterion named *Ordered Sequential Consistency* (OSC), which captures a range of criteria, from sequential consistency [53] to linearizability [49].

We use OSC to capture the semantics of coordination services such as ZooKeeper [50]. 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 [55]. 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 atomic writes and FIFO ordered operations for each client [50]. This definition is not accurate in two ways: (1) linearizability of updates has no meaning when no operation reads the written values; and (2) this definition allows read operations to read from a future write, which obviously does

not occur in any real-world service. A special case of OSC, which we call OSC(U), captures the actual guarantees of existing coordination services.

Although supporting OSC(U) semantics instead of atomicity of all operations enables fast local reads, this makes services *non-composable*: correct OSC(U) coordination services may fail to provide the same level of consistency when combined [58]. Intuitively, the problem arises because OSC(U), similarly to sequential consistency [53], allows sub-set of operations to occur "in the past", which can introduce cyclic dependencies.

In a companion systems paper [58] 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 ordered operations*. 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. Interestingly, Vitenberg and Friedman [80] 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 having leading ordered operations is not a local property.

5.2 Model and Notation

We use a standard shared memory execution model [49], 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 all 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 of x that updates it 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.

We refer to an operation that changes the object's value as an *update operation*. 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.

5.3 Ordered Sequential Consistency

Definition 5.3.1 (OSC(A))**.** A history σ is *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'), satisfying the following:

> OSC_1 (sequential specification): $\forall x \in$ 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 (A-real-time order): $\forall x \in$ X, for an operation $o \in A$ and an operation o' (not necessarily in A) s.t. $o, o' \in \sigma_x$, if $o' <_\sigma o$ then $o' <_\pi o$.

Such π is called a *serialization* of σ. An object is OSC(A) if all of its histories are OSC(A).

We assume that $\forall x \in$ X, $di_x(v_0) \in A$. Linearizability and sequential consistency are both special cases of OSC(A): (1) we get linearizability using A that consist of all of the objects' operations; and (2) we get sequential consistency with A that consists only of dummy initialization operations, which means that there is no operation that precedes an A-operation, i.e., OSC_3 is null, and we left with the sequential specification and process order of an object.

If A consists of the objects' update operations, denoted U, then OSC(U) captures the semantics of coordination services: (1) updates are globally ordered (by OSC_3); and (2) all operations see some prefix of that order (by OSC_3), while respecting each client process order (by OSC_2).

5.4 OSC(A) Composability via Leading A-Operations

In this section we show that a history σ of OSC(A) objects satisfies OSC(A), if σ has leading ordered A-operations. Generally, we prove the composition by ordering every A-operation o_A on object x, according to the first event $e \in \sigma$ s.t. $e \leq_\sigma r_{o_A}$ and $i_{o_A} <_{\pi_x} e$. Then, we extend that order to a total order on all operations, by placing every non-A-operation after the A-operation that precedes it in their object's serialization. Finally, we show that if σ has leading ordered A-operations, then the total order satisfies OSC(A).Intuitively, we can think of the leading A-operations as a barrier for the non-A-operations, that maintains the total order between objects.

Given a history σ of OSC(A) objects, and a set of serializations $\Pi = \{\pi_x\}_{x \in \mathrm{X}}$ of $\{\sigma_x\}_{x \in \mathrm{X}}$, we define a strict total order on all operations in Π. We refer to an operation $o \in A$ as an A-operation, and define the future set of an A-operation as follows:

Definition 5.4.1 (A-operation future set)**.** Given a history σ of OSC(A) objects, an object $x \in \sigma$, a serialization π_x of σ_x, and an A-operation $o_A \in \sigma_x$, the *future set of o_A in π_x* is $F_\sigma^{\pi_x}(o_A) \triangleq \{o \in \pi_x | o_A \leq_{\pi_x} o\}$.

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

Definition 5.4.2 (First response event)**.** Given a history σ of OSC(A) objects, an object $x \in \sigma$, a serialization π_x of σ_x, and an A-operation $o_A \in \pi_x$, the *first response event of o_A in π_x*, denoted $fr_\sigma^{\pi_x}(o_A)$, is the earliest response event in σ of an operation in $F_\sigma^{\pi_x}(o_A)$.

Note that it is possible that $fr_\sigma^{\pi_x}(o_A)$ is o_A's response event. We make two observations regarding first responses:

Observation 5.4.3. Given OSC(A) objects' σ, an object $x \in \sigma$, a serialization π_x of σ_x, and an A-operation $o_A \in \pi_x$, then $i_{o_A} <_\sigma fr_\sigma^{\pi_x}(o_A)$.

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

Observation 5.4.4. Let σ be OSC(A) objects' history, and let π_x be a serialization of σ_x for some x. For two A-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 $F_\sigma^{\pi_x}(o') \subset F_\sigma^{\pi_x}(o)$. By Definition 5.4.2, $fr_\sigma^{\pi_u}(o')$ is a response event of an operation $o_1 \in F_\sigma^{\pi_x}(o')$, and therefore $o_1 \in F_\sigma^{\pi_x}(o)$. 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 A-operations:

Definition 5.4.5 (A-Π-order)**.** Let σ be a history of OSC(A) objects. Let $\Pi = \{\pi_x\}_{x\in \mathrm{X}}$ be a set of serializations of $\{\sigma_x\}_{x\in \mathrm{X}}$. Let $x, y \in \mathrm{X}$, then for two A-operations $o_A \in \pi_x$ and $o'_A \in \pi_y$, we define their A-Π-order, denoted $<_{A\Pi}$, as follows: (<) If $x = y$, i.e., $o_A, o'_A \in \pi_x$, then $o_A <_{A\Pi} o'_A$ iff $o_A <_{\pi_x} o'_A$; otherwise, (fr) $x \neq y$, and $o_A <_{A\Pi} o'_A$ iff $fr_\sigma^{\pi_x}(o_A) <_\sigma fr_\sigma^{\pi_y}(o'_A)$.

Lemma 5.4.6. *For a history σ of OSC objects and a set of serializations $\Pi = \{\pi_x\}_{x\in \mathrm{X}}$ of $\{\sigma_x\}_{x\in \mathrm{X}}$, A-Π-order is a strict total order on A-operations in Π.*

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

Let o_A, o'_A, and o''_A be three A-operations s.t. $uo_1 <_{A\Pi} uo_2 <_{A\Pi} uo_3$; we need to prove that $uo_1 <_{A\Pi} uo_3$. We consider four cases according to the condition by which each of the pairs is ordered:

(<,<) If $\exists x \in \mathrm{X}\ o_A, o'_A, o''_A \in \pi_x$, then $o_A <_{\pi_x} o'_A <_{\pi_x} o''_A$ implies $o_A <_{\pi_x} o''_A$, and thus $o_A <_{A\Pi} o''_A$.

(<,fr) If $\exists x, y \in \mathrm{X}, x \neq y : o_A <_{\pi_x} o'_A$, $o''_A \in \pi_y$, and $fr_\sigma^{\pi_x}(o'_A) <_\sigma fr_\sigma^{\pi_y}(o''_A)$, by Observation 5.4.4, $fr_\sigma^{\pi_x}(o_A) \leq_\sigma fr_\sigma^{\pi_x}(o'_A)$, therefore $fr_\sigma^{\pi_x}(o_A) <_\sigma fr_\sigma^{\pi_y}(o''_A)$, and $o_A <_{A\Pi} o''_A$.

(fr,<) If $\exists x, y \in \mathrm{X}, x \neq y : o_A \in \pi_x$, $o'_A <_{\pi_y} o''_A$, and $fr_\sigma^{\pi_x}(o_A) <_\sigma fr_\sigma^{\pi_y}(o'_A)$, by Observation 5.4.4, $fr_\sigma^{\pi_y}(o'_A) \leq_\sigma fr_\sigma^{\pi_y}(o''_A)$. We get $fr_\sigma^{\pi_x}(o_A) <_\sigma fr_\sigma^{\pi_y}(o''_A)$, therefore $o_A <_{A\Pi} o''_A$.

(fr,fr) If $\exists x, y, z \in \mathrm{X}, x \neq y, y \neq z : o_A \in \pi_x$, $o'_A \in \pi_y$, and $o''_A \in \pi_z$, this means that $fr_\sigma^{\pi_x}(o_A) <_\sigma fr_\sigma^{\pi_y}(o'_A)$ and $fr_\sigma^{\pi_y}(o'_A) <_\sigma fr_\sigma^{\pi_z}(o''_A)$. By transitivity of $<_\sigma$, $fr_\sigma^{\pi_x}(o_A) <_\sigma fr_\sigma^{\pi_z}(o''_A)$. If $z \neq x$, then $o_A <_{A\Pi} o''_A$. If $z = x$, by the contrapositive of Observation 5.4.4, $o_A <_{\pi_x} o''_A$, and $o_A <_{A\Pi} o''_A$. ■

We extend $<_{A\Pi}$ to a weak total order in the usual way: $o_1 \leq_{A\Pi} o_2$ if $o_1 <_{A\Pi} o_2$ or $o_1 = o_2$. For a history σ, a serialization π_x of σ_x, and an operation o in π_x, the *last A-operation before o in π_x*, denoted $\mathrm{l}A_{\pi_x}(o)$, is the latest A-operation in the prefix of π_x that ends with o. Note that if o is an A-operation then $\mathrm{l}A_{\pi_x}(o) = o$; and that since every history starts with a dummy initialization, every operation that is not in A is preceded by at least one A-operation and so $\mathrm{l}A_{\pi_x}(o)$ is well-defined. We use last A-operations to extend the A-Π-order to a strict total order on all operations in Π.

Definition 5.4.7 (Π-order)**.** Let σ be a history of OSC(A) objects. Let $\Pi = \{\pi_x\}_{x\in \mathrm{X}}$ be a set of serializations of $\{\sigma_x\}_{x\in \mathrm{X}}$, and let x and y be objects in X. For two operations $o_1 \in \pi_x$, and $o_2 \in \pi_y$, we define Π-order, denoted $<_\Pi$, as follows:

$(\mathrm{l}A_{\pi_x}(o_1) \neq \mathrm{l}A_{\pi_y}(o_2))$ if the last A-operation before o_1 and o_2 are different, then $o_1 <_\Pi o_2$ iff $\mathrm{l}A_{\pi_x}(o_1) <_{A\Pi} \mathrm{l}A_{\pi_y}(o_2)$;

$(\mathrm{l}A_{\pi_x}(o_1) = \mathrm{l}A_{\pi_y}(o_2))$ 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 5.4.8. Let σ be a history of OSC(A) objects, and $\pi_x \in \Pi$ a serialization of σ_x for some object $x \in \mathrm{X}$. 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 $\mathrm{l}A_{\pi_x}(o_1) \leq_{\pi_x} \mathrm{l}A_{\pi_x}(o_2)$. If $\mathrm{l}A_{\pi_x}(o_1) = \mathrm{l}A_{\pi_x}(o_2)$ then by Definition 5.4.7, $o_1 <_\Pi o_2$. Otherwise, by Definition 5.4.5, $\mathrm{l}A_{\pi_x}(o_1) <_{A\Pi} \mathrm{l}A_{\pi_x}(o_2)$ and by Definition 5.4.7, $o_1 <_\Pi o_2$. ■

Lemma 5.4.9. *Let σ be a history of OSC(A) objects, and $\Pi = \{\pi_x\}_{x\in\mathrm{X}}$ be a set of serializations of $\{\sigma_x\}_{x\in\mathrm{X}}$, then Π-order is a strict total order on all operations in Π.*

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 5.4.7, $o_i <_\Pi o_j$ implies $\mathrm{l}A_{\pi_i}(o_i) \leq_{A\Pi} \mathrm{l}A_{\pi_j}(o_j)$. By transitivity of $\leq_{A\Pi}$ (Lemma 5.4.6), we get from $\mathrm{l}A_{\pi_x}(o_1) \leq_{A\Pi} \mathrm{l}A_{\pi_y}(o_2) \leq_{A\Pi} \mathrm{l}A_{\pi_z}(o_3)$ that $\mathrm{l}A_{\pi_x}(o_1) \leq_{A\Pi} \mathrm{l}A_{\pi_z}(o_3)$.

If $\mathrm{l}A_{\pi_x}(o_1) <_{A\Pi} \mathrm{l}A_{\pi_z}(o_3)$ then by Definition 5.4.7 $o_1 <_\Pi o_3$. If $\mathrm{l}A_{\pi_x}(o_1) = \mathrm{l}A_{\pi_z}(o_3)$, then by $\mathrm{l}A_{\pi_x}(o_1) \leq_{A\Pi} \mathrm{l}A_{\pi_y}(o_2) \leq_{A\Pi} \mathrm{l}A_{\pi_z}(o_3)$ we get $\mathrm{l}A_{\pi_x}(o_1) = \mathrm{l}A_{\pi_y}(o_2) = \mathrm{l}A_{\pi_z}(o_3)$, and $x = y = z$. Therefore by $o_1 <_\Pi o_2 <_\Pi o_3$ and Definition 5.4.7, $o_1 <_{\pi_x} o_2 <_{\pi_x} o_3$, and thus by Definition 5.4.7 $o_1 <_\Pi o_3$. ■

Note that Π-order is always defined for compositions of OSC objects. Since it generalizes all the serializations π_x (Observation 5.4.8), it preserves OSC_1 and OSC_3. Nevertheless, OSC_2 is not guaranteed.

To support OSC(A) composition we extend each object with a *sync* operation, which does not change the object's state and does not return any value, but belongs to A. For example, to compose OSC($\{di_x(v_0)|\forall x \in \mathrm{X}\}$) objects, we extend each of them to be an OSC($\{\mathrm{sync}\} \cup \{di_x(v_0)|\forall x \in \mathrm{X}\}$) object and then compose them via adding sync operations.

We say that in a history σ there are *leading ordered operations* if for every operation $o \notin A$ by a process p in σ, the last operation of p before o is on the same object. This also means that between every two operations $o \notin A$ and $o' \notin A$ of different objects by the same process in σ, there is an operation $o_A \in A$ to the second object. We next prove that adding leading ordered operations allows for correct OSC composition.

Theorem 5.1. *If a history σ of OSC(A) objects has leading ordered operations, then σ is OSC(A).*

Proof. Let $\Pi = \{\pi_x\}_{x \in \mathrm{X}}$ be a set of serializations of $\{\sigma_x\}_{x \in \mathrm{X}}$, and let π be the sequential permutation of σ defined by $<_\Pi$. We now prove that π satisfies OSC(A). **OSC**$_1$ and **OSC**$_3$ follow immediately from Observation 5.4.8.

We prove **OSC**$_2$. Let o_1 and o_2 be two operations in Π 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 5.4.8, $o_1 <_\Pi o_2$, as needed. Otherwise, $\exists x, y \in \mathrm{X}, x \neq y : o_1 \in \pi_x, o_2 \in \pi_y$, and o_1 immediately precedes o_2 in $\sigma|p$. By leading ordered operations, since o_1 and o_2 are not on the same object, o_2 is a A-operation and hence $\mathrm{l}A_{\pi_y}(o_2) = o_2$.

By definition, $fr_\sigma^{\pi_x}(\mathrm{l}A_{\pi_x}(o_1)) \leq_\sigma r_{o_1}$. Since $r_{o_1} <_\sigma i_{o_2}$, and by Observation 5.4.3, $i_{o_2} <_\sigma fr_\sigma^{\pi_y}(o_2)$, we get that $fr_\sigma^{\pi_x}(\mathrm{l}A_{\pi_x}(o_1)) <_\sigma fr_\sigma^{\pi_y}(o_2)$. By Definition 5.4.5, $\mathrm{l}A_{\pi_x}(o_1) <_{A\Pi} o_2$, and by Definition 5.4.7, $o_1 <_\Pi o_2$.

Thus, every two consecutive operations $o^i, o^{i+1} \in \Pi$ that are in $\sigma|p$ satisfy $o^i <_\Pi o^{i+1}$. By Lemma 5.4.9, $<_\Pi$ is a strict total order on all operations, and therefore by transitivity, we get $o_1 <_\Pi o_2$. ■

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.

Chapter 6

Discussion

In this thesis we presented the following results:

- We defined ordered sequential consistency (OSC), a generic criterion for concurrent objects. We show that OSC encompasses a range of criteria, from sequential consistency to linearizability, and captures the typical behavior of real-world coordination services, such as ZooKeeper. In Section 6.4 we further discuss the results and implications of our "Composing Ordered Sequential Consistency" paper.

- We presented a system design for modular composition of services that addresses the performance-correctness trade-off. We implemented ZooNet, a prototype of this concept over ZooKeeper. ZooNet allows users to compose multiple instances of the service in a consistent fashion, facilitating applications that execute in multiple regions. In Section 6.3 we further discuss the results and implications of our "Modular Composition of Coordination Services" paper.

- We presented a comprehensive methodology for proving linearizability and related criteria of concurrent data structures. We exemplified our methodology by using it to give a road-map for proving linearizability of the popular Lazy List implementation of the concurrent set abstraction. In Section 6.2 we further discuss the results and implications of our "A Constructive Approach for Proving Data Structures' Linearizability" paper, and in Section 6.1 of our "On Correctness of Data Structures under Reads-Write Concurrency" paper.

The papers we discuss in Section 6.1 and Section 6.2 handle linearizability, which is one of the most common correctness criteria in use nowadays for shared memory objects.

One of the main reasons linearizability is preferred over other correctness criteria is that linearizability is composable. The paper discussed in Section 6.3 presents our solution for the composability problem of coordination services, that are essential components of distributed systems nowadays. In distributed systems, linearizable operations suffer from a performance penalty, and therefore relax correctness criteria are in use. The paper we discuss in Section 6.4 generalizes linearizability along with the coordination services' semantics into a single definition, and states a generic and sufficient condition for composability.

6.1 On Correctness of Data Structures under Reads-Write Concurrency

In this paper we introduced a new framework for reasoning about correctness of data structures in concurrent executions, which facilitates the process of verifiable parallelization of legacy code. Our methodology consists of identifying base conditions in sequential code, and ensuring regularity base points for these conditions under concurrency. This yields two essential correctness aspects in concurrent executions – the internal behavior of the concurrent code, which we call validity, and the external behavior, in this case regularity, which we have generalized here for data structures. Linearizability is guaranteed if the implementation further satisfies linearizability base point consistency.

We believe that this paper is only the tip of the iceberg, and that many interesting connections can be made using the observations we have presented. For a start, an interesting direction to pursue is to use our methodology for proving the correctness of more complex data structures than the linked lists in our examples.

Currently, using our methodology involves manually identifying base conditions. It would be interesting to create tools for suggesting a base condition for each local state. One possible approach is to use a dynamic tool that identifies likely program invariants, as in [42], and suggests them as base conditions. Alternatively, a static analysis tool can suggest base conditions, for example by iteratively accumulating read shared variables and omitting ones that are no longer used by the following code (i.e., shared variables whose values are no longer reflected in the local state).

Another interesting direction for future work might be to define a synchronization mechanism that uses the base conditions in a way that is both general purpose and fine-

grained. A mechanism of this type will use default conservative base conditions, such as verifying consistency of the entire read-set for every local state, or two-phase locking of accessed shared variables. In addition, the mechanism will allow users to manually define or suggest finer-grained base conditions. This can be used to improve performance and concurrency, by validating the specified base condition instead of the entire read-set, or by releasing locks when the base condition no longer refers to the value read from them.

From a broader perspective, we showed how correctness can be derived from identifying inner relations in a sequential code, (in our case, base conditions), and maintaining those relations in concurrent executions (via base points). It may be possible to use similar observations in other models and contexts, for example, looking at inner relations in synchronous protocols, in order to derive conditions that ensure their correctness in asynchronous executions.

And last but not least, the definitions of internal behavior correctness can be extended to include weaker conditions than validity, (which is quiet conservative). These weaker conditions will handle local states in concurrent executions that are un-reachable via sequential executions but still satisfy the inner correctness of the code.

6.2 A Constructive Approach for Proving Data Structures' Linearizability

In this paper we introduced a constructive methodology for proving correctness of concurrent data structures and exemplified it with a popular data structure. Our methodology outlines a road-map for proving correctness. While we have exemplified its use for writing semi-formal proofs, we believe it can be used at any level of formalism, from informal correctness arguments to formal verification. In particular, our framework has the potential to simplify the proof structure employed by existing formal methodologies for proving linearizability [35, 36, 39, 41, 44, 67, 79], thus making them more accessible to practitioners.

Currently, using our methodology involves manually identifying base conditions, commuting steps, and base point preserving steps. It would be interesting to create tools for suggesting a base condition for each local state, and identifying the interesting steps in update operations using either static or dynamic analysis.

6.3 Modular Composition of Coordination Services

Coordination services provide consistent and highly available functionality to applications, relieving them of implementing common (but subtle) distributed algorithms on their own. Yet today, when applications are deployed in multiple data centers, system architects are forced to choose between consistency and performance. In this paper we showed that this does not have to be the case. Our modular composition approach maintains the performance and simplicity of deploying independent coordination services in each data center, and yet does not forfeit consistency.

We demonstrated that the simplicity of our technique makes it easy to use with existing coordination services, such as ZooKeeper – it does not require changes to the underlying system, and existing clients may continue to work with an individual coordination service without any changes (even if our client library is used, such applications will not incur any overhead). Moreover, the cost for applications requiring consistent multi-data center coordination is low for workloads that exhibit high spatial or temporal locality.

In this work we have focused on the advantages of our composition design in wide-area deployments. It is possible to leverage the same design for deployments within the data center boundaries that currently suffer from lack of sharing among coordination services. Indeed, a typical data center today runs a multitude of coordination service backend services. For example, it may include: Apache Kafka message queues [1], backed by ZooKeeper and used in several applications; Swarm [7], a Docker [65] clustering system running an etcd backend; Apache Solr search platform [2] with an embedded ZooKeeper instance; and Apache Storm clusters [77], each using a dedicated ZooKeeper instance. Thus, installations end up running many independent coordination service instances, which need to be independently provisioned and maintained. This has a number of drawbacks: (1) it does not support cross-application sharing; (2) it is resource-wasteful, and (3) it complicates system administration. Our modular composition approach can potentially remedy these short comings.

Our composition algorithm supports individual query and update operations. It can natively support transactions (e.g., ZooKeeper's *multi* operation) involving data in single service instance. An interesting future direction could be to support transactions involving multiple service instances. This is especially challenging in the face of possible client and service failures, if all cross-service coordination is to remain at the client side.

6.4 Composing Ordered Sequential Consistency

Coordination services are broadly deployed nowadays in backends of large-scale distributed systems. In this paper we defined OSC(A), which encompasses a range of criteria, from sequential consistency to linearizability, and captures the typical behavior of coordination services, such as ZooKeeper.

By itself, OSC(A) is non-composable. 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 ordered operations. Composability of coordination services enables low-latency local updates, while having global consistency among services.

Bibliography

[1] Apache Kafka – A high-throughput distributed messaging system., . URL http://kafka.apache.org. [Online; accessed 1-Jan-2016].

[2] Apache Solr – a standalone enterprise search server with a REST-like API., . URL http://lucene.apache.org/solr/. [Online; accessed 1-Jan-2016].

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

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

[5] Consul HTTP API, . URL https://www.consul.io/docs/agent/http.html. [Online; accessed 28-Jan-2016].

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

[7] Swarm: a Docker-native clustering system. URL https://github.com/docker/swarm. [Online; accessed 1-Jan-2016].

[8] Daniel Abadi: Problems with CAP, and Yahoo's little known NoSQL system. URL http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html. [Online; accessed 28-Jan-2016].

[9] Access Control in Google Cloud Storage. URL https://cloud.google.com/storage/docs/access-control. [Online; accessed 28-Jan-2016].

[10] Question about multi-datacenter key-value consistency (Consul). URL https://goo.gl/XMWCcH. [Online; accessed 28-Jan-2016].

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

[12] Google Compute Engine – Scalable, High-Performance Virtual Machines. URL https://cloud.google.com/compute/. [Online; accessed 1-Jan-2016].

[13] Observers: Making ZooKeeper Scale Even Further. URL https://blog.cloudera.com/blog/2009/12/observers-making-zookeeper-scale-even-further/. [Online; accessed 1-Jan-2016].

[14] Proposal: mounting a remote ZooKeeper. URL https://wiki.apache.org/hadoop/ZooKeeper/MountRemoteZookeeper. [Online; accessed 28-Jan-2016].

[15] ZooKeeper's Jira - Major throughput improvement with mixed workloads. URL https://issues.apache.org/jira/browse/ZOOKEEPER-2024. [Online; accessed 16-May-2016].

[16] Igor Serebryany: SmartStack vs. Consul. URL http://igor.moomers.org/smartstack-vs-consul/. [Online; accessed 28-Jan-2016].

[17] Solr Cross Data Center Replication. URL http://yonik.com/solr-cross-data-center-replication/. [Online; accessed 28-Jan-2016].

[18] Vitess deployment: global vs local. URL http://vitess.io/doc/TopologyService/#global-vs-local. [Online; accessed 28-Jan-2016].

[19] Question about number of nodes spread across datacenters (ZooKeeper)., . URL https://goo.gl/oPC2Yf. [Online; accessed 28-Jan-2016].

[20] Question about cross-datacenter setup (ZooKeeper)., . URL http://goo.gl/0sD0MZ. [Online; accessed 28-Jan-2016].

[21] Has anyone deployed a ZooKeeper ensemble across data centers? URL https://www.quora.com/Has\anyone\deployed\a\ZooKeeper\ensemble\across\data\centers. [Online; accessed 1-Jan-2016].

[22] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 739–753, 2016. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya`.

[23] Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic software lock-elision. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 297–311, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33650-8. doi: 10.1007/978-3-642-33651-5_21.

[24] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2944-6. doi: 10.1145/2611462.2611471.

[25] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy-update techniques for system v ipc in the linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309. USENIX, 2003. ISBN 1-931971-11-0.

[26] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004. ISBN 0471453242.

[27] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011. URL `http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf`.

[28] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-65-6.

[29] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785.

[30] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10715-5.

[31] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[32] Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. Scalable state-machine replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 331–342, 2014. doi: 10.1109/DSN.2014.41. URL `http://dx.doi.org/10.1109/DSN.2014.41`.

[33] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *PODC*, pages 13–22, 2013.

[34] 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. ISBN 1-931971-47-1. URL `http://dl.acm.org/citation.cfm?id=1298455.1298487`.

[35] Gregory Chockler, Nancy Lynch, Sayan Mitra, and Joshua Tauber. Proving atomicity: An assertional approach. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 152–168, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29163-6, 978-3-540-29163-3. doi: 10.1007/11561927_13.

[36] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set. In *In 18th CAV*, pages 475–488. Springer, 2006.

[37] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey

Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387905`.

[38] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[39] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Verifying linearisability with potential linearisation points. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 323–337, 2011. doi: 10.1007/978-3-642-21437-0_25.

[40] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[41] Brijesh Dongol and John Derrick. Proving linearisability via coarse-grained abstraction. *CoRR*, abs/1212.5116, 2012.

[42] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302467.

[43] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233.

[44] Rachid Guerraoui and Marko Vukolic. A scalable and oblivious atomicity assertion. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2008. ISBN 978-3-540-85360-2. URL `http://dblp.uni-trier.de/db/conf/concur/concur2008.html#GuerraouiV08`.

[45] Raluca Halalai, Pierre Sutra, Etienne Riviere, and Pascal Felber. Zoofence: Principled service partitioning and application to the zookeeper coordination service. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014*, pages 67–78, 2014. doi: 10.1109/SRDS.2014.41. URL `http://dx.doi.org/10.1109/SRDS.2014.41`.

[46] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36321-1, 978-3-540-36321-7. doi: 10.1007/11795490_3.

[47] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. doi: 10.1145/1810479.1810540.

[48] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. ISSN 0163-5964. doi: 10.1145/173682.165164.

[49] 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. ISSN 0164-0925. doi: 10.1145/78969.78972.

[50] 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`.

[51] 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. ISBN 978-1-4244-9232-9. doi: 10.1109/DSN.2011.5958223. URL `http://dx.doi.org/10.1109/DSN.2011.5958223`.

[52] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387903`.

[53] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439.

[54] Leslie Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

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

[56] Kfir Lev-Ari, Gregory Chockler, and Idit Keidar. On correctness of data structures under reads-write concurrency. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 273–287, 2014. doi: 10.1007/978-3-662-45174-8_19.

[57] Kfir Lev-Ari, Gregory Chockler, and Idit Keidar. A constructive approach for proving data structures' linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 356–370, 2015. doi: 10.1007/978-3-662-48653-5_24. URL `https://doi.org/10.1007/978-3-662-48653-5_24`.

[58] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, pages 251–264, 2016. URL `https://www.usenix.org/conference/atc16/technical-sessions/presentation/lev-ari`.

[59] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Composing ordered sequential consistency. *Inf. Process. Lett.*, 123:47–50, 2017. doi: 10.1016/j.ipl.2017.03.004. URL `https://doi.org/10.1016/j.ipl.2017.03.004`.

[60] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384, 2008. URL `http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf`.

[61] Parisa Jalili Marandi, Carlos Eduardo Bezerra, and Fernando Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 368–377, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5169-7. doi: 10.1109/ICDCS.2014.45. URL `http://dx.doi.org/10.1109/ICDCS.2014.45`.

[62] Paul E. McKenney. Selecting locking primitives for parallel programming. *Commun. ACM*, 39(10):75–82, Oct. 1996. ISSN 0001-0782. doi: 10.1145/236156.236174.

[63] Paul E. McKenney. RCU part 3: the RCU API. January 2008.

[64] Paul E. McKenney and John D. Slingwine. Read-copy update: using execution history to solve concurrency problems, parallel and distributed computing and systems, 1998.

[65] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[66] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14, 47–30, 2007. Chapman and Hall/CRC Press.

[67] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 85–94, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835722.

[68] 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. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643666.

[69] David L. Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*, 2003. URL http://www.usenix.org/events/usits03/tech/oppenheimer.html.

[70] Behrokh Samadi. B-trees in a system with multiple users. *Inf. Process. Lett.*, 5(4): 107–112, 1976.

[71] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM. ISBN 1-58113-994-2. doi: 10.1145/1073814.1073861.

[72] Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.*, 40(1):28–62, 2011.

[73] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. VLDB Endow.*, 8(12):1490–1501, Aug. 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824047. URL http://dx.doi.org/10.14778/2824032.2824047.

[74] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224987.

[75] Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. ACMS: the akamai configuration management system. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings.*, 2005. URL http://www.usenix.org/events/nsdi05/tech/sherman.html.

[76] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 328–343, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815401. URL `http://doi.acm.org/10.1145/2815400.2815401`.

[77] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2595641. URL `http://doi.acm.org/10.1145/2588555.2595641`.

[78] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. A safety proof of a lazy concurrent list-based set implementation. Technical Report UCAM-CL-TR-659, University of Cambridge, Computer Laboratory, jan 2006.

[79] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 129–136, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122992.

[80] Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *DISC'03*.

[81] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[82] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium*

on Operating Systems Principles, SOSP '11, pages 159–172, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043572. URL http://doi.acm.org/10.1145/2043556.2043572.

למופעים נפרדים של אותו שירות תיאום באופן עקבי, תוך רתימת המקומיות עבור שיפור הביצועים. באופן כללי, משתמש אשר ניגש למידע מקומי בלבד לא נענש מבחינת ביצועים בהשוואה לעבודה עם מופע יחיד של אותו שירות תיאום. כמו כן, משתמשים אשר ניגשים למידע במספר מופעים של שירותי תיאום, מציגים שיפור בביצועים של פי 7 בהשוואה לפתרונות עקביים הזמינים כיום. המימוש שלנו מאפשר ללקוחות לעבוד עם מספר מופעים של שרתי תיאום ללא צורך בשינוי כל שהוא בצד השרת של אותם שירותים, על ידי כך שהפעולות אשר הלקוח מבצע מתורגמות לפעולות אותן שרת התיאום מכיר ממשתמשים רגילים של המערכת, כלומר משתמשים שלא ניגשים למספר שירותי תיאום במקביל.

אנו מציגים מתודולוגיה מקיפה להוכחת נכונותם של מבני נתונים מקבילים. אנו מדגימים את המתודולוגיה שלנו בכך שאנו מספקים מפת דרכים להוכחת אטומיות של אובייקטים מקביליים. אטומיות, שהינה מקרה מיוחד של עקביות רציפה וסדורה, היא אחת מקריטריוני הנכונות הנפוצים ביותר לאובייקטים משותפים. אנו מדגימים יישום של המתודולוגיה שלנו להוכחת אטומיות של רשימה מקושרת המעודכנת באופן עצל, שהינה מבנה נתונים מקבילי פופולרי עבור השוואה בין טכניקות להוכחת נכונות. הוכחת הנכונות שאנו מציגים מבוססת על משפט המפתח שלנו, שמאגד תנאים מספיקים עבור אטומיות. בניגוד לעבודות קודמות, התנאים שלנו נגזרים ישירות מהמאפיינים של מבנה הנתונים בריצות סדרתיות, ללא צורך במציאת נקודות אטומיות עבור פעולות מבנה הנתונים.

תקציר

בעבודת מחקר זו אנו מגדירים עקביות רציפה וסדורה, קריטריון נכונות גנרי לאובייקטים מקבילים. אנו מראים כי עקביות רציפה וסדורה מכלילה טווח רחב של קריטריונים מוכרים, החל מעקביות רציפה ועד לינאריות (אטומיות), כמו גם לוכדת התנהגות אופיינית של שירותי תיאום שנמצאים בשימוש נרחב כיום. קומפוזיציה פשוטה של אובייקטים עקביים באופן רציף וסדור אינה בהכרח עקבית באופן רציף וסדור, למשל, ידוע שהרכבה של אובייקטים עקביים ורציפים אינה עקבית ורציפה (אי ההרכבה הוכחה בעבר ביחס לתכונות מקומיות של אובייקטים).

לשם הרכבה של אובייקטים עקביים רציפים וסדורים, אנו מגדירים תכונה גלובלית שאנו מכנים פעולות סדורות מובילות, ומוכיחים כי תכונה זאת מאפשרת הרכבה נכונה של אוביקטים עקביים רציפים וסדורים. הרעיון הוא להשתמש בתת קבוצה של פעולות להן קיים סדר מלא (התואם לסידורן על פי ציר הזמן) כדי לתחום את שאר הפעולות להן לא קיים סדר מלא שכזה. בפועל התכונה מגדירה שכל רצף פעולות שמבצע תהליך על אובייקט צריך להתחיל בפעולה מאותה תת קבוצה לה קיים אותו סידור מלא.

השלכה ישירה של העקביות הרציפה והסדורה יחד עם תכונת ההרכבה הגלובלית שהגדרנו, היא היכולת לחבר שירותי תיאום תוך שמירה על נכונותם. שירותים כאלה נהפכו בשנים האחרונות לאבן בניין הכרחית עבור יישומים מבוזרים לשם ביצוע תיאום עקבי, אמין ומהיר. כאשר יישומים מבוצעים באזורים גיאוגרפיים מרובים, אופן פריסת שירות התיאום מכריע בין בחירה בביצועים, (המושגים באמצעות שירותים עצמאיים באזורים נפרדים), לבין בחירה בעקביות. הגישות השונות שהיו בשימוש לפני העבודה שלנו הראו ביצועים ירודים עבור כתיבות, או אי־שמירה על עקביות המערכת.

אנו מציגים תכן מערכת להרכבה מודולרית של שירותי תיאום שמתייחס לשקלול התמורות הנ"ל. באופן כללי, התכן הוא תוספת של שכבת עיבוד בצד הלקוח. שכבה זאת עוקבת אחרי הפעולות שהלקוח מבצע, ובמידה והלקוח פונה להשתמש בשירות תיאום השונה מזה אליו פנה בפעולתו האחרונה, אז השכבה מוסיפה פעולת סנכרון בין הלקוח לשירות התיאום החדש. פעולות הסנכרון מסודרות באופן מלא ביחס לסידורן על פי ציר הזמן, ומכאן שמתקיימת תכונת הפעולות הסדורות המובילות באופן גלובלי, כלומר ההרכבה שומרת על עקביות. בנוסף, אנו מציגים מימוש לתכן הנ"ל, המאפשר למשתמשים להתחבר

המחקר בוצע בהנחייתה של פרופסור עדית קידר, בפקולטה להנדסת חשמל על שם אנדרו וארנה ויטרבי.

התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי־עת במהלך תקופת מחקר הדוקטורט של המחבר. גרסאותיהם העדכניות ביותר הינן אלו הנמצאות בחיבור זה.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

עקביות רציפה וסדורה: הרכבה ונכונות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

דוקטור לפילוסופיה

כפיר לב־ארי

הוגש לסנט הטכניון — מכון טכנולוגי לישראל

תמוז התשע"ז חיפה יולי 2017

עקביות רציפה וסדורה: הרכבה ונכונות

כפיר לב־ארי