ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service

Raluca Halalai, Pierre Sutra, Étienne Rivière, Pascal Felber
University of Neuchâtel, Switzerland
first.last@unine.ch

***Abstract*—Cloud computing infrastructures leverage fault-tolerant and geographically distributed services in order to meet the requirements of modern applications. Each service deals with a large number of clients that compete for the resources it offers. When the load increases, the service needs to scale. In this paper, we investigate a scalability solution which consists in partitioning the service state. We formulate specific conditions under which a service is partitionable. Then, we present a general algorithm to build a dependable and consistent partitioned service. To assess the practicability of our approach, we implement and evaluate the ZooFence coordination service. ZooFence orchestrates several instances of ZooKeeper and presents the exact same API and semantics to its clients. It automatically splits the coordination service state among ZooKeeper instances while being transparent to the application. By reducing the convoy effect on operations and leveraging the workload locality, our approach allows proposing a coordination service with a greater scalability than with a single ZooKeeper instance. The evaluation of ZooFence assesses this claim for two benchmarks, a synthetic service of concurrent queues and the BookKeeper distributed logging engine.**

I. Introduction

Distributed services form the basic building blocks of modern computer architectures. A large number of clients access these services, and when a client performs a command on a service, it usually expects the service to be responsive and consistent. The seminal state machine replication (SMR) approach offers both guarantees. By replicating the service on multiple servers, the commands are wait-free despite failures, and by executing them in the same order at all replicas, they are linearizable. However, it is well-known that this last strategy has a performance cost: because SMR serializes all commands, it does not leverage the intrinsic parallelism of the workload.

To overcome the above problems, several directions have been investigated. First, operations that do not change the service state can be executed at a single replica. This approach implies dropping linearizability for sequential consistency, but such a limitation is unavoidable in a partially-asynchronous system [1]. Second, SMR can leverage the commutativity of updates to improve response time. This strategy exhibits a performance improvement of at most 33% in comparison to the baseline [2]. Third, one can partition the state of the service and distribute the partitions between replicas [3]. When the workload is fully parallel, the scale-out of the partitioning approach is optimal. Hence, we consider this approach as the most promising direction.

To illustrate in practice the benefits of partitioning, let us consider Figure 1. In this figure, we compare a partitioned queue service versus a non-partitioned one. In more details, we execute a hypothetical consumer/producer workload where (i) each consumer pulls resources from some defined queue, and (ii) each producer pushes with 80% chance a resource to a random queue, and with 20% to all the queues. In the top-curves, we partition the queues, each being implemented with an instance of the Apache ZooKeeper coordination service [4]. In the non-partitioned case (bottom-curve), all queues employ the same ZooKeeper. The total computational power remains the same for the partitioned and non-partitioned implementations. For 32 clients, the partitioned approach outperforms the non-partitioned one by a factor of 1.6.

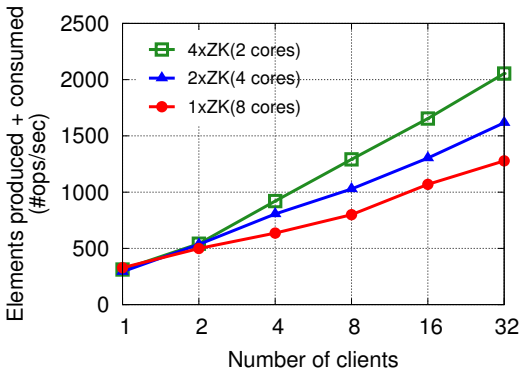


Fig. 1. Partitioned versus non-partitioned queue service.

Despite its obvious interest, to the best of our knowledge, few research efforts have been devoted to a principled study of service partitioning. In this paper, we try to bridge this gap. Our first contribution is to show that the partitioning theorem of Marandi et al. [3] omits some cases. We extend it and state a more general result under which it is possible to partition a shared service. Our second contribution consists in a general algorithm to partition a service. Our third contribution is ZooFence, a system that partitions the ZooKeeper coordination service following the previously introduced algorithm. ZooFence orchestrates multiple vanilla ZooKeepers, delegating portions of its state to each of them, and forwarding the operations that act on their respective partitions. In our last contribution, we evaluate ZooFence with two benchmarks, a synthetic concurrent queues service in a geo-distributed setting and the BookKeeper logging service. This evaluation shows that ZooFence improves the performance of the coordination

service compared to a single ZooKeeper, while offering at core the same guarantees.

The remainder of this paper is organized as follows. We formulate our results on service partitioning in Section II. Section III presents our general partitioning algorithm. We give an overview of ZooFence and the internals of its implementation in Section IV. We present a detailed evaluation of ZooFence in Section V. Section VI surveys related work. We conclude in Section VII. For readability purposes, we defer all our proofs to the appendix.

II. Consistent Service Partitioning

In what follows, we define the elements of our system model and the notion of partition. Further, we present two results that characterize if the partitioning of a service is consistent. These results form the guidelines of our approach to split a shared service into multiple parts in order to leverage the parallelism of its operations.

A. System Model

A service is specified by some serial data type. The serial data type defines the possible states of the service, the operations (or *commands*) to access it, as well as the response values from these commands. Formally, a serial data type is an automaton $S = (States, s^0, Cmd, Values, \tau)$ where $States$ is the set of states of S, $s^0 \in States$ its initial state, Cmd the commands of S, $Values$ the response values and $\tau : States \times Cmd \to States \times Values$ defines the transition relation. A command c is *total* if $States \times \{c\}$ is in the domain of τ. Command c is *deterministic* if the restriction of τ to $States \times \{c\}$ is a function. Hereafter, we assume that all commands are total and deterministic. We use $.st$ and $.val$ selectors to respectively extract the state and the response value components of a command, i.e., given a state s and a command c, $\tau(s,c) = (\tau(s,c).st, \tau(s,c).val)$. Function τ^+ is defined by repeated application of τ, i.e., given a sequence of commands $\sigma = \langle c_1, \ldots, c_{n\geq 1}\rangle$ and a state s:

$$\tau^+(s,\sigma) \triangleq \begin{cases} \tau(s,c_1) & \text{if } n = 1, \\ \tau^+(\tau(s,c_1).st, \langle c_2, \ldots, c_n\rangle) & \text{otherwise.} \end{cases}$$

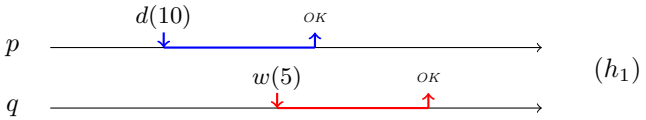
Two commands c and d *commute*, written $c \asymp d$, if in every state s we have:

$$c \asymp d \triangleq \begin{cases} \tau^+(s,\langle c,d\rangle).st = \tau^+(s,\langle d,c\rangle).st \\ \tau^+(s,\langle d,c\rangle).val = \tau^+(s,c).val \\ \tau^+(s,\langle c,d\rangle).val = \tau^+(s,d).val \end{cases}$$

For any two commands c and d, we write $c = d$ when in every state s, $\tau(s,c) = \tau(s,d)$. By extension, for some command c and some sequence $\sigma = \langle c_1, \ldots, c_{n\geq 2}\rangle$, we write $c = \sigma$ when $\tau^+(s,\langle c_1, \ldots, c_n\rangle) = \tau(s,c)$.

To illustrate the above notations, let us consider a bank account equipped with the usual withdraw and deposit operations. We define $States$ as $\mathbb{N}$, with $s_0 = 0$. A deposit operation $d(10)$ brings s to $s + 10$ and returns OK. In case the bank prohibits overdrafts, a withdraw operation $w(x)$ returns NOK if $s < x$; otherwise it brings s to $s - x$.

a) History: We consider a global time model and some bounded set of client processes that may fail-stop by crashing. A history is a sequence of invocations and responses of commands by the clients on one or more services. When command c precedes d in history h, we write $c <_h d$. We use timelines to illustrate histories. For instance the timeline below depicts the interleaving of commands $d(10)$ and $w(5)$, executed by respectively clients p and q in some history h_1.



Following Herlihy and Wing [5], histories have various properties according to the way invocations and responses interleave. For the sake of completeness, we recall these properties in what follows. A history h is *complete* if every invocation has a matching response. A *sequential* history h is a non-interleaved sequence of invocations and matching responses, possibly terminated by a non-returning invocation. When a history h is not sequential, we say that it is *concurrent*. A history h is *well-formed* if (i) $h|p$ is sequential for every client process p, (ii) for every command c, c is invoked at most once in h, and (iii) for every response $res_i(c)\,v$ there exists an invocation $inv_i(c)$ that precedes it in h.[1] A well-formed history h is *legal* if for every service S, $h|S$ is both complete and sequential, and denoting $\langle c_1, \ldots, c_{n\geq 1}\rangle$ the sequence of commands appearing in $h|S$, if for some command c_k a response value appears in $h|S$, it equals $\tau^+(s^0, \langle c_1, \ldots c_k\rangle).val$.

b) Linearizability: Two histories h and h' are said *equivalent* if they contain the same set of events. Given a service S and a history h of S, h is *linearizable* [5] if it can be extended (by appending zero or more responses) to some complete history h' equivalent to a legal and sequential history l of S with $<_{h'} \subseteq <_l$. In such a case, history l is named a *linearization* of h. For instance, the history h_1 above is linearizable since it is equivalent to a sequential one in which $d(10)$ occurs before $w(5)$. The histories of a service S are all the linearizable histories that are constructible with the commands of S. A service S *implements* a service T when for every linearizable history h of S, there exists a linearizable history h' of T such that h' is a high-level view of h [6].[2]

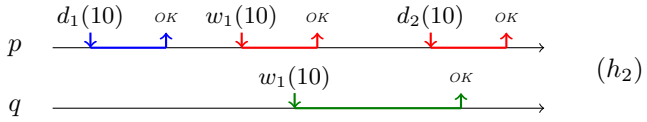
c) Partition: Given a finite family of services $(S_k)_{1\leq k\leq n}$, the synchronized product of $(S_k)_k$ is the service defined by $(\prod_k States_k, (s_1^0, \ldots, s_n^0), \bigcup_k Cmd_k, \bigcup_k Values_k, \tau)$ where for every state $s = (s_1, \ldots, s_n)$ and every command c in some Cmd_k, the transition function τ is given by $\tau(s,c) = ((s_1, \ldots, \tau_k(s_k,c).st, \ldots, s_n), \tau_k(s_k,c).val)$. Given a service S, the family $(S_k)_{1\leq k\leq n}$ is a *partition* of S when its synchronized product satisfies(i) $States \subset \prod_k States_k$, (ii) $s^0 = (s_1^0, \ldots, s_n^0)$, and (iii) for every command c, there

[1] For some service or client x, $h|x$ is the projection of history h over x.

[2] A high-level view is generally constructed via a refinement mapping from the states of S to the states of T [7].

exists a unique sequence σ in $\bigcup_k Cmd_k$, named the *sub-commands* of c, satisfying $\sigma = c$. The partition $(S_k)_k$ is said *consistent* when it implements S.

To illustrate the notion of partition, let us go back to our banking example. A simplistic bank service allows its clients to withdraw and deposit money on an account, and to transfer money between two accounts. We can partition this service into a set of branches, each holding one or more accounts. A transfer of an amount x between accounts i and j is modeled as the sequence of sub-commands $\langle w_i(x).d_j(x)\rangle$, where $w_i(x)$ and $d_j(x)$ are respectively a withdrawal and a deposit of the amount x on the appropriate branch. However, precautions must be taken when concurrent commands occur on the partitioned service. For instance, the following history should be forbidden by the concurrency control mechanism to avoid money creation (concurrent withdrawals cannot both succeed as the balance is not sufficient).



In the section that follows, we characterize precisely when the partition of a service is correct.

B. Partitioning Theorems

When there is no invariant across the partition and every command is a valid sub-command for one of its parts, the partition is *strict*. We first establish that a strict partition is always consistent.

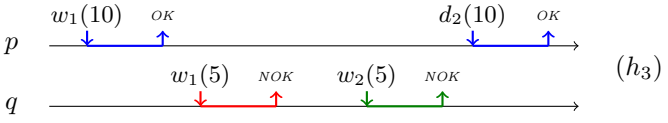
Theorem 1. *Consider a service S and a partition $(S_k)_k$ of S. If both $\prod_k States_k = States$ and $Cmd = \bigcup_k Cmd_k$ hold then $(S_k)_k$ is a consistent partition of S.*

The above theorem is named the locality property of linearizability [5]. It states that the product of linearizable implementations of a set of services is linearizable. From the perspective of service partitioning, this suggests to implement each part as a replicated state machine. Such an idea forms the basic building block of our protocols.

When commands contain several sub-commands, Theorem 1 does not hold anymore. Nevertheless, it is possible to state a similar result when constraining the order in which sub-commands interleave. This is the observation made by Marandi et al. [3], despite a small omission in the original paper. Below, we show where the error occurs and propose a corrected and extended formulation. To state our results, we first need to introduce the notion of conflict graph.

Definition 1. *(Conflict Graph) Consider a history h of a partition $(S_k)_k$ of some service S. The* conflict graph *of S induced by h is the graph $G_h = (V, E)$ such that V contains the set of commands executed in h, and E is the set of pairs of commands (c, d) with $c \neq d$, for which there exist two sub-commands c_i and d_j executed on some S_k such that $c_i <_{h|S_k} d_j$.*

In [3], the authors claim that the partition $(S_k)_k$ is consistent, provided that h is linearizable for each part S_k and G_h is acyclic. Unfortunately, this characterization is incorrect because G_h does not take into account the causality with which commands are executed in h. We argue this point with a counter-example. Let us consider again that our banking service is partitioned in a set of branches. Clients p and q execute the three commands $\langle w_1(10), d_2(10)\rangle$, $w_1(5)$ and $w_2(5)$ as in history h_3 where account 1 is initially provisioned with the amount 10.



Since both withdrawals of client q fail, history h_3 is not linearizable. However, G_{h_3} remains acyclic since it does not capture that process q creates the order $w_1(5) <_{h_3} w_2(5)$.

In what follows, we prove an extended and corrected formulation of the partitioning result of Marandi et al. [3]. Our characterization is based on the notion of semantic graph that we define next.

Definition 2. *(Semantic Graph) Consider a history h of a partition $(S_k)_k$ of some service S. The* semantic graph *of S induced by h is the graph $G_h = (V, E)$ such that V contains the set of commands that appear in h and E is the set of pairs (c, d), with $c \neq d$, for which either(i) there exist two non-commuting sub-commands c_i and d_j in some part S_k such that $c_i <_{h|S_k} d_j$, or (ii) $c <_h d$, where we note $c <_h d$ when all the sub-commands of c precede all the sub-commands of d in history h.*

In contrast to the notion of conflict graph, a semantic graph takes into account the commutativity of sub-commands. To understand why, assume that the banking service allows unlimited overdraft. In such a case, any interleaving of the sub-commands would produce a linearizable history. The partitioning theorem that follows generalizes this observation. It states that the acyclicity of non-commuting sub-commands in the semantic graph is a sufficient condition to attain consistency. A proof appears in Appendix A.

Theorem 2. *A partition $(S_k)_k$ of a service S is consistent if for every history h of $(S_k)_k$, there exists some linearization l of h such that the semantic graph of S induced by l is acyclic.*

III. Protocols

Building upon our previous theorems, this section describes several constructions to partition a shared service. Our presentation follows a refinement process. We start with an initial construction requiring that strictly disjoint services form the partition, then we introduce a more general technique that can accommodate with any type of partitioning. Our last construction improves parallelism at the cost of constraining how the partition is structured. To ease the presentation of our algorithms, we shall be assuming hereafter that sub-commands are idempotent and that no two sub-commands

Algorithm 1 Base construction – code at client p

```
1: invoke(c) :=
2:     let S_k such that c ∈ Cmd_k
3:     return M(S_k).invoke(c)
```

in the same command access the same part. Nevertheless, all of our algorithms can be easily adapted to handle the cases where such properties do not hold.

A. Initial Construction

We depict in Algorithm 1 a first construction when the partition $(S_k)_k$ of service S is strict. This algorithm makes use of a mapping $\mathcal{M}$ satisfying that for every S_k, $\mathcal{M}(S_k)$ is a replicated state machine implementing S_k. When a client p executes a command c on S, it uses $\mathcal{M}$ to retrieve the state machine implementing S_k, where S_k is the service on which c executes (line 2). Then, client p invokes the command on $\mathcal{M}(S_k)$ and returns the result of this invocation (line 3).

Since $(S_k)_k$ is strict and $\mathcal{M}(S_k)$ is a linearizable implementation of S_k, Algorithm 1 implements a consistent partition of S by Theorem 1. Besides, the implementation of $(S_k)_k$ obtained through Algorithm 1 is wait-free [8]. This property is inherited from the underlying replicated state machines that support Algorithm 1. In addition, this base construction is optimal in terms of scalability since, when clients access uniformly the parts, the throughput of the partitioned service is $|(S_k)_k|$ times the throughput of S.

B. A Queue-based Construction

In what follows, we refine Algorithm 1 to handle the case where multiple sub-commands compose a command. A naive solution would consist in modifying Algorithm 1 so that when the client process p executes a command $c = \langle c_1, \ldots, c_n \rangle$, it applies in order all the sub-commands $c_1, \ldots, c_n$ to the appropriate part. Such an approach however fails since(i) an invariant may link different parts of the partition, and (ii) if client p crashes in the middle of its execution, not all the parts will reflect the effects of command c.

Algorithm 2 depicts a solution to deal with these two issues. This algorithm ensures that either all the sub-commands of a command execute, or none of them, and that the state of the partitioned service remains consistent. It is based on a shared FIFO queue abstraction (variable Q) and an eventual leader election service (variable Ω). Clients use Q to submit the commands they wish to execute. Submitted commands are then executed in the queue order by the leader elected by Ω.

With more details, our algorithm works as follows. Upon invoking a command $c = \langle c_1, \ldots, c_n \rangle$, a client p appends c to the queue Q, then it starts participating in the leader election (line 8). In case p is elected, it processes the commands in Q (line 15). For each such command d, p executes all the sub-commands of d once every non-commuting command before d has been executed (line 15). The result of the last sub-command of d is stored as the response of d in the queue Q (lines 17 to 20). This pattern is repeated until the leader, which might not be p, executes command c.

Algorithm 2 Queue-based construction – code at client p

```
1: Shared Variables:
2:     Ω                                          // a leader election
3:     Q                                          // an atomic queue
4:
5: invoke(c) :=
6:     r ← ⊥
7:     Q ← Q ∘ (c, r)
8:     Ω.register()
9:     wait until r ≠ ⊥
10:    Ω.unregister()
11:    Q ← Q \ (c, r)
12:    return r
13:
14: when p = Ω.leader()
15:    let (d, r') ∈ Q : ∀(e, r̂) <_Q (d, r') : r̂ ≠ ⊥ ∨ d ≍ e
16:    let d_1, ..., d_m : d = ⟨d_1, ..., d_m⟩
17:    for all j ∈ [[1, m]] do
18:        let S_k : d_j ∈ Cmd_k
19:        r'' ← M(S_k).invoke(d_j)
20:    r' ← r''
```

The leader election service Ω allows a process to register (line 8) and to unregister (line 10). This service satisfies that eventually(i) only registered processes are elected, and (ii) at least one correct process considers itself as the leader. We note here that property (ii) was previously mentioned in [9], and that Ω is a form of restricted leader election [10]. This makes Ω strictly weaker than the leader oracle used in consensus [11].

C. Ensuring Disjoint Access Parallelism

Both the protocol of Marandi et al. [3] and Algorithm 2 order submitted commands through some global shared object: an instance of the Ring Paxos protocol in the case of [3], and a shared queue for Algorithm 2. As a consequence, the synchronization cost of executing a command is related to the number of concurrent commands. This defeats the primary goal of partitioning which is to scale-up the service by leveraging parallelism for commands that access different parts of the service. Such a property is named *disjoint-access parallelism* (DAP) in the literature on shared memory computing [12]. In what follows, we depict a refinement of our previous algorithm that ensures the following DAP property:

Definition 3 (Disjoint-Access Parallelism)**.** *Consider an algorithm $\mathcal{A}$ implementing a partition $(S_k)_k$ of some service S. We say that $\mathcal{A}$ is* disjoint-access parallel *when in each of its histories h there exists a linearization l of h, such that if p and q concurrently executing commands c and d contend on some shared object in $\mathcal{A}$, there exists a non-directed path linking c to d in the conflict graph of l.*

Algorithm 3 depicts our construction of a DAP consistent partitioning. For each part S_k, we assign respectively a queue $Q[k]$ and a leader election $\Omega[k]$. When a client p invokes a command $c = \langle c_1, \ldots, c_m \rangle$, it iteratively executes each sub-command c_i on the appropriate replicated state machine. To that end, client p adds c_i to the queue $Q[k]$ and then joins leader election $\Omega[k]$ to execute all the sub-commands in $Q[k]$. The helping mechanism in Algorithm 3 is similar to the one we employed in Algorithm 2: when p is the leader and a sub-

Algorithm 3 DAP construction – code at client p

```
1: Shared Variables:
2:     Ω                                   // an array of leader election objects
3:     Q                                   // an array of atomic queues
4:
5: invoke(c) :=
6:     return invoke_sub(first(c))
7:
8: invoke_sub(c_i) :=
9:     r ← ⊥
10:    let S_k : c_i ∈ Cmd_k
11:    Q[k] ← Q[k] ∘ (c_i, r)
12:    Ω[k].register()
13:    wait until r ≠ ⊥
14:    Ω[k].unregister()
15:    Q[k] ← Q[k] \ (c_i, r)
16:    return r
17:
18: when p = Ω[l].leader()                 // for some l
19:    let (d_j, r') ∈ Q[l] : ∀(e_j', r̂) <_Q[l] (d_j, r') : r̂ ≠ ⊥ ∨ d_j ≍ e_j'
20:    r'' ← M(S_l).invoke(d_j)
21:    if d_j ≠ last(d) then
22:        r'' ← invoke_sub(d_j+1)
23:    r' ← r''
```

command d_j occurs before c_i in the queue $Q[k]$, p must first execute d_j as well as the sub-commands following it, before it can execute c_i (lines 18 to 23). This pattern ensures the correctness of our construction in the case where the following property holds:

(P1) There exists an ordering $\ll$ of $(S_k)_k$ such that for any two sub-commands c_i and c_j accessing respectively parts S_k and $S_{k'}$, if c_i precedes c_j in c then $S_k \ll S_{k'}$ holds.

Unfortunately, property P1 does not hold for every partition $(S_k)_k$ of a service S. For instance in our previous banking example, P1 only holds if money transfers between accounts in different branches occur in some canonical order: e.g., $\langle w_i(x), d_j(x)\rangle$ is allowed if and only if $i < j$ holds.

Our key observation is that we can nevertheless enforce the acyclicity of the semantic graph by implementing the partition in a hierarchical manner. We achieve this via two modifications to Algorithm 3. First, we replace P1 by the fact that:

(P2) Function $\mathcal{M}$ returns a set of replicated state machines for each S_k such that for any two sub-commands c_i and c_j accessing respectively parts S_k and $S_{k'}$, if c_i precedes c_j in c then either $\mathcal{M}(S_k) \subseteq \mathcal{M}(S_{k'})$ or the converse holds.

Second, upon executing a sub-command c_i (at line 20 in Algorithm 3), we apply c_i in some canonical order to all the replicated state machines in $\mathcal{M}(S_k)$ before returning the value r''. These two modifications ensure that the premises of Theorem 2 hold for any partition of some shared service S. We sketch a proof of correctness for Algorithm 3 in Appendix B.

Going back to the design of a partitioned banking service, applying P2 requires the addition of a special account t replicated at all branches, such that when a money transfer occurs between two accounts in different branches, money goes through account t, i.e., $\langle w_i(x), d_t(x).w_t(x).d_j(x)\rangle$.

Algorithm 3 with property P2 is the general method we employ to partition a service. We implemented it in ZooFence where we partition the shared tree interface exposed by the Apache Zookeeper coordination service. By partitioning the tree, ZooFence reduces contention and leverages the locality of operations. Both effects contribute to improve latency of operations and increase overall throughput. We describe ZooFence in detail in the next section.

IV. The ZooFence Service

In this section, we present an application of our principled partitioning approach to the popular Apache ZooKeeper [4] coordination service. The resulting system, named ZooFence, orchestrates several independent instances of ZooKeeper. The use of ZooFence is transparent to applications: it offers the exact same semantics and API as a single instance of ZooKeeper. However, the design of ZooFence allows avoiding synchronization between parts when it is not necessary. This reduces the impact of convoy effects that synchronization causes [13].

The partitioning of the ZooKeeper service follows the approach introduced in Algorithm 3. ZooFence splits the tree structure between multiple ZooKeeper instances. Commands that access distinct parts of the tree run in parallel on distinct instances, while guaranteeing both strong consistency and wait-freedom. Commands that access a single part of the tree run on a single instance. This section presents the main components of ZooFence, discusses our design choices with regard to Algorithm 3, then details some specific aspects of its implementation. We give a high-level specification of ZooFence in Appendix C.

A. Overview

Figure 2 depicts the general architecture of ZooFence. The system has four components:(i) A set of independent ZooKeeper instances that ZooFence orchestrates; (ii) A client-side library; (iii) A set of queues storing commands that need to be executed on multiple instances; and (iv) A set of executors that fetch commands from the queues, delegate them to the appropriate instances, and return the result to the calling clients.

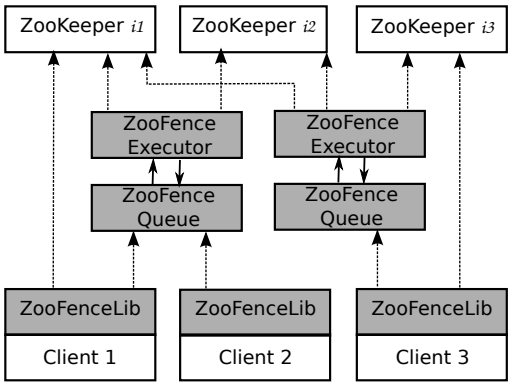


Fig. 2. ZooFence design.

B. Client-Side Library

ZooFence clients execute commands through a client-side library that implements the ZooKeeper API. This interface

consists of a set of commands accessing a concurrent tree data structure composed of *znodes*. A znode in the tree stores some data, and is accessible via a *path* as in UNIX filesystems. Znodes can be *persistent*, meaning they belong to the tree until a client explicitly deletes them, or *ephemeral*, in which case they are automatically removed once the client that created them disconnects or crashes. In addition, znodes can be *sequential*. For such znodes, the system automatically appends a monotonically increasing counter to their names at creation time. A client can manipulate a znode through read or write commands. Read commands, e.g., `exists`, `getChildren` or `getData` return a sub-state of the shared tree without modifying it. Write commands such as `create`, `delete` or `setData`, modify the state of the tree and return metadata information to the client.

Clients execute a command on one or more of the orchestrated ZooKeeper instances according to a *flattening function*. This function plays the role of function $\mathcal{M}$ in Algorithm 3 and maps paths to ZooKeeper instances. Multiple flattening functions can be used. The choice of a flattening function depends on how the application accesses the concurrent tree structure. When a client executes a command c on a znode n, ZooFence determines, using the flattening function, the set of ZooKeeper instances $I = f(n)$ on which c is executed. Command c is *trivial* when $f(n)$ returns a single ZooKeeper instance. In such a case, the command is directly forwarded to that instance. If c is *non-trivial*, it is inserted into the appropriate queue. The executor associated to the queue forwards the command to the corresponding ZooKeeper instances, then returns the result to the client.

The flattening function f satisfies property P2. This means that if a znode n with parent p is mapped to a set of instances $I = f(n)$ then I is also a subset of $f(p)$. To understand why, consider that znode `/a` is mapped to instances $\{i_1, i_2, i_3\}$, and `/a/b` to $\{i_1, i_2\}$. In case a client executes `create(/a/b)`, because both instances i_1 and i_2 hold a copy of `/a`, the creation of `/a/b` succeeds if and only if `/a` was created previously. This ensures that a znode n with parent p is in the tree if and only if p also exists in the tree. The next section provides additional details on the internals of ZooFence, explaining how we maintain this key invariant.

C. Executor

The core notion of ZooFence is the executor. Each executor implements the logic of the sub-commands execution mechanism we depicted at lines 18 to 23 in Algorithm 3. There is one executor for each set of ZooKeeper instances replicating a common path. For instance, in the above example, there is one executor for `/a`, replicated at $\{i_1, i_2, i_3\}$ and one for `/a/b`, replicated at $\{i_1, i_2\}$. As explained previously, each executor is associated with a FIFO queue. When a client executes a non-trivial command, it adds that command to the corresponding queue according to the flattening function. The executor scans the queue in order to retrieve the next command c it has to execute. Then, it forwards c to all the associated ZooKeeper instances, merges their results, and sends the final result back to the client before deleting c from the queue.

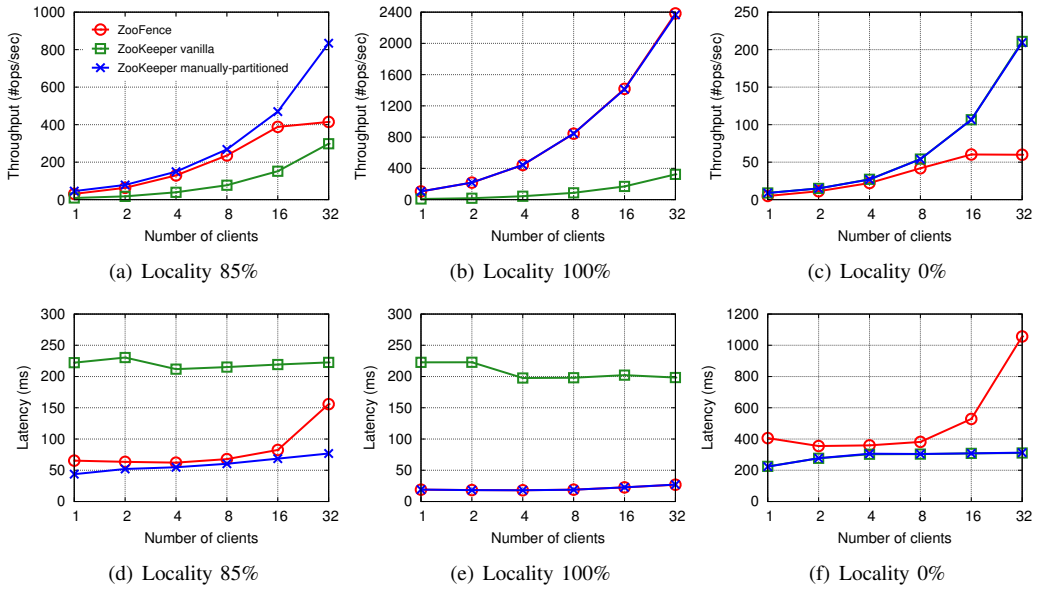
d) Queue synchronization: Using multiple executors improves the performance of ZooFence, but requires additional synchronization. To illustrate this point, let us consider again that `/a` is replicated at $\{i_1, i_2, i_3\}$ and `/a/b` at $\{i_1, i_2\}$. The set of instances associated with znode `/a/b` has a smaller cardinality than the set of instances associated with its parent, `/a`; we call `/a/b` a *fringe* znode. Assume that a client attempts to delete `/a`, while another client concurrently attempts to delete `/a/b`. Due to the tree invariant, the deletion of `/a` succeeds only if it does not have any children. In the scenario above, if the deletion of znode `/a/b` finishes on i_1, but not on i_2, before the deletion of znode `/a` is executed, then `/a` would be deleted from i_1, but not from i_2, leaving replicas in an inconsistent state. We solve the problems related to fringe znodes by synchronizing queues following the approach depicted at lines 21 to 22 in Algorithm 3: Upon creation of such a znode, the executor adds the command to the parent queue. Upon deletion, the executor first executes the command then, in case of success, it adds the command to the parent queue. When adding a command to the parent queue, the executor waits for a result before returning.

e) Failure Recovery Mechanism: The executor is a dependable component of ZooFence. To ensure this guarantee, we replicate each executor and employ the same leader election mechanism as in Algorithm 3. When the previously elected executor is unresponsive or has crashed, ZooFence nominates a new one and resumes the execution of commands. ZooFence prevents inconsistencies that might result, as follows: (i) We ensure idempotency at the client side; and (ii) We use the command semantics to resume incomplete commands on the associated ZooKeeper instances.

f) Queue monitoring: Executors retrieve commands from their respective queues and keep them in a local cache. Instead of actively polling the queue for new commands, executors use the ZooKeeper event notification mechanism, watches, which are triggered when the queues are modified. Executors check their local caches every time the watch set on their queue is triggered; if the cache is empty, the executor performs a `getChildren` command to repopulate its cache.

g) Asynchronous commands: Each executor retrieves commands from its local cache and forwards them to its set of ZooKeeper instances. We use asynchronous commands between an executor and its ZooKeeper instances, regardless of the type of command issued by the ZooFence client. The only difference between synchronous and asynchronous client commands is local, in the ZooFence client library; for synchronous commands, the library blocks and waits for the reply to arrive. Since ZooKeeper guarantees FIFO order even for asynchronous commands, this optimization which is transparent to the clients, increases the throughput of the executor without changing the semantics of synchronous

h) Colocation: An executor is a stand-alone process of ZooFence that runs on a different machine than the clients. To reduce network latency for queue accesses, an executor usually



(a) Locality 85% (b) Locality 100% (c) Locality 0%

(d) Locality 85% (e) Locality 100% (f) Locality 0%

Fig. 3. Comparison of ZooFence against a vanilla and a manually-partitioned ZooKeeper.

executes at the same site as the ZooKeeper instance that hosts its corresponding queue.

D. Implementation Details

Our prototype implementation of ZooFence is written in Java and contains around 2,500 SLOC. It is built upon the out-of-the-box ZooKeeper 3.4.5 distribution. Clients transparently instantiate a ZooFence deployment when the connection string contains multiple ZooKeeper instances separated by a "|" character, e.g., "127.0.0.1:2181 | 127.0.0.1:2182". As discussed in Section IV-B, ZooFence clients forward non-trivial commands to executors via queues. Client commands are serialized and stored in *persistent sequential znodes*; we use the *sequential* attribute offered by ZooKeeper to assign a sequence number to each command in the queue. Queues are stored in dedicated administrative ZooKeepers. Executors are identified by *ephemeral znodes*, also stored in administrative ZooKeepers. The choice of the administrative ZooKeepers among the existing instances is currently not automated and left as a future work. Clients open TCP connections to executors before putting commands in queues. When commands complete, executors send the results over the respective connections.

V. Evaluation

We evaluate our ZooFence prototype in two scenarios: a synthetic concurrent queue service deployed at multiple geographically-distributed sites, and the BookKeeper distributed logging service [14]. We compare ZooFence against vanilla ZooKeeper deployments in terms of throughput, latency and scalability.

A. Concurrent Queues Service

In this experiment, we show that ZooFence can leverage access patterns locality to improve both throughput and responsiveness of distributed applications. To that end, we emulate a geographically-distributed deployment. Each site consists of dual-core machines with 2 GB RAM, and hosts one ZooKeeper instance and several clients. Inside a site, machines communicate using a (native) gigabit network. Between sites, we set up the round-trip delay to approximately 50 ms. Our experiment uses a single executor, collocated with the administrative ZooKeeper on an eight-core machine at a different site than all clients.

In this environment, we deploy a service consisting of five concurrent queues. Four of the queues exhibit strong locality, and are used exclusively by clients from the same site (i.e., *queue1* is used only by clients from *site1*, *queue2* from *site2*, etc.). We refer to these queues as *site queues*. The fifth queue is used by clients from all sites. We refer to it as the *geo-distributed queue*. We vary the number of clients within each site from one to eight. Each client runs two producer processes. Producers mostly create znodes in their site queue, but occasionally write to the geo-distributed queue as well. We note that this is a write-heavy workload, which represents a worst-case scenario for both ZooFence and ZooKeeper.

Our experiments compare in terms of performance three different deployments:(1) ZooFence with a flattening function that assigns site queues to local ZooKeeper instances, and the geo-distributed queue to all instances. (2) A vanilla ZooKeeper, which involves all the available ZooKeeper instances: a leader and three followers, with synchronization bound set to 175 ms. All queues are stored by this ZooKeeper instance. This is the baseline for our experiments – how ZooKeeper would be used in the present. (3) A manually-partitioned ZooKeeper. Each machine runs two ZooKeeper instances: one instance stores the site queue, and the other one stores the geo-distributed queue; the latter is a ZooKeeper instance covering all sites. This deployment is the optimum an experienced ZooKeeper administrator can achieve: writes accessing the geo-distributed queue are broadcast to all sites, otherwise they are served locally.

Figure 3 presents the latency and the throughput of the

system for each of the three deployments when we vary the number of clients and the percentage of operations on the geo-distributed queue.

As shown by figures 3(a) and 3(d), with a locality of 85%, ZooFence is close to the manually-partitioned ZooKeeper. This happens because in both deployments most operations occur inside a site, avoiding the cross-site communication in most cases. The vanilla ZooKeeper deployment exhibits lower throughput and higher latency because all queues are replicated across all sites. Since the leader ZooKeeper propagates updates to its followers, the inter-site communication penalty cannot be avoided.

When locality is maximum, ZooFence is identical to the manually-partitioned ZooKeeper (figures 3(b) and 3(e)). The vanilla ZooKeeper deployment performs significantly worse because it fully replicates all znodes.

Finally, when all operations are performed on the geo-distributed queue (figures 3(c) and 3(f)), the performance of ZooFence becomes worse than that of the other two deployments, both in terms of throughput and latency. This is due to the overhead incurred by our execution mechanism for operations on replicated znodes: the executor fetches commands from the execution queue, delegates them to the responsible ZooKeeper instances based on the flattening function and forwards the result to the client.

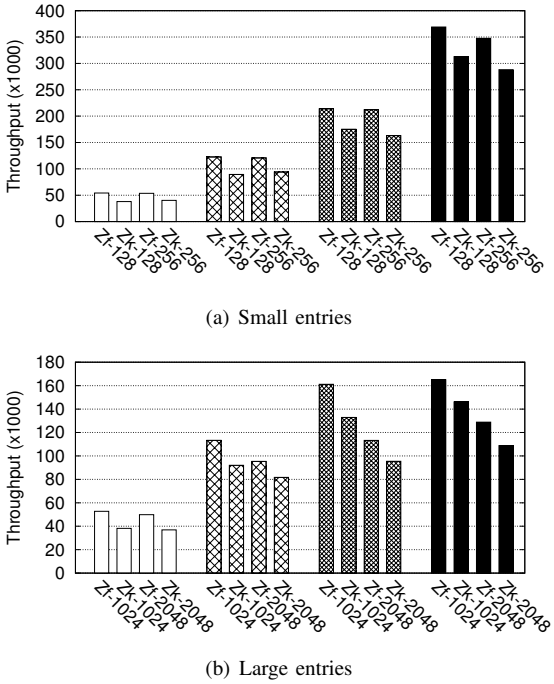
The executor itself can become saturated and degrade performance, in terms of both throughput and latency. In this experiment, we used a single executor, deployed on the same VM as the one hosting the administrative ZooKeeper. The executor becomes saturated at around 60 operations per second, as shown in Figure 3(c) and Figure 3(a) (only 15% of operations go through the executor, and 15% * 400 = 60). This is mainly due to our design choice of making ZooFence modular, on top of ZooKeeper. This choice implies that our system does not benefit from optimizations such as batching, which require tighter integration. Since the executor is in a separate site, it pays the latency penalty two times for global operations by communicating with the ZooKeeper instances and the clients, which limits performance.

We have performed a similar experiment for read workloads, where conclusions are similar. ZooFence allows local reads to exhibit low latency, and not interfere with the performance of remote partitions.

Overall, our experiments show that ZooFence performs close to an optimal deployment when access patterns exhibit strong locality. ZooFence can enable even inexperienced administrators to obtain good performance without the burden of partitioning the state manually.

B. BookKeeper

This section presents experimental results that assess the performance gains of ZooFence over ZooKeeper. To that end, we compare the two systems in a cluster deployment when supporting the BookKeeper logging service [14]. Below, we first give a brief description of BookKeeper, then detail our experimental protocol and comment on our results.



(a) Small entries

(b) Large entries

Fig. 4. BookKeeper performance *(from left to right, the amount of written entries is 100, 250, 500 and 1000).*

BookKeeper is a distributed system that reliably stores streams of records. Each stream is stored in a write ahead log, or *ledger*, that contains multiple *entries*. A ledger is replicated at one or more dedicated storage servers, named *bookies*. BookKeeper uses the ZooKeeper tree structure to stores the metadata of the ledgers, as well as the set of available bookies. The tree also ensures that a single client writes to a ledger at a time. BookKeeper supports close-to-open semantics: when writing to a ledger, a client appends new entries at a quorum of its replicas. Changes to an open ledger are visible to a concurrent client only when the ledger is closed.

Our experiments consist in evaluating the maximal throughput of BookKeeper when clients concurrently open ledgers and start appending entries to them. In all our experiments, we employ 18 bookies with a replication factor of 3. Such a setting ensures that the system can sustain the failure of one bookie without interruption. We vary both the length of an entry and the frequency at which clients create a new ledger. This last parameter is controlled by the number of entries each client writes to a ledger before closing it.

Figures 4(a) and 4(b) present a detailed performance comparison of BookKeeper when using ZooKeeper and ZooFence. We vary the length of an entry from 128 to 2048 bits, and the number of written entries from 100 to 1000. In both figures, Zk stands for ZooKeeper, while Zf means ZooFence; the throughput is measured as the total amount of operations per second. When clients write 1000 entries (or more, not shown on the plots), the two systems achieve close performance. In such a case, the throughput is limited by either the bookies or the network. During our experiments, the MTU is set to 1500 bytes. This explains the performance gap between large and small entries. On the other hand, when clients open concurrently more

ledgers, fast operations on the metadata storage matter. In such a case, because ZooFence provides parallel accesses to the shared tree, it outperforms ZooKeeper. The difference increases as clients access new ledgers more frequently. In our experiments, ZooFence improves the throughput of BookKeeper by up to 45%.

VI. Related Work

Coordination services find their roots in pioneering works on locks and synchronization primitives [15]. The need for coordination spreads over multiple areas, from distributed databases [16], file stores [17], multicore systems [18] and Cloud computing [19].

Several paradigms exist for coordinating distributed processes in the Cloud. Microsoft Azure [20] exports a common lock interface. Clients of Google Chubby [21] make use of a lease mechanism to gain exclusive access to a shared resource. The API of Apache ZooKeeper [4] consists in a concurrent tree data structure, close to UNIX filesystems. All these three systems back-up committed operations in a replay log. The Corfu system [22] implements an atomic log on top of dedicated hardware. With Tango [23], developers have access to any type of strongly consistent in-memory object, e.g., queue, map. Each Tango object is backed by the Corfu log.

Coordination is closely related to dependability. Indeed, redundancy is the usual mean to mask failures, but replicas of a service need to coordinate in order to implement a consistent service execution. State machine replication (SMR) is the seminal approach [24] to build dependable distributed services. It allows a set of replicas to agree on the order in which they execute service operations. SMR relies on a consensus algorithm, e.g., Paxos [25], and this approach is at heart of the Cloud systems we reviewed above. Several recent works (e.g., [26, 27]) observe that there is no need to order commuting service operations. This observation was used recently to build a distributed database system [28].

The seminal CAP result [29] states that a system cannot be at the same time responsive, consistent and robust to network outages. In the same vein as the SMR approach, ZooFence favors consistency over availability. This means that availability can be forfeited in the presence of network faults. For instance, if one of the ZooKeeper instances implementing ZooFence cannot progress, commands accessing the data it replicates are frozen until the instance come-back.

The virtual synchrony paradigm [30] is close, but slightly different from SMR. Under this paradigm, distributed processes execute a sequence of views, agreeing in each view upon the participants and the set of received messages (but not on their order). Virtual synchrony is used to build the ISIS middleware and its successors [31].

Some approaches [32, 33] leverage application semantics to compute dependencies among commands in order to parallelize SMR. Commands are assigned to groups such that commands within a group are unlikely to interfere with each other. Eve [33] allows replicas to execute commands from different groups in parallel, verifies if replicas can reach agreement on state and output, and rolls back replicas if necessary. Parallel State-Machine Replication (P-SMR) [32] proposes to parallelize both the execution and the delivery of commands by using several multicast groups that partially order commands across replicas. These two techniques aim at speeding-up the execution of commands at each replica by enabling parallel execution of commands on multi-core systems. Our approach is orthogonal to this body of work. It improves performance by enabling different replicated state machines to execute commands in parallel without agreement.

Marandi et al. [3] employ Multi-Ring Paxos to implement consistent accesses to disjoint parts of a shared data structure. However, by construction, if an invariant is maintained between two or more partitions, the approach requires that a process receives all the messages addressed to the groups, defeating the purpose of DAP. Oster et al. [34] and Preguiça et al. [35] construct a shared tree in a purely asynchronous system under strong eventual consistency. In both cases however, the tree structure is replicated at all replicas.

Concurrently to our work, Bezerra et al. [36] have described recently an approach to partition a shared service. To execute a command, the client multicasts it to the partitions in charge of the state variables read or updated by that command. Each partition executes its part of the command, waiting (if necessary) for the results of other partitions. In comparison to our approach, this solution(i) does not take into account the application semantics implying in some cases an unnecessary convoy effect, and (ii) it requires to approximate *in advance* the range of partitions touched by the command. In the ZooKeeper use case (Volery), P-SMR stores the tree at all replicas and commands modifying the structure of the tree (`create` and `delete`) are sent to all replicas. In contrast, ZooFence exploits application semantics to split the tree into overlapping sub-trees, one stored at each partition.

The transactional paradigm is a natural candidate to partition a concurrent tree (e.g., [37]). In ZooFence, there is no need for transactional semantics because the implementation of the tree is hierarchical. A transactional history is serializable when its serialization graph is acyclic [16]. Theorem 2 can be viewed as the characterization of strictly serializable histories over abstract operations.

Ellen et al. [38] prove that no universal construction can be both DAP and wait-free in the case where the implemented shared object can grow arbitrarily. We pragmatically sidestep this impossibility result in our algorithms by bounding the size of the partition.

VII. Conclusion

This paper presents ZooFence, a system that automatically partitions ZooKeeper. ZooFence reduces contention and increases the overall throughput of applications using ZooKeeper, especially in geo-distributed scenarios. This system is based on a principled approach to partition a distributed service that we present in detail. We assess the practicability of a prototype implementation of ZooFence on two benchmarks: a concurrent queue service and the BookKeeper distributed logging engine.

Our experiments show that when locality is optimum, ZooFence improves ZooKeeper performance by almost one order of magnitude.

VIII. Acknowledgments

We are thankful to the anonymous reviewers and our shepherd for their many useful comments. The research leading to this publication was partly funded by the European Commission's FP7 under grant agreement number 318809 (LEADS), as well as by the Swiss National Science Foundation under Sinergia Project No. CRSII2_136318/1.

References

[1] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, May 1994.

[2] P. Sutra and M. Shapiro, "Fast Genuine Generalized Consensus," in *Proceedings of the 30th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, Madrid, Spain, Oct. 2011, pp. 255–264.

[3] P. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, ser. DSN, 2011.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX technical conference*, ser. ATC, 2010.

[5] M. Herlihy and J. Wing, "Linearizability: a correcteness condition for concurrent objects," *ACM Trans. on Prog. Lang.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[6] L. Lamport, "On interprocess communication. part i: Basic formalism," *Distributed Computing*, vol. 1, no. 2, 1986.

[7] M. Abadi and L. Lamport, "The existence of refinement mappings," in *3rd Annual IEEE Symposium on Logic in Computer Science*, ser. LICS, July 1988.

[8] M. Herlihy, "Wait-free synchronization," *ACM Trans. on Prog. Lang. and Systems*, vol. 11, no. 1, Jan. 1991.

[9] P. Sutra and M. Shapiro, "Fault-tolerant partial replication in large-scale database systems," in *European Conf. on Parallel Computing*, ser. Euro-Par, 2008.

[10] M. Raynal and J. Stainer, "From a store-collect object and Ω to efficient asynchronous consensus," in *European Conf. on Parallel Computing*, ser. Euro-Par, 2012.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[12] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *13th Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC, 1994.

[13] K. Birman, G. Chockler, and R. van Renesse, "Toward a Cloud Computing research agenda," *ACM SIGACT News*, vol. 40, no. 2, pp. 68–80, Jun. 2009.

[14] Apache Software Foundation, "Apache Bookkeeper," 2013. [Online]. Available: zookeeper.apache.org/bookkeeper

[15] E. W. Dijkstra, "The origin of concurrent programming," P. B. Hansen, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Cooperating Sequential Processes, pp. 65–138.

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

[17] Sun Microsystems, Inc., "NFS: Network file system protocol specification," Network Information Center, SRI International, RFC 1094, Mar. 1989.

[18] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *19th ACM Symposium on Operating Systems Principles*, ser. SOSP, 2003.

[20] B. Calder and *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *23rd ACM Symposium on Operating Systems Principles*, ser. SOSP, 2011.

[21] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *7th symposium on Operating Systems Design and Implementation*, ser. OSDI, 2006.

[22] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "Corfu: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 10:1–10:24, Dec. 2013.

[23] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *24th ACM Symposium on Operating Systems Principles*, ser. SOSP, 2013.

[24] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[25] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[26] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, 2002.

[27] L. Lamport, "Generalized consensus and paxos," Microsoft, Tech. Rep. MSR-TR-2005-33, March 2005.

[28] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *8th ACM European Conference on Computer Systems*, ser. EuroSys, 2013.

[29] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[30] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *11th ACM Symposium on Operating Systems Principles*, ser. SOSP, 1987.

[31] K. P. Birman, "Replication and fault-tolerance in the ISIS system," in *10th ACM Symposium on Operating Systems Principles*, ser. SOSP, Dec. 1985.

[32] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *34th International Conference on Distributed Computing Systems*, ser. ICDCS, July 2014.

[33] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2012.

[34] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *ACM Conference on Computer-Supported Cooperative Work*, ser. CSCW, Nov. 2006.

[35] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *29th International Conference on Distributed Computing Systems*, ser. ICDCS, Montréal, Canada, Jun. 2009, pp. 395–403.

[36] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable state-machine replication," in *45th International Conference on Dependable Systems and Networks*, ser. DSN, June 2014.

[37] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 598–609, Aug. 2008.

[38] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers, "Universal constructions that ensure disjoint-access parallelism and wait-freedom," in *ACM Symposium on Principles of Distributed Computing*, ser. PODC, 2012.

APPENDIX

A. Proof of Theorems 1 and 2

Theorem. *Consider a service S and a partition $(S_k)_k$ of S. If both $\prod_k States_k = States$, and $Cmd = \bigcup_k Cmd_k$ hold then $(S_k)_k$ is a consistent partition of S.*

Proof: Let h be a linearizable history of $(S_k)_k$. For each part S_k, history $h|S_k$ is linearizable. Thus, there exists per S_k a history h'_k completing $h|S_k$ and a sequential history l_k equivalent to h'_k such that l_k is legal for S_k and $<_{h'_k} \subseteq <_{l_k}$ holds.

Name $<$ the transitive closure of the union of $(\bigcup_k <_{l_k})$ and $<_h$. For the sake of contradiction, assume that $<$ is not a partial order over the commands in h. It follows that for some commands $c_1, \ldots, c_{n>2}$, we have a cycle $c_1 < \ldots < c_n < c_1$. Clearly, we cannot have in this cycle neither only orders $<_h$, nor at most one order $<_h$. Thus, consider some i for which we have $c_{i-1[n+1]} <_h c_i <_{l_k} c_{i+1[n+1]} <_h c_{i+2[n+1]}$. Necessarily, c_{i+1} does not precede c_i in h. From which it follows (in a global time model) that $c_{i-1} <_h c_{i+2}$ holds. By applying this reduction, we conclude that there exists a cycle with only one order $<_h$; a contradiction.

We append to h all the responses that are in the histories h'_k and not in h to form H. Since $(S_k)_k$ is a partition of S, H is a history of S. Moreover, every command c is complete in H as it was complete in some h'_k.

We concatenate the commands in $(l_k)_k$ in the order $<$ to form L. By definition of $<$, we have $<_H \subseteq <_L$. History L is by construction sequential. Then, consider that a transition from $s = (s_1, \ldots, s_n)$ to $s' = (s'_1, \ldots, s'_n)$ occurs in L for some command $c \in Cmd_k$ with a response value val. Since $(S_k)_k$ is strict, we have $\tau(s, c) = ((s_1, \ldots, \tau_k(s_k, c).st, \ldots, s_n), \tau_k(s_k, c).val) = (s', val)$. This leads to $(s, c, val, s') \in \tau$. As a consequence, the history L is legal. It follows that L is a linearization of H. ∎

Theorem. *A partition $(S_k)_k$ of a service S is consistent if for every history h of $(S_k)_k$, there exists some linearization l of h such that the semantic graph of S induced by l is acyclic.*

Proof: Let us assume the existence of some history l such that(i) for every S_k, history $l|S_k$ is a linearization of $h|S_k$, and (ii) the semantic graph $G_l = (V, E)$ of S induced by l is acyclic. We have to show that there exists some linearizable history H of S such that H is a high-level view of h.

First, let us define the following mapping $\mathfrak{F}$ from h to a set of invocation and response events: Given a command $c = \langle c_1, \ldots, c_m \rangle$ that appears in h, we map every event e regarding the sub-commands of c, except $res(c_m)$, via $\mathfrak{F}$ to $inv(c)$. Provided that the event $res(c_m)$ exists in h, we map it via $\mathfrak{F}$ to $res(c)$. Then, we consider the following relation $\ll$ on the image of h via $\mathfrak{F}$;

$$e \ll e' = \begin{array}{l}(\exists (c, d) \in E : e = inv(c) \ \wedge \ e' = inv(d)) \\ \vee \ (\mathfrak{F}^{-1}(e) <_h \mathfrak{F}^{-1}(e'))\end{array}$$

Clearly, $\ll^*$ is a partial order on $\mathfrak{F}(h)$. Hence, we can define H as the concatenation of the invocations and response events in $\mathfrak{F}(h)$ following some topological order compatible with $\ll^*$. By construction, H is a higher-level view of h. Moreover, since $(S_k)_k$ is a partition of S, H is a history of S.

Second, we append to h all the responses that are in l and not in h to form h'. Since for every S_k, history $l|S_k$ is a linearization of $h|S_k$, every sub-command in l is complete. As a consequence, history h' completes history h. Then, following some order compatible with E, for every command c incomplete in h', we append to h' some sequential execution of the sub-commands in c missing in h'. Following the algorithm we employed to construct H, we then build H' based on h'. By construction, H' completes H.

Now, let us choose L as some sequential history equivalent to H' and satisfying $<_{H'} \subseteq <_L$. We argue by induction that L is legal for the service S. Consider some command $c = \langle c_1, \ldots c_m \rangle$ returning in L a value val, and assume that $L_{|<c}$ is legal. Note $s = (s_1, \ldots, s_n)$ the final state of S in $L_{|<c}$. Without lack of generality, we assume that $m = |(S_k)_k|$ and that every sub-command c_k executes on a distinct S_k. We have to show that(i) for every k, denoting $\hat{s}_k$ the state of S_k in $l|S_k$ before the execution of c_k, we have $\hat{s}_k = s_k$, as well as, (ii) val is the response returned by c_m in l. We observe that (ii) holds by definition of $\mathfrak{F}$. Then for (i), let d_k be the sub-command of some command d preceding c in L that produces s_k. Observe that for every command e_k such that $d_k <_{l|S_k} e_k <_{l|S_k} c_k$, because $<_L$ is compatible with the order E, d_k and e_k commute. From which we deduce that $\hat{s}_k = s_k$. ∎

B. Correctness of Algorithm 2 and Algorithm 3

In this section, we sketch a correctness proof of Algorithms 2 and 3. To that end, we consider some partition $(S_k)_k$ of a service S. First of all, we observe that since for every part S_k, $\mathcal{M}(S_k)$ is a linearizable implementation of S_k, all our constructions are linearizable implementation of $(S_k)_k$. In the case where $|\mathcal{M}(S_k)| > 1$, this follows from the idempotency of sub-commands and the fact that we apply each sub-command c_i in some canonical order at all the replicated state machines in $\mathcal{M}(S_k)$ before returning the response of c_i.

Algorithm 2. First of all, consider that some client p executes a command c. Since p registers at line 8, by the properties of the leader election service Ω, eventually some correct client process q executes lines 15 to 20 for command c. Then, since every replicated state machine $\mathcal{M}(S_k)$ is wait-free, all the sub-commands of c returns. It follows that eventually p returns from its invocation of c. Besides, we observe that the preconditions at line 15 implies that if two commands are concurrently executed in some execution of Algorithm 2, they must be commuting. As a consequence, the semantics graph produced by any execution of Algorithm 2 is acyclic. Applying Theorem 2, we conclude that the partition implemented by Algorithm 2 is consistent.

Algorithm 3. Consider some linearization l of an history h produced by Algorithm 3. At first glance, assume that any two commands of S executed in l are commuting. It follows that the semantic graph induced by l is acyclic, and we may apply

Theorem 2. Then, assume for the sake of contradiction that G_l contains a cycle $\mathcal{C}$ of non-commuting commands. We note $(S_{k'})_{k'}$ the restriction of $(S_k)_k$ to the parts appearing in cycle $\mathcal{C}$. Then, we consider the two following cases depending on whether property P1 or P2 holds:

(P1.) Consider a sub-command c_i applying on part S_i such that S_i is the smallest part for the order $\ll$ over the parts $(S_{k'})_{k'}$, and c_i creates an order (d, c) or (c, d) in G_l.

(Case (d, c).) The sub-command c_i is preceded by some d_i with $d \in \mathcal{C}$ in l. By definition of c_i, every sub-command preceding c_i in c commutes with all the sub-commands of the commands in $\mathcal{C}$. Hence, by the helping mechanism at lines 18 to 23, every sub-command d_j commutes with the sub-commands of c before c_i and is executed before c_i in l. Then, since some d_j is preceded by a sub-command $e_{j'}$ with $e \in \mathcal{C}$, the very same reasoning tells us that the sub-commands of e commute with the sub-commands before c_i and that they are executed before c_i in l. Hence, by induction, no order $(c, \_)$ occurs in $\mathcal{C}$; a contradiction.

Case (c, d) This case is symmetrical to the previous one, and thus omitted.

(P2.) Assume that both relations (c, d) and (d, e) are in G_l. Note respectively (c_i, d_i) and (d_j, e_j) the pair of sub-commands that created them. In addition, let S_i and S_j be the parts on which respectively (c_i, d_i) and (d_j, e_j) do not commute. By property P2, we have either $\mathcal{M}(S_i) \subseteq \mathcal{M}(S_j)$ or the converse that holds. From which it follows that some replicated state machine linearizes c_i and e_j. By applying this reasoning to the cycle G_l, we obtain the desired contradiction.

Let us now consider that some correct client process p executes a command c. Since no two sub-commands in c access the same part, command c never blocks waiting for itself at line 19 Then, consider that when executing line 19, a client process always returns the first command that satisfies the predicate. In such a case, a short induction on the eventual properties of the leader election services in S together with the fact that G_l is acyclic, tell us that every sub-command eventually returns from its invocation. From which we deduce that Algorithm 3 is wait-free.

Finally, we observe that if two commands c and d access the same replicated state machine in some $\mathcal{M}(S_k)$, there must exist a non-directed path in conflict graph of l. Hence, Algorithm 3 is disjoint-access parallel.

C. High-Level Specification of ZooFence

We consider a tree as an undirected graph $T = (N, E)$ in which any two vertices are connected by exactly one simple path. We note n_0 the root of the tree. Each node n in the tree stores some (initially null) data $n.d$. As usual, we make no distinction between a node and the unique path starting from n_0 that reaches it. For some path p, we note $\mathit{parent}(p)$ the parent of p, and given two paths p and p', we write $p \sqsubseteq p'$ when p prefixes p'. Clients manipulate the tree through the following interface:

- `exist`(x) return true iff x is in T.
- `getChildren`(x) returns the children of x in T.
- `create`(x): if $\mathit{parent}(x)$ exists in T, add x as a child of $\mathit{parent}(x)$ and returns true; otherwise false is returned.
- `delete`(x): if x has no children then deletes it from T and returns true; otherwise returns false.
- `update`(x, d): if node x exists in T, updates its content with data d and returns true; otherwise false is returned.

To partition the above interface, we need to satisfy property P2. We translate this as the fact that if $p \sqsubseteq p'$ then $\mathcal{M}(p') \subseteq \mathcal{M}(p)$. Multiple mappings are possible, and as pointed out in Section IV, such a choice is application-dependent. For instance, let $\kappa, \lambda \in \mathbb{N}$ be some *flattening parameters*. We define $\mathcal{M}(n_0)$ as the set of all the replicated state machines. Then, for every path p of length $|p|$, if $|p| \neq 0\,[\kappa]$, we remove deterministically λ replicated state machines from $\mathcal{M}\mathit{parent}(p)$; otherwise we let $\mathcal{M}(p)$ equals $\mathcal{M}(\mathit{parent}(p))$.

Based on the above assignment of nodes to replicated state machines, we then partition the tree as follows: Each replicated state machine implements a shared tree that exposes the very same interface as the one we described above. Commands at the client level are implemented by the following sequences of commands at the replicated state machine level (underlined):

```
exist(x)       = return M(x).exist(x)
update(x, d)   = return M(x).update(x, d)
getChildren(x) = return M(x).getChildren(x)
create(x)      = return M(parent(x)).create(x)
delete(x)      = if M(x).getChildren(x) ≠ ∅
                   return false
                 M(parent(x)).delete(x)
                 return true
```